Monte Documentation

Release 0.1

Monte Project

December 18, 2018
# Table of Contents

1 Introduction 3  
1.1 Why Monte? 3  
1.2 Object Capability Discipline 4  
1.3 Why not Monte? 5  
1.4 Getting Started 5  
1.5 Acknowledgements 7  

2 A Taste of Monte: Hello Web 9  
2.1 Expressions 10  
2.2 Objects and Message Passing 10  
2.3 Cooperation Without Vulnerability 10  

3 Practical Security: The Mafia game 13  
3.1 Objects 13  
3.2 Traditional Datatypes and Operators 15  
3.3 String Interpolation with quasi-literals 19  
3.4 Dynamic “type checking” with guards 19  
3.5 Final, Var, and DeepFrozen 19  
3.6 Assignment and Equality 20  
3.7 Data Structures for Game Play 20  
3.8 Destructuring with Patterns 21  

4 Monte Idioms Quick Reference 23  
4.1 Simple Statements 23  
4.2 Basic Flow 23  
4.3 File I/O and Modules 25  
4.4 Web Applications 25  
4.5 Data Structures 26  
4.6 Eventual Sends 27  

5 Python-Monte Idioms 29  
5.1 Iteration 29  
5.2 Objects 30  

6 The Type System 33  
6.1 Untyped 33  
6.2 Dynamic 33  
6.3 Strong 33  
6.4 Nominal 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.5 Manifest</td>
<td>34</td>
</tr>
<tr>
<td>6.6 Optional</td>
<td>34</td>
</tr>
<tr>
<td><strong>7 Misuse-Resistant Language Design</strong></td>
<td>35</td>
</tr>
<tr>
<td>7.1 Unicode Identifiers</td>
<td>35</td>
</tr>
<tr>
<td>7.2 Parenthesized Sub-Expressions</td>
<td>35</td>
</tr>
<tr>
<td><strong>8 Secure Distributed Computing</strong></td>
<td>37</td>
</tr>
<tr>
<td>8.1 Practical Security II: The Mafia IRC Bot (WIP)</td>
<td>37</td>
</tr>
<tr>
<td>8.2 Ejectors &amp; Escape Expressions</td>
<td>42</td>
</tr>
<tr>
<td>8.3 Vats</td>
<td>43</td>
</tr>
<tr>
<td>8.4 Brands</td>
<td>47</td>
</tr>
<tr>
<td>8.5 Promises</td>
<td>48</td>
</tr>
<tr>
<td>8.6 Streamcaps</td>
<td>49</td>
</tr>
<tr>
<td>8.7 Working with Packages</td>
<td>51</td>
</tr>
<tr>
<td><strong>9 Language Reference</strong></td>
<td>53</td>
</tr>
<tr>
<td>9.1 Collections</td>
<td>53</td>
</tr>
<tr>
<td>9.2 Quasiliterals</td>
<td>53</td>
</tr>
<tr>
<td>9.3 Interfaces</td>
<td>56</td>
</tr>
<tr>
<td>9.4 Miranda Protocol</td>
<td>57</td>
</tr>
<tr>
<td>9.5 Loops and the Iteration Protocol</td>
<td>59</td>
</tr>
<tr>
<td>9.6 Guard Protocol</td>
<td>61</td>
</tr>
<tr>
<td>9.7 Controllers</td>
<td>62</td>
</tr>
<tr>
<td>9.8 Slots</td>
<td>63</td>
</tr>
<tr>
<td>9.9 Auditors</td>
<td>64</td>
</tr>
<tr>
<td>9.10 Semantics of Monte</td>
<td>66</td>
</tr>
<tr>
<td>9.11 Categorial Semantics</td>
<td>73</td>
</tr>
<tr>
<td><strong>10 Appendixes, Indices and Tables</strong></td>
<td>77</td>
</tr>
<tr>
<td>10.1 Monte Grammar</td>
<td>77</td>
</tr>
<tr>
<td>10.2 Roadmap: Montefesto</td>
<td>80</td>
</tr>
<tr>
<td>10.3 safeScope</td>
<td>83</td>
</tr>
<tr>
<td>10.4 Entrypoint Arguments</td>
<td>112</td>
</tr>
<tr>
<td>10.5 Colophon: Monte Documentation Build Tools</td>
<td>114</td>
</tr>
<tr>
<td>10.6 Glossary</td>
<td>118</td>
</tr>
</tbody>
</table>

**Python Module Index** | 121 |
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. A major cause is the “water balloon” design philosophy of contemporary languages and frameworks: Security is only enforced at the edges of programs, and not within the structure of programs themselves. Monte takes the object-capability paradigm of E and expands on the approach, delivering a powerful and expressive language.

Can we reuse existing languages?

Disciplined use of existing languages such as Java and ECMAScript can be used to build object capability systems, but the standard practices and libraries are not compatible with this discipline. We believe that nothing short of a capability-safe-by-construction language can enforce capability discipline.

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.

• Monte offers cryptographic services directly to its users, easing the use of good cryptographic primitives.

Because Readability Matters

1 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.
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.

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 \(^2\), 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**.

\(^2\) in formal type theory, Monte is untyped.
**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.

---

**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, RPM, etc.

**From Source**

Builds of Monte from source are straightforward, using Nix:

```
git clone https://github.com/monte-language/typhon/
nix-env -f typhon -iA monte
```

**From Cachix**

One of our community members maintains a Cachix instance. Instructions are at the Monte Cachix page.

Once that’s set up, you can install Monte by running:

```
nix-env -i 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:

```shell
>>> 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
Let’s see what a simple web server looks like in Monte:

```
import "lib/http/server" =~ [=> makeHTTPEndpoint]
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: that is, 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 \texttt{def body := ...} is an expression with a value, just like string literals and method calls.

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

\begin{verbatim}
> def portNum := 8080
> `serving on port $portNum`
>serving on port 8080
\end{verbatim}

Another quasiliteral is \texttt{b \texttt{\textbackslash<p>Hello!</textbackslash<p>}}, which denotes a \texttt{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 \texttt{argv.last()} as well as \texttt{function calls} such as \texttt{traceln(portNum)} and even syntax for constructing lists like \texttt{[200, [], body]} and maps like \texttt{["C" \texttt{\textRightarrow} "\texttt{t}"]}. 

Cooperation Without Vulnerability

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

\begin{verbatim}
import "lib/http/server" =~ [=> makeHTTPEndpoint]
import "lib/http/tag" =~ [=> tag]
import "formData" =~ [=> fieldMap]
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"
\end{verbatim}
```python
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`]
```

```python
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. 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 2. A Taste of Monte: Hello Web
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:

```monte
>>> 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:

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

... and this object:
Monte Documentation, Release 0.1

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> -3.5 // 1
-4
```
Monte Documentation, Release 0.1

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:

```monte
# 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:
  # Initial mafia 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))):
        ix.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):
        try:
            game.vote(randName(), randName())
            continue
        except _:
            step := game.advance()
        if (step != `@n votes cast.`):
            steps.push(step)
            steps.push('game')

        steps.push(step)
        steps.push('game')

    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; \texttt{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 \texttt{mafiososCount :Int}), parameters (such as \texttt{players :Set}), and return values (such as \texttt{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:

\begin{verbatim}
def x := 42 + true
\end{verbatim}

This statement will result in an error, because \texttt{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 \texttt{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

\texttt{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.
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

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

Basic Flow

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

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

```monte
>>> var resource := "reserved"
... try:
... 3 // 0
... catch err:
... "error!"
... finally:
... resource := "released"
... resource
"released"
```
Monte Documentation, Release 0.1

```python
>>> 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**

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

**Singleton Object (stateless)**

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

**Objects with state**

```python
>>> 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

```python
>>> 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

```python
>>> 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:

```python
>>> def main(argv, => makeFileResource):
...     def fileA := makeFileResource("fileA")
...     fileA <- setContents(b"abc
def")
...     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) {
... traceeln("Lost connection to carRcvr")
... })
... carRcvr
"shiny Mercedes"

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

```python
squares = [x**2 for x in range(10)]
more_squares = {x: x**2 for x in (2, 4, 6)}
```

Monte:

```monte
def squares := [for x in (0..9) 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:

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

Monte:

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

Classes

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

Python:

```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:

```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:

```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:

```monte
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
```
Monte Documentation, Release 0.1

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

Monte, styled like Monte:

```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:**

```python
class ClassName(object):

    _state = 42

    def _private(self):
        return self._state

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

Monte, styled like Python:

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

    def private():
        return state

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

Monte, styled like Monte:

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

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

5.2. Objects
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 contrast, 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.
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 Identifiers**

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"
::"liπ"

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.
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:

```monte
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 makeTCP4ClientEndpoint, getAddrInfo, and Timer.

```monte
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)
                        if (addr.getFamily() == "INET" &&
                            addr.getSocketType() == "stream") addr.getAddress()
                ]
                def ep := makeTCP4ClientEndpoint(address, port)
                connectIRCClient(client, ep)
```

Distributed Systems

Monte comes with built-in 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('&lt;channel $name&gt;')
        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.

---

1 As documented in the DarpaBrowser report

---

8.1. Practical Security II: The Mafia IRC Bot (WIP)
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 $mafiaChan to meet the others.', notHere)
                traceln("waiting for", mafiosos, "in", mafiaChan)
                when (mafiaChan.waitFor(mafiosos)) ->
                    traceln("done waiting for", mafiosos)
                    night0 := false
                    # Morning of day 1...
                    chan.say(game.advance())
    to said(who :Str, message :Str) :Bool:
        "Return true to continue, false if game over."
        mafiaChan.notify()
        traceln("notifying", mafiaChan)
        if (night0):
            return true
        if (message =~ `lynch @whom!`):
            escape notPlaying:
                def p := moderator.getPlayer(who, notPlaying)
                p.voteFor(whom)
                traceln("lynch", who, whom)
                if (game.getWinner() =~ winner ? winner != null):
                    makerd.end()
                    return game.getWinner() == null
    to getPlayer(name, notPlaying):
        return toPlayer.fetch(name, notPlaying)
    to end():
        chan.say('$/game')
        chan.part("Good game!")
        mafiaChan.part("bye bye")
Note the way `makeMafiaBot` provides a secret channel for the mafiosos to collude at night:

```python
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()
```
Ejectors & Escape Expressions

Ejectors can be hard to explain with words alone, so we will start with code:

```
# 42
escape ej { 42 }

# 42
escape ej { ej(42) }

# null
escape ej { ej() }
```

### What’s in a Name?

Ejectors are traditionally named `ej`, from the E programming language, but other names are common too. `k` is a traditional name from Scheme meaning “continuation”, referring to the technical definition of ejectors as single-use delimited continuations. For catch clauses, the traditional name `problem` (or `p` for short) is common.

An *escape expression* creates an *ejector*, which is an ordinary-looking object, and then evaluates its body. Calling `.run()` on an ejector will change the return value from the body’s return value to whatever is passed, or `null` by default.

We can also optionally catch the value and manipulate it. However, any `catch` clause will only be run if the ejector is called:

```
# 42
escape ej { 42 } catch p { 5 }

# 5
escape ej { ej() } catch p { 5 }

# 7
escape ej { ej(42) } catch p { p // 6 }
```

### Ejector-based Control Flow

The first major use for ejectors is in implementing several common kinds of control flow. By themselves, ejectors can be used to prematurely end or ‘short-circuit’ a computation; calling an ejector prevents any future computation:

```
# 42, no exception
escape ej { ej(42); 5 // 0 }
```

Ejectors even work when called by other objects:

```
# 6
def f(x, ej):
    return ej(x) * 7
escape ej { f(6, ej) }
```
Conditional Definitions

```lisp
# 0
escape ej {
    def x :Int exit ej := "five"
    x
} catch problem { 0 }
```

**throw.eject**

Often we might want to ensure that the object we are calling will actually alter control flow. We will see many motivating examples shortly. In these cases, we can use `throw.eject/2` to ensure that we will not continue computation:

```lisp
if (weAreFinished):
    throw.eject(ej, "finished")
launchMissiles<-()
```

This is equivalent to `ej("finished")` but will only launch missiles conditionally. We might imagine a simple implementation of this method:

```lisp
def throwEject(ej, problem):
    ej(problem)
    throw(problem)
```

**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:

```lisp
]> 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:

```lisp
]> 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:
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:

```python
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:

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

A variant that might be more useful in the future:

```python
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.

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 probabilistic 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 reasons, 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.

```python
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.

```python
# 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:

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

Brands are usable as map keys:

```python
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:

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

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

```python
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:

```python
brandMap[box.getBrand()] # should be `[ana, cata]`
```
Monte Documentation, Release 0.1

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**:

```d
# 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:

```d
# 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.

```d
# 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:

```plaintext
# 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:

```plaintext
source(sink)
```

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

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

We can also handle errors in case of failed delivery:

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

Hand-delivering data to a sink is easy:

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

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

```plaintext
object sink:
  to run(datum):
    return process<-(datum)
  to complete():
```

8.6. Streamcaps
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:

```monte
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.

```monte
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:

```monte
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:

```python
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:

```python
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
```

**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:

```bash
$ 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.
Monte Documentation, Release 0.1

(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.
Monte has three built-in types of collections, each of which come in “const” (immutable) and “flex” (mutable) flavors.

Sets

Monte’s sets are ordered containers with the standard assortment of set-theoretic tools, like membership testing, iteration, union, and intersection. Members are stored based on the sameness test; two members overlap if, and only if, they are the same.

Sets support syntactic comparison using the $\subseteq$ and related operators. The comparison takes the form of a subset test. Two sets $s$ and $t$ are equivalent, $s \subseteq t$, if, and only if, they contain the same members and are the same size.

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:

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

And any pattern can be used with brackets:

```monte
def `x := @[var x]` := "x := 7"
x += "-11"  # What? I like slushies!
```

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

```monte
# `''` 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**

The simple or empty QP builds strings:

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

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

```monte`
`$(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:

```monte
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("how could it be to match " + "@this?", [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"
```
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.
Monte Documentation, Release 0.1

_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:

```monte
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:

```monte
# 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:

```monte
def names := ["Scooby", "Shaggy", "Velma"]
def names := ["Bagira" => "panther",
"Baloo" => "bear",
"Shere Khan" => "tiger",
]
for i => name in (names):
    traceln('Name $i: $name')
for animal => species in (animals):
    traceln('Animal $animal is a $species')
```

while loops

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

```python
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:

```python
>>> { 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:

```python
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:

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

```python
>>> def evens := [for number in (1..10) if (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 the `__makeIterator__()` method, which returns an iterator. Iterators need to have the `next(ej)` method, 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:

```python
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:

```python
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.

```monte
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:

```monte
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`.

Controllers

Sometimes, when designing an API, we want to be able to customize Monte’s behavior while retaining the general Monte idioms for values and layouts. Controller syntax lets us change behavior of code blocks in a safe and coherent
How to Implement a Controller

Controller Expansion

Suppose that we have a standard if-expression:

```plaintext
if (cond()) {
  advance()
} else {
  fallback()
}
```

Now, suppose that we wished to customize this. We could define a controller named `ifController`, and then call it with very similar syntax:

```plaintext
ifController (cond()) do {
  advance()
} else {
  fallback()
}
```

This expands roughly to the following:

```plaintext
(ifController :DeepFrozen).control("do", 1, 0, fn {
  [[cond()], fn { advance() }]
}).control("else", 0, 0, fn {
  [[]], fn { fallback() }
}).controlRun()
```

We see that controllers must be `DeepFrozen`, and that each code block, which we’ll call a “lambda-block”, corresponds to a `.control/4` call, with a `.controlRun()` to indicate the end of blocks.

Control with Lambda-Blocks

The power of controllers is locked within the lambda-blocks. Each block is a function which returns an `[args, lambda]` pair. The controller can choose how many times it wants to call the block, and similarly, the block can return new arguments every time it is called. Indeed, note above that `cond()` is called every time its containing lambda-block is called.

What are the other arguments to `.control(verb :Str, argCount :Int, paramCount :Int, block)`? The control verb is the bare word preceding each block. The argument count specifies how many arguments will be returned by the block. Where are the parameters?

Let us imagine another hypothetical controller:

```plaintext
m (action) do x { f(x) }
```

In this situation, `x` is the one and only parameter, and so the controller receives a parameter count of `1`.

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:
**Final Slots**

Final slots are created by final definitions:

```plaintext
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.

```plaintext
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:

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

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

```plaintext
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.

9.9. Auditors
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 evaluated, 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.
Monte Documentation, Release 0.1

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 produce 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 facilitates 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:

```
<table>
<thead>
<tr>
<th>try</th>
<th>finally</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>success</td>
<td>success</td>
</tr>
<tr>
<td>error</td>
<td>success</td>
<td>error</td>
</tr>
<tr>
<td>success</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>error</td>
<td>error</td>
<td>error</td>
</tr>
</tbody>
</table>
```
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.

**Categorial Semantics**

**DF-Mont-Mess**

Let **DF-Mont-Mess** be the category whose objects are DeepFrozen messages and whose arrows are DeepFrozen Monte objects. For our diagrams, we will follow the convention that arrows are arrows and objects are encircled.

Since **DF-Mont-Mess** is a category, it must have an identity arrow for all messages.

```
["run", [42], []).asMap()]
```

In Monte, this object simply repeats messages delivered to it:

```mont
object id {
    match message {
        message
    }
}
```

**DF-Mont**

Let **DF-Mont** be the category whose objects are DeepFrozen values, not just messages, and whose arrows are DeepFrozen objects, as well as several primitives. The most important primitive is likely the ability to perform a call.
This is like the Monte expression 1 + 1, or (1).add(1). It is also like the Monte expression 2. In DF-Mont, Monte execution is represented by diagrams which commute, and the direction of computation is indicated by the direction of arrows.

**Initial Object**

We can formalize the statement that every object in DF-Mont is DeepFrozen by showing that there is a unique arrow (up to isomorphism) \( ! \) from DeepFrozen to any other object \( \text{obj} \) in the category.

This diagram commutes. The up-to-isomorphism limitation comes from \( \text{null} \) in \( \text{coerce}/2 \); we may replace it in this diagram with any other object.
Products

Lists act as our products. We can either use calls to do work on lists, or we can use categorical logic. The arrow \([[1, 2], [3, 4]] \rightarrow [[1, 2], "add", [[3, 4]], [].asMap()\) is a member of a family of list-building arrows.
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@@ */ | block )]

| 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 ",")]

guard ::= 
| /* empty */
| IDENTIFIER "(" ((expr ",")+) | /* empty */ ")"

module_header ::= "imports" StrExpr "=~" ( (pattern)+ ) [exports ] sequence

exports ::= "exports" 
| ("(" name "," ")+ | /* empty */ ")")

sequence ::= (a 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 */) "]

MapPatt ::= [" (mapPattItem ",")+ "]

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
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
  – Branding
  – Object capability community outreach

• Improve the core
  – Speed: Nobody should have to wait for code to compile
  – Safe objects
    * Many method improvements to builtin collections
    * Semitransparent
    * Vow
  – Unsafe objects
    * Timers
    * Property tests
  – Typhon-specific improvements
    * Even faster interpreting

• Develop important libraries
  – HTTP
– Records
  • Monte-related R&D
    – Capn Proto

2018

• Advanced safe objects
  – PassByCopy
  – makeWeakMap
  – Twines
  – Elusive Eight: Useful numerical analysis methods for doubles

• Production-ready unsafe objects
  – FS
  – Tamed timers
  – Typhon-specific improvements
    * Even faster interpreting

• Develop Monte packaging
  – Muffins
  – Packages
  – Environments
  – mtpkgs

• Develop important libraries
  – Debugger
  – Pretty-printers

• Monte-related R&D
  – Rationals
  – Capn Proto: Message generation, CapTP/VatTP
  – kubeless integration

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.
safeScope

Bindings in the safe scope are available to modules by default. They are all DeepFrozen.

Todo

Fix the module.name notation resulting from abuse of sphinx python support.

Todo

When Bool is fixed to reveal its interface, re-run mtDocStrings to document and, or, xor, not, butNot, pick, op__cmp.

Basic guards

class safeScope.Bool

The set of Boolean values: [true, false].asSet()

This guard is unretractable.

static coerce(_, _)
  no docstring

static getDocstring()
  no docstring

static getMethods()
  no docstring

static supersetOf(_)
  no docstring

class safeScope.Str

An ordered vector space.

As a guard, this object admits any value in the set of objects in the space. Comparison operators may be used on this object to create subguards which only admit a partition of the set.

static _printOn(_)
  no docstring

static _uncall()
  no docstring

static add(_)
  no docstring

static coerce(_, _)
  no docstring

static makeEmptyRegion()
  no docstring

static makeRegion(_, _, _, _)
  no docstring

static op__cmp(_)
  no docstring
static subtract
  no docstring

_makeIterator()
  no docstring

add
  no docstring

asList
  no docstring

asSet
  no docstring

contains
  no docstring

endsWith
  no docstring

get
  no docstring

getSpan
  no docstring

indexOf
  no docstring

isEmpty
  no docstring

join
  no docstring

lastIndexOf
  no docstring

multiply
  no docstring

op__cmp
  no docstring

quote
  no docstring

replace
  no docstring

size
  no docstring

slice
  no docstring

split
  no docstring

startsWith
  Whether this string has s as a prefix.
```plaintext
toLowerCase()  
  no docstring
toUpperCase()  
  no docstring
trim()  
  no docstring
with(_)  
  no docstring
class safeScope.Char  
  An ordered vector space.
  As a guard, this object admits any value in the set of objects in the space. Comparison operators may be used on this object to create subguards which only admit a partition of the set.
  static _printOn(_)  
    no docstring
  static _uncall()  
    no docstring
  static add(_)  
    no docstring
  static coerce(_,_)  
    no docstring
  static makeEmptyRegion()  
    no docstring
  static makeRegion(_,_,_,_)  
    no docstring
  static op__cmp(_)  
    no docstring
  static subtract(_)  
    no docstring
  add(_)  
    no docstring
  asInteger()  
    no docstring
  asString()  
    no docstring
  getCategory()  
    no docstring
  max(_)  
    no docstring
  min(_)  
    no docstring
  next()  
    no docstring
```

10.3. safeScope
class safeScope.Double

An ordered vector space.

As a guard, this object admits any value in the set of objects in the space. Comparison operators may be used on this object to create subguards which only admit a partition of the set.
cos()  
   no docstring

floor()  
   no docstring

floorDivide(_)  
   no docstring

isZero()  
   no docstring

log()  
   no docstring

multiply(_)  
   no docstring

negate()  
   no docstring

op__cmp(_)  
   no docstring

pow(_)  
   no docstring

sin()  
   no docstring

sqrt()  
   no docstring

subtract(_)  
   no docstring

tan()  
   no docstring
toBytes()  
   no docstring
class safeScope.Int
   An ordered vector space.
   As a guard, this object admits any value in the set of objects in the space. Comparison operators may be used on this object to create subguards which only admit a partition of the set.

static _printOn(_)
   no docstring

static _uncall()
   no docstring

static add(_)
   no docstring

static coerce(_, _)
   no docstring

static makeEmptyRegion()
   no docstring

10.3. safeScope
static makeRegion(., ., ., .)
  no docstring

static op__cmp(.)
  no docstring

static subtract(.)
  no docstring

aboveZero()
  no docstring

abs()
  no docstring

add(.)
  no docstring

and(.)
  no docstring

approxDivide(.)
  no docstring

asDouble()
  no docstring

atLeastZero()
  no docstring

atMostZero()
  no docstring

belowZero()
  no docstring

bitLength()
  no docstring

complement()
  no docstring

floorDivide(.)
  no docstring

isZero()
  no docstring

max(.)
  no docstring

min(.)
  no docstring

mod(.)
  no docstring

modPow(. .)
  no docstring

multiply(.)
  no docstring
class safeScope.Bytes
An ordered vector space.
As a guard, this object admits any value in the set of objects in the space. Comparison operators may be used
on this object to create subguards which only admit a partition of the set.

static _printOn(_)
no docstring

static _uncall()
no docstring

static add(_)
no docstring

static coerce(_, _)
no docstring

static makeEmptyRegion()
no docstring

static makeRegion(_, _, _, _)
no docstring

static op__cmp(_)
no docstring

static subtract(_)
no docstring

_makeIterator()
no docstring
class safeScope.List
   A guard which admits lists.
Only immutable lists are admitted by this object. Mutable lists created with `diverge/0` will not be admitted; freeze them first with `snapshot/0`.

```plaintext
static _printOn (_,) no docstring
static coerce (_,_,) no docstring
static extractGuard (_,_,) no docstring
static get (_,) no docstring
_makeIterator () no docstring
_printOn (_,) no docstring
_uncall () no docstring
add (_,) no docstring
asMap () no docstring
asSet () no docstring
contains (_,) no docstring
diverge () no docstring
empty () no docstring
get (_,) no docstring
indexOf (_,) no docstring
isEmpty () no docstring
join (_,) no docstring
last () no docstring
multiply (_,) no docstring
op__cmp (_,) no docstring
```
class safeScope.Map
A guard which admits maps.

Only immutable maps are admitted by this object. Mutable maps created with diverge/0 will not be admitted; freeze them first with snapshot/0.
class safeScope.Set
A guard which admits sets.

Only immutable sets are admitted by this object. Mutable sets created with diverge/0 will not be admitted; freeze them first with snapshot/0.

static _printOn (_, _)
    no docstring

static coerce (_, _)
    no docstring

static extractGuard (_, _)
    no docstring

static get (_, _)
    no docstring

_makeIterator ()
    no docstring
safeScope.Pair
A guard which admits immutable pairs.
Pairs are merely lists of size two.
safeScope._printOn(_)
no docstring
safeScope.coerce(_, _)
no docstring
safeScope.extractGuards(_, _)  
   no docstring

safeScope.get(_, _)  
   no docstring

class safeScope.FinalSlot  
   A guard which emits makers of FinalSlots.

   static coerce(_, _)  
      no docstring

   static extractGuard(_, _)  
      no docstring

   static get(_)  
      no docstring

   static getDocstring()  
      no docstring

   static getGuard()  
      no docstring

   static getMethods()  
      no docstring

   static supersetOf(_)  
      no docstring

class safeScope.VarSlot  
   A guard which admits makers of VarSlots.

   static coerce(_, _)  
      no docstring

   static extractGuard(_, _)  
      no docstring

   static get(_)  
      no docstring

   static getDocstring()  
      no docstring

   static getGuard()  
      no docstring

   static getMethods()  
      no docstring

   static supersetOf(_)  
      no docstring

Guard utilities

class safeScope.Any  
   A guard which admits the universal set.

   This object specializes to a guard which admits the union of its subguards: Any[X, Y, Z] =~ X Y Z

   This guard is unretractable.
static coerce(_:_)  
  no docstring

static extractGuards(_:_)  
  no docstring

static getMethods()  
  no docstring

static supersetOf(_)  
  no docstring

class safeScope.Void  
The singleton set of null: [null].asSet()  
  This guard is unretractable.

static coerce(_:_)  
  no docstring

static getDocstring()  
  no docstring

static getMethods()  
  no docstring

static supersetOf(_)  
  no docstring

safeScope.Empty  
  An unretractable predicate guard.  
  This guard admits any object which passes its predicate.

safeScope._printOn(_)  
  no docstring

safeScope.coerce(_:_)  
  no docstring

safeScope.NullOk  
  A guard which admits null.  
  When specialized, this object returns a guard which admits its subguard as well as null.

safeScope.coerce(_:_)  
  no docstring

safeScope.extractGuard(_:_)  
  no docstring

safeScope.get(_)  
  no docstring

safeScopeSAME  
  When specialized, this object yields a guard which only admits precisely the object used to specialize it.  
  In simpler terms, Same[x] will match only those objects o for which o == x.

safeScope.extractValue(_:_)  
  no docstring

safeScope.get(_)  
  no docstring
safeScope.Vow
A guard which admits promises and their entailments.

Vows admit the union of unfulfilled promises, fulfilled promises, broken promises, and Near values. The unifying concept is that of a partial future value to which messages will be sent but that is not Far.

When specialized, this guard returns a guard which ensures that promised prizes either conform to its subguard or are broken.

safeScope._printOn(_)
no docstring

safeScope.coerce(_, _)
no docstring

safeScope.extractGuard(_, _)
no docstring

safeScope.get(_)
no docstring

safeScope.SubrangeGuard
The maker of subrange guards.

When specialized with a guard, this object produces an auditor for those guards which admit proper subsets of that guard.

safeScope.get(_)
no docstring

safeScope._auditedBy
Whether an auditor has audited a specimen.

safeScope.run(_, _)
no docstring

Primitive values

safeScope.true
: Bool

safeScope.false
: Bool

safeScope.null
: Void

safeScope.NaN
: Double

safeScope.Infinity
: Double

Data Constructors

safeScope._makeInt
A maker of 'Int's.

This maker can handle radices from 2 to 36:

> _makeInt.withRadix(36)("zxcvasdfqwer1234") 7942433573816828193485776
safeScope.fromBytes(_, _)
   no docstring

safeScope.run(_)
   no docstring

safeScope.withRadix(_)
   no docstring

safeScope._makeDouble
   The maker of ‘Double’s.

safeScope.fromBytes(_, _)
   no docstring

safeScope.run(_, _)
   no docstring

safeScope._makeStr
   The maker of ‘Str’s.

safeScope.fromChars(_)  
   no docstring

safeScope.fromString(_, _)
   no docstring

safeScope._makeString
   The maker of ‘Str’s.

safeScope.fromChars(_)  
   no docstring

safeScope.fromString(_, _)
   no docstring

safeScope._makeBytes
   The maker of Bytes.

safeScope.fromInts(_)  
   no docstring

safeScope.fromString(_)
   no docstring

safeScope._makeList
   The maker of ‘List’s.

safeScope.fromIterable(_)
   no docstring

safeScope._makeMap
   Given a List[Pair], produce a Map.

safeScope.fromPairs(_)
   no docstring

safeScope._makeOrderedSpace
   The maker of ordered vector spaces.

   This object implements several Monte operators, including those which provide ordered space syntax.

safeScope.op__thru(_, _)
   no docstring
safeScope.\texttt{op\_till}(_, _)  
\textit{no docstring}

safeScope.\texttt{spaceOfGuard}(_)  
\textit{no docstring}

safeScope.\texttt{spaceOfValue}(_)  
\textit{no docstring}

\textbf{safeScope.\_makeTopSet}

safeScope.\texttt{run}(_, _, _, _, _, _)  
\textit{no docstring}

\textbf{safeScope.\_makeOrderedRegion}
Make regions for sets of objects with total ordering.

safeScope.\texttt{run}(_, _, _)  
\textit{no docstring}

\textbf{safeScope.\_makeSourceSpan}
\textit{no docstring}

safeScope.\texttt{run}(_, _, _, _, _, _, _)  
\textit{no docstring}

\textbf{safeScope.\_makeFinalSlot}
A maker of final slots.

safeScope.\texttt{asType}()  
\textit{no docstring}

safeScope.\texttt{run}(_, _, _)  
\textit{no docstring}

\textbf{safeScope.\_makeVarSlot}
A maker of var slots.

safeScope.\texttt{asType}()  
\textit{no docstring}

safeScope.\texttt{run}(_, _, _)  
\textit{no docstring}

\textbf{safeScope.makeLazySlot}
Make a slot that lazily binds its value.

safeScope.\texttt{run}(_)  
\textit{no docstring}

\textbf{Tracing}

safeScope.\texttt{trace}
Write a line to the trace log.

This object is a Typhon standard runtime \texttt{traceln}. It prints prefixed lines to stderr.

Call \texttt{.exception(problem)} to print a problem to stderr, including a formatted traceback.

safeScope.\texttt{exception}(_)  
\textit{no docstring}

10.3. \texttt{safeScope}

99
safeScope.traceIn
Write a line to the trace log.

This object is a Typhon standard runtime traceln. It prints prefixed lines to stderr.
Call .exception(problem) to print a problem to stderr, including a formatted traceback.

safeScope.exception(_)
no docstring

Brands

safeScope.makeBrandPair
Make a [sealer, unsealer] pair.

safeScope.run(_)
no docstring

Quasiparsers

```
A quasiparser of Unicode strings.

This object is the default quasiparser. It can interpolate any object into a string by pretty-printing it; in fact, that
is one of this object’s primary uses.

When used as a pattern, this object performs basic text matching. Patterns always succeed, grabbing zero or
more characters non-greedily until the next segment. When patterns are concatenated in the quasiliteral, only
the rightmost pattern can match any characters; the other patterns to the left will all match the empty string.
```

safeScope.matchMaker(_)
no docstring

safeScope.patternHole(_)
no docstring

safeScope.valueHole(_)
no docstring

```
A quasiparser for Bytes.

This object behaves like simple_quasiParser; it takes some textual descriptions of bytes and returns a
bytestring. It can interpolate objects which coerce to Bytes and Str.

As a pattern, this object performs slicing of bytestrings. Semantics mirror simple_quasiParser with respect to
concatenated patterns and greediness.
```

safeScope.matchMaker(_)
no docstring

safeScope.patternHole(_)
no docstring

safeScope.valueHole(_)
no docstring
A quasiparser for the Monte programming language.
This object will parse any Monte expression and return an opaque value. In the near future, this object will
instead return a translucent view into a Monte compiler and optimizer.

A quasiparser for the Monte programming language's patterns.
This object is like `m''`, but for patterns.

Flow control

Miscellaneous vat management and quoting services.

10.3. safeScope
safeScope.sendOnly(_._._._)  
   no docstring

safeScope.toQuote(_)  
   no docstring

safeScope.toString(_)  
   no docstring

safeScope.throw  
   no docstring

safeScope.eject(_._)  
   no docstring

safeScope.run(_)  
   no docstring

<table>
<thead>
<tr>
<th>safeScope._loop</th>
<th>Perform an iterative loop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>safeScope.run(<em>.</em>)</td>
<td>no docstring</td>
</tr>
</tbody>
</table>

safeScope._iterForever  
   Implementation of while-expression syntax.

<table>
<thead>
<tr>
<th>safeScope._makeIterator</th>
<th>()</th>
</tr>
</thead>
<tbody>
<tr>
<td>no docstring</td>
<td></td>
</tr>
</tbody>
</table>

|safeScope.next(_)| no docstring |

**Evaluation**

safeScope.eval  
   Evaluate Monte source.

This object respects POLA and grants no privileges whatsoever to evaluated code. To grant a safe scope, pass safeScope.

<table>
<thead>
<tr>
<th>safeScope.evalToPair(<em>.</em>._)</th>
<th>()</th>
</tr>
</thead>
<tbody>
<tr>
<td>no docstring</td>
<td></td>
</tr>
</tbody>
</table>

|safeScope.run(_._)| no docstring |

<table>
<thead>
<tr>
<th>safeScope.astEval</th>
<th>()</th>
</tr>
</thead>
<tbody>
<tr>
<td>no docstring</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>safeScope.evalToPair(<em>.</em>._)</th>
<th>()</th>
</tr>
</thead>
<tbody>
<tr>
<td>no docstring</td>
<td></td>
</tr>
</tbody>
</table>

|safeScope.run(_._)| no docstring |

**Reference/object operations**

safeScope.Ref  
   Ref management and utilities.
safeScope\texttt{.broken(_)}
\textit{no docstring}

safeScope\texttt{.fulfillment(_)}
\textit{no docstring}

safeScope\texttt{.isBroken(_)}
\textit{no docstring}

safeScope\texttt{.isDeepFrozen(_)}
\textit{no docstring}

safeScope\texttt{.isEventual(_)}
\textit{no docstring}

safeScope\texttt{.isFar(_)}
\textit{no docstring}

safeScope\texttt{.isNear(_)}
\textit{no docstring}

safeScope\texttt{.isResolved(_)}
\textit{no docstring}

safeScope\texttt{.isSelfish(_)}
\textit{no docstring}

safeScope\texttt{.isSelfless(_)}
\textit{no docstring}

safeScope\texttt{.makeProxy(_, _, _)}
\textit{no docstring}

safeScope\texttt{.optProblem(_)}
\textit{no docstring}

safeScope\texttt{.promise()}
\textit{no docstring}

safeScope\texttt{.state(_)}
\textit{no docstring}

safeScope\texttt{.whenBroken(_, _)}
\textit{no docstring}

safeScope\texttt{.whenBrokenOnly(_, _)}
\textit{no docstring}

safeScope\texttt{.whenResolved(_, _)}
\textit{no docstring}

safeScope\texttt{.whenResolvedOnly(_, _)}
\textit{no docstring}

safeScope\texttt{.promiseAllFulfilled}

safeScope\texttt{.run(_)}
\textit{no docstring}

\texttt{safeScope.DeepFrozen}
\textit{Auditor and guard for transitive immutability.}
safeScope.audit(_)  
no docstring

safeScope.coerce(_, _)  
no docstring

safeScope.supersetOf(_)  
no docstring

safeScope(Selfless)
A stamp for incomparable objects.

Selfless objects are generally not equal to any objects but themselves. They may choose to implement alternative comparison protocols such as Transparent.

safeScope.audit(_)  
no docstring

safeScope.coerce(_, _)  
no docstring

safeScope.passes(_)  
no docstring

safeScope.Transparent
Objects that Transparent admits have reliable ._uncall() methods, in the sense that they correctly identify their maker and their entire state, and that invoking the maker with the given args will produce an object with the same state. Objects that are both Selfless and Transparent are compared for sameness by comparing their uncalls.

safeScope.coerce(_, _)  
no docstring

safeScope.makeAuditorKit()  
no docstring

safeScope.Near
A guard over references to near values.

This guard admits any near value, as well as any resolved reference to any near value.

This guard is unretractable.

safeScope.coerce(_, _)  
no docstring

class safeScope.Binding
A guard which admits bindings.

static coerce(_, _)  
no docstring

static getDocstring()  
no docstring

static getMethods()  
no docstring

static supersetOf(_)  
no docstring
Abstract Syntax

safeScope.astBuilder

safeScope.AndExpr(_, _, _)
   no docstring

safeScope.AssignExpr(_, _, _)
   no docstring

safeScope.AugAssignExpr(_, _, _)
   no docstring

safeScope.BinaryExpr(_, _, _, _)
   no docstring

safeScope.BindPattern(_, _)
   no docstring

safeScope.BindingExpr(_, _)
   no docstring

safeScope.BindingPattern(_, _)
   no docstring

safeScope.CatchExpr(_, _, _, _)
   no docstring

safeScope.Catcher(_, _, _)
   no docstring

safeScope.CoerceExpr(_, _, _, _)
   no docstring

safeScope.CompareExpr(_, _, _, _)
   no docstring

safeScope.CurryExpr(_, _, _)
   no docstring

safeScope.DefExpr(_, _, _)
   no docstring

safeScope.EscapeExpr(_, _, _, _, _, _)
   no docstring

safeScope.ExitExpr(_, _, _)
   no docstring

safeScope.FinalPattern(_, _, _)
   no docstring

safeScope.FinallyExpr(_, _, _)
   no docstring

safeScope.ForExpr(_, _, _, _, _, _, _)
   no docstring

safeScope.ForwardExpr(_, _)
   no docstring

safeScope.FunCallExpr(_, _, _)
   no docstring
safeScope.FunSendExpr(_, _, _)
   no docstring
safeScope.FunctionExpr(_, _, _)
   no docstring
safeScope.FunctionInterfaceExpr(_, _, _, _, _)
   no docstring
safeScope.FunctionScript(_, _, _)
   no docstring
safeScope.GetExpr(_, _)
   no docstring
safeScope.HideExpr(_, _)
   no docstring
safeScope.IfExpr(_, _, _, _)
   no docstring
safeScope.IgnorePattern(_, _)
   no docstring
safeScope.InterfaceExpr(_, _, _, _, _, _)
   no docstring
safeScope.ListComprehensionExpr(_, _, _, _)
   no docstring
safeScope.ListExpr(_, _)
   no docstring
safeScope.ListPattern(_, _)
   no docstring
safeScope.LiteralExpr(_, _)
   no docstring
safeScope.MapComprehensionExpr(_, _, _, _, _)
   no docstring
safeScope.MapExpr(_, _)
   no docstring
safeScope.MapExprAssoc(_, _, _)
   no docstring
safeScope.MapExprExport(_, _)
   no docstring
safeScope.MapPattern(_, _, _)
   no docstring
safeScope.MapPatternAssoc(_, _, _, _)
   no docstring
safeScope.MapPatternImport(_, _, _)
   no docstring
safeScope.MatchBindExpr(_, _, _)
   no docstring
safeScope.Matcher(_._._)
   no docstring
safeScope.MessageDesc(_._._._)
   no docstring
safeScope.MetaContextExpr(_)
   no docstring
safeScope.MetaStateExpr(_)
   no docstring
safeScope.Method(_._._._._._)
   no docstring
safeScope.MethodCallExpr(_._._._._)
   no docstring
safeScope.MismatchExpr(_._._)
   no docstring
safeScope.Module(_._._)
   no docstring
safeScope.NamedArg(_._._)
   no docstring
safeScope.NamedArgExport(_._)
   no docstring
safeScope.NamedParam(_._._._)
   no docstring
safeScope.NamedParamImport(_._._)
   no docstring
safeScope.NounExpr(_._)
   no docstring
safeScope.ObjectExpr(_._._._._)
   no docstring
safeScope.OrExpr(_._._)
   no docstring
safeScope.ParamDesc(_._._)
   no docstring
safeScope.PatternHoleExpr(_._)
   no docstring
safeScope.PatternHolePattern(_._)
   no docstring
safeScope.PrefixExpr(_._._)
   no docstring
safeScope.QuasiExprHole(_._._)
   no docstring
safeScope.QuasiParserExpr(_._._)
   no docstring
safeScope::QuasiParserPattern(_._._)
  no docstring

safeScope::QuasiPatternHole(_._)
  no docstring

safeScope::QuasiText(_._)
  no docstring

safeScope::RangeExpr(_._._)
  no docstring

safeScope::SameExpr(_._._)
  no docstring

safeScope::SamePattern(_._._)
  no docstring

safeScope::Script(_._._._)
  no docstring

safeScope::SendExpr(_._._._)
  no docstring

safeScope::SeqExpr(_._)
  no docstring

safeScope::SlotExpr(_._)
  no docstring

safeScope::SlotPattern(_._._)
  no docstring

safeScope::SuchThatPattern(_._._)
  no docstring

safeScope::SwitchExpr(_._._)
  no docstring

safeScope::TempNounExpr(_._)
  no docstring

safeScope::To(_._._._._._)
  no docstring

safeScope::TryExpr(_._._._)
  no docstring

safeScope::ValueHoleExpr(_._)
  no docstring

safeScope::ValueHolePattern(_._)
  no docstring

safeScope::VarPattern(_._._)
  no docstring

safeScope::VerbAssignExpr(_._._._)
  no docstring

safeScope::ViaPattern(_._._)
  no docstring
safeScope .WhenExpr(_, _, _, _)  
no docstring

safeScope .WhileExpr(_, _, _)  
no docstring

safeScope .getAstGuard()  
no docstring

safeScope .getExprGuard()  
no docstring

safeScope .getNamePatternGuard()  
no docstring

safeScope .getNounGuard()  
no docstring

safeScope .getPatternGuard()  
no docstring

Utilities for syntax expansions

safeScope ._accumulateList  
Implementation of list comprehension syntax.

safeScope .run(_, _)  
no docstring

safeScope ._accumulateMap  
Implementation of map comprehension syntax.

safeScope .run(_, _)  
no docstring

safeScope ._bind  
Resolve a forward declaration.

safeScope .run(_, _)  
no docstring

safeScope ._booleanFlow  
Implementation of implicit breakage semantics in conditionally-defined names.

safeScope .broken()  
no docstring

safeScope .failureList(_)  
no docstring

safeScope ._comparer  
A comparison helper.

This object implements the various comparison operators.

safeScope .asBigAs(_, _)  
no docstring

safeScope .geq(_, _)  
no docstring

safeScope .greaterThan(_, _)  
no docstring
safeScope.leq(_, _)  
no docstring

safeScope.lessThan(_, _)  
no docstring

safeScope._equalizer  
A perceiver of identity.
This object can discern whether any two objects are distinct from each other.

safeScope.isSettled(_)  
no docstring

safeScope.makeTraversalKey(_)  
no docstring

safeScope.optSame(_, _)  
no docstring

safeScope.sameEver(_, _)  
no docstring

safeScope.sameYet(_, _)  
no docstring

safeScope._makeVerbFacet  
The operator obj.'method'.

safeScope.curryCall(_, _)  
no docstring

safeScope.currySend(_, _)  
no docstring

safeScope._mapEmpty  
An unretractable predicate guard.
This guard admits any object which passes its predicate.

safeScope._printOn(_)  
no docstring

safeScope.coerce(_, _)  
no docstring

safeScope._mapExtract  
Implementation of key pattern-matching syntax in map patterns.

safeScope.run(_)  
no docstring

safeScope.withDefault(_, _)  
no docstring

safeScope._matchSame
safeScope._quasiMatcher
Implementation of quasiliteral pattern syntax.

safeScope.run(_)
no docstring

safeScope._slotToBinding
Implementation of bind-pattern syntax for forward declarations.

safeScope.run(_)
no docstring

safeScope._splitList
Implementation of tail pattern-matching syntax in list patterns.

m`def [x] + xs := l`.expand() == m`def via (_splitList.run(1)) [x, xs] := l`

safeScope.run(_)
no docstring

safeScope._suchThat
The pattern patt ? (expr).

safeScope.run(_)
no docstring

safeScope._switchFailed
The implicit default matcher in a switch expression.

This object throws an exception.

safeScope._validateFor
Ensure that flag is true.

This object is a safeguard against malicious loop objects. A flag is set to true and closed over by a loop body; once the loop is finished, the flag is set to false and the loop cannot be reentered.

safeScope.run(_)
no docstring

Interface constructors

safeScope._makeMessageDesc
Describe a message.

safeScope.run(_, _, _, _)
no docstring

safeScope._makeParamDesc
Describe a parameter.

safeScope.run(_, _)
no docstring

safeScope._makeProtocolDesc
Produce an interface.

safeScope.makePair(_, _, _, _)
no docstring

safeScope.run(_, _, _, _)
no docstring

10.3. safeScope
Entrypoint Arguments

Todo

Fix the `module.name` notation resulting from abuse of sphinx python support.

Time

```python
__entrypoint_io__.Timer
An unsafe nondeterministic clock.
This object provides a useful collection of time-related methods: *
* `fromNow(delay: Double)`: Produce a promise which will fully resolve after at least `delay` seconds have elapsed in the runtime. The promise will resolve to a `Double` representing the precise amount of time elapsed, in seconds.
* `sendTimestamp(callable)`: Send a `Double` representing the runtime’s clock to `callable`.
There is extremely unsafe functionality as well: *
* `unsafeNow()`: The current system time.
Use with caution.
```  
```python
__entrypoint_io__.fromNow(_)
no docstring
__entrypoint_io__.sendTimestamp(_)
no docstring
__entrypoint_io__.unsafeNow()
no docstring
```

I/O

```python
__entrypoint_io__.stdio
A producer of streamcaps for the ancient standard I/O bytestreams.
```  
```python
__entrypoint_io__.stderr()
no docstring
__entrypoint_io__.stdin()
no docstring
__entrypoint_io__.stdout()
no docstring
__entrypoint_io__.makeStdErr
no docstring
__entrypoint_io__.run()
no docstring
__entrypoint_io__.makeStdIn
no docstring
__entrypoint_io__.run()
no docstring
__entrypoint_io__.makeStdOut
no docstring
```
Monte Documentation, Release 0.1

__entrypoint_io__.run()
  no docstring

__entrypoint_io__.makeFileResource
  Make a file Resource.
  __entrypoint_io__.run(____)  no docstring

Networking

__entrypoint_io__.makeTCP4ClientEndpoint
  Make a TCPv4 client endpoint.
  __entrypoint_io__.run(____)
  no docstring

__entrypoint_io__.makeTCP4ServerEndpoint
  Make a TCPv4 server endpoint.
  __entrypoint_io__.run(____)
  no docstring

__entrypoint_io__.makeTCP6ClientEndpoint
  Make a TCPv6 client endpoint.
  __entrypoint_io__.run(____)
  no docstring

__entrypoint_io__.makeTCP6ServerEndpoint
  Make a TCPv6 server endpoint.
  __entrypoint_io__.run(____)
  no docstring

__entrypoint_io__.getAddrInfo
  no docstring
  __entrypoint_io__.run(____)
  no docstring

Runtime

__entrypoint_io__.currentRuntime
  The Typhon runtime.

  This object is a platform-specific view into the configuration and performance of the current runtime in the current process.

  This object is necessarily unsafe and nondeterministic.

  __entrypoint_io__.getCrypt()
  no docstring

  __entrypoint_io__.getHeapStatistics()
  no docstring

  __entrypoint_io__.getReactorStatistics()
  no docstring

10.4. Entrypoint Arguments 113
__entrypoint_io__.unsealException
Unseal a specimen.

__entrypoint_io__.run(_, _)  
no docstring

Processes and Vats

__entrypoint_io__.currentProcess
The current process on the local node.

__entrypoint_io__.getArguments()  
no docstring

__entrypoint_io__.getEnvironment()  
no docstring

__entrypoint_io__.getPID()  
no docstring

__entrypoint_io__.interrupt()  
no docstring

__entrypoint_io__.makeProcess
Create a subordinate process on the current node from the given executable, arguments, and environment.

=> stdin, => stdout, and => stderr control the same-named methods on the resulting process object, which will return a sink, source, and source respectively. If any of these named arguments are true, then the corresponding method on the process will return a live streamcap which is connected to the process; otherwise, the returned streamcap will be a no-op.

=> stdinFount, if not null, will be treated as a fount and it will be flowed to a drain representing stdin. => stdoutDrain and => stderrDrain are similar but should be drains which will have founts flowed to them.

__entrypoint_io__.run(_, _, _)  
no docstring

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 */
Todo

interface syntax diagram @@s

Todo

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

Todo

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

Todo

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

Todo

special operator rules because of security

Todo

VERB_ASSIGN lexical details

discuss, doctest SlotExpression &x, BindingExpression & &x
Todo

named args in argList

Todo
discuss matchers in object expressions

Todo
document docstrings

Todo
document named args, defaults

Todo

document guards: Guards play a key role in protecting security properties.

Todo

Fix the module.name notation resulting from abuse of sphinx python support.

Todo

When `Bool` is fixed to reveal its interface, re-run mtDocStrings to document and, or, xor, not, butNot, pick, op__cmp.
Todo

Fix the module.name notation resulting from abuse of sphinx python support.

Todo

discuss SlotExpr

Todo

discuss sameness and doctest _equalizer

Todo

specify canStartIndentedBlock, braceStack exactly

Todo

Document how to compile and run such a script.

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 facilitate computation.
prize : Coercion  The result of a successful coercion.
quasilateral, 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
__entrypoint_io__, 112

S
safeScope, 83
Symbols

_indexpoint_io_ (module), 112
__accumulateList (in module safeScope), 109
__accumulateMap (in module safeScope), 109
__auditedBy (in module safeScope), 97
__bind (in module safeScope), 109
__booleanFlow (in module safeScope), 109
__comparer (in module safeScope), 109
__equalizer (in module safeScope), 110
__iterForever (in module safeScope), 102
__loop (in module safeScope), 102
__makeBytes (in module safeScope), 98
__makeDouble (in module safeScope), 98
__makeFinalSlot (in module safeScope), 99
__makeInt (in module safeScope), 97
__makeIterator() (in module safeScope), 102
__makeIterator() (safeScope.Bytes method), 89
__makeIterator() (safeScope.Bytes static method), 89
__makeIterator() (safeScope.Char method), 85
__makeIterator() (safeScope.Char static method), 85
__makeIterator() (safeScope.Double method), 86
__makeIterator() (safeScope.Double static method), 86
__makeIterator() (safeScope.Int method), 88
__makeIterator() (safeScope.Int static method), 87
__makeIterator() (safeScope.List method), 91
__makeIterator() (safeScope.List static method), 91
__makeIterator() (safeScope.Map method), 92
__makeIterator() (safeScope.Map static method), 92
__makeIterator() (safeScope.Set method), 93
__makeIterator() (safeScope.Set static method), 93
__makeIterator() (safeScope.Str method), 84
__makeIterator() (safeScope.Str static method), 84
__makeList (in module safeScope), 98
__makeMap (in module safeScope), 98
__makeMessageDesc (in module safeScope), 111
__makeOrderedRegion (in module safeScope), 99
__makeOrderedSpace (in module safeScope), 98
__makeParamDesc (in module safeScope), 111
__makeProtocolDesc (in module safeScope), 111
__makeSourceSpan (in module safeScope), 99
__makeStr (in module safeScope), 98
__makeString (in module safeScope), 98
__makeTopSet (in module safeScope), 99
__makeVarSlot (in module safeScope), 99
__makeVerbFacet (in module safeScope), 110
__mapEmpty (in module safeScope), 110
__mapExtract (in module safeScope), 110
__matchSame (in module safeScope), 110
__printOn() (in module safeScope), 94, 96, 97, 110
__printOn() (safeScope.Bytes method), 89
__printOn() (safeScope.Bytes static method), 86
__printOn() (safeScope.Double method), 86
__printOn() (safeScope.Double static method), 86
__printOn() (safeScope.Int method), 88
__printOn() (safeScope.Int static method), 87
__printOn() (safeScope.List method), 91
__printOn() (safeScope.List static method), 91
__printOn() (safeScope.Map method), 92
__printOn() (safeScope.Map static method), 92
__printOn() (safeScope.Set method), 93
__printOn() (safeScope.Set static method), 93
__switchFailed (in module safeScope), 111
__validateFor (in module safeScope), 111

A
aboveZero() (safeScope.Double method), 86
aboveZero() (safeScope.Int method), 88
abs() (safeScope.Double method), 86
abs() (safeScope.Int method), 88
abstract syntax, 66
add() (safeScope.Bytes method), 90
add() (safeScope.Bytes static method), 89
add() (safeScope.Char method), 85
add() (safeScope.Char static method), 85
add() (safeScope.Double method), 86
add() (safeScope.Double static method), 86
add() (safeScope.Int method), 88
add() (safeScope.Int static method), 87
add() (safeScope.List method), 91
add() (safeScope.Str method), 84
add() (safeScope.Str static method), 83
and() (safeScope.Int method), 88
and() (safeScope.Set method), 94
AndExpr() (in module safeScope), 105
Any (class in safeScope), 95
approxDivide() (safeScope.Double method), 86
approxDivide() (safeScope.Int method), 88
asBigAs() (in module safeScope), 109
asDouble() (safeScope.Int method), 88
asInteger() (safeScope.Char method), 85
asList() (safeScope.Bytes method), 90
asList() (safeScope.Set method), 94
asList() (safeScope.Str method), 84
asMap() (safeScope.List method), 91
asSet() (safeScope.Bytes method), 90
asSet() (safeScope.List method), 91
asSet() (safeScope.Map method), 92
asSet() (safeScope.Set method), 94
asSet() (safeScope.Str method), 84
AssignExpr() (in module safeScope), 105
asString() (safeScope.Char method), 85
astBuilder (in module safeScope), 105
astEval (in module safeScope), 102
asType() (in module safeScope), 99
atLeastZero() (safeScope.Double method), 86
atLeastZero() (safeScope.Int method), 88
atMostZero() (safeScope.Double method), 86
atMostZero() (safeScope.Int method), 88
audit() (in module safeScope), 103, 104
AugAssignExpr() (in module safeScope), 105
B
belowZero() (safeScope.Double method), 86
belowZero() (safeScope.Int method), 88
BinaryExpr() (in module safeScope), 105
binding, 67
Binding (class in safeScope), 104
BindingExpr() (in module safeScope), 105
BindingPattern() (in module safeScope), 105
BindPattern() (in module safeScope), 105
bitLength() (safeScope.Int method), 88
Bool (class in safeScope), 83
broken() (in module safeScope), 102, 109
butNot() (safeScope.Set method), 94
Bytes (class in safeScope), 89
C
call() (in module safeScope), 101
callWithMessage() (in module safeScope), 101
Catcher() (in module safeScope), 105
CatchExpr() (in module safeScope), 105
Char (class in safeScope), 85
coerce() (in module safeScope), 94, 96, 97, 104, 110
coerce() (safeScope.Any static method), 95
coerce() (safeScope.Binding static method), 104
coerce() (safeScope.Bool static method), 83
coerce() (safeScope.Bytes static method), 89
coerce() (safeScope.Char static method), 85
coerce() (safeScope.Double static method), 86
coerce() (safeScope.FinalSlot static method), 95
coerce() (safeScope.Int static method), 87
coerce() (safeScope.List static method), 91
coerce() (safeScope.Map static method), 92
coerce() (safeScope.Set static method), 93
coerce() (safeScope.Str static method), 83
coerce() (safeScope.VarSlot static method), 95
coerce() (safeScope.Void static method), 96
CoerceExpr() (in module safeScope), 105
CompareExpr() (in module safeScope), 105
complement() (safeScope.Int method), 88
contains() (safeScope.Bytes method), 90
contains() (safeScope.List method), 91
contains() (safeScope.Map method), 92
contains() (safeScope.Set method), 94
contains() (safeScope.Str method), 84
cos() (safeScope.Double method), 86
currentProcess (in module __entrypoint_io__), 114
currentRuntime (in module __entrypoint_io__), 113
curryCall() (in module safeScope), 110
CurryExpr() (in module safeScope), 105
currySend() (in module safeScope), 110
D
DeepFrozen (in module safeScope), 103
DefExpr() (in module safeScope), 105
different() (in module safeScope), 110
diverge() (safeScope.List method), 91
diverge() (safeScope.Map method), 92
diverge() (safeScope.Set method), 94
Double (class in safeScope), 86
E
eject() (in module safeScope), 102
ejector, 69
ejector : Coercion, 118
Empty (in module safeScope), 96
empty() (safeScope.List method), 91
empty() (safeScope.Map method), 92
empty() (safeScope.Set method), 94
double() (safeScope.Set method), 94
evaluation semantics, 66
extException() (in module safeScope), 99, 100
ExitExpr() (in module safeScope), 105
expression, 66
extractGuard() (in module safeScope), 96, 97
extractGuard() (safeScope.FinalSlot static method), 95
extractGuard() (safeScope.List static method), 91
extractGuard() (safeScope.Set static method), 93
extractGuard() (safeScope.VarSlot static method), 95
extractGuards() (in module safeScope), 94
extractGuards() (safeScope.Any static method), 96
extractGuards() (safeScope.Map static method), 92
extractValue() (in module safeScope), 96

FailureList() (in module safeScope), 109
false (in module safeScope), 97
fetch() (safeScope.Map method), 92
FinallyExpr() (in module safeScope), 105
FinalPattern() (in module safeScope), 105
FinalSlot (class in safeScope), 95
floor() (safeScope.Double method), 87
floorDivide() (safeScope.Double method), 87
floorDivide() (safeScope.Int method), 88
FinalSlot (class in safeScope), 95
floor() (safeScope.Double method), 87
floorDivide() (safeScope.Double method), 87
floorDivide() (safeScope.Int method), 88
ForExpr() (in module safeScope), 105
ForwardExpr() (in module safeScope), 105
getAddrInfo (in module __entrypoint_io__), 113
guard : Coercion, 118

G
geq() (in module safeScope), 109
get() (in module safeScope), 95–97
get() (safeScope.Bytes method), 90
get() (safeScope.FinalSlot static method), 95
get() (safeScope.List static method), 91
get() (safeScope.Map method), 93
get() (safeScope.Map method), 93
get() (safeScope.Map method), 93
get() (safeScope.Set static method), 92
get() (safeScope.Str method), 84
get() (safeScope.VarSlot static method), 95
getAddrInfo (in module __entrypoint_io__), 113
getArguments() (in module __entrypoint_io__), 114
getAstBuilder() (in module safeScope), 101
getAstGuard() (in module safeScope), 109
getCategory() (safeScope.Char method), 85
getCrypt() (in module __entrypoint_io__), 113
getDocstring() (safeScope.Binding static method), 104
getDocstring() (safeScope.Bool method), 83
getDocstring() (safeScope.FinalSlot static method), 95
getDocstring() (safeScope.VarSlot static method), 95
getDocstring() (safeScope.Void static method), 96
getEnvironment() (in module __entrypoint_io__), 114
getExpr() (in module safeScope), 106
getExprGuard() (in module safeScope), 109
getGuard() (safeScope.FinalSlot static method), 95
getGuard() (safeScope.VarSlot static method), 95
getHeapStatistics() (in module __entrypoint_io__), 114
getKeys() (safeScope.Map method), 93
getMethods() (safeScope.Any static method), 96
getMethods() (safeScope.Binding static method), 104
getMethods() (safeScope.Bool static method), 83
getMethods() (safeScope.FinalSlot static method), 95
getMethods() (safeScope.VarSlot static method), 95
getMethods() (safeScope.Void static method), 96
getNamePatternGuard() (in module safeScope), 109
getNounGuard() (in module safeScope), 109
getPatternGuard() (in module safeScope), 109
getPID() (in module __entrypoint_io__), 114
getReactorStatistics() (in module __entrypoint_io__), 113
getSpan() (safeScope.Str method), 84
getValues() (safeScope.Map method), 93
getguard : Coercion, 118

H
HideExpr() (in module safeScope), 106
IfExpr() (in module safeScope), 106
IgnorePattern() (in module safeScope), 106
indexOf() (safeScope.Bytes method), 90
indexOf() (safeScope.List method), 91
indexOf() (safeScope.Str method), 84
Infinity (in module safeScope), 97
Int (class in safeScope), 87
InterfaceExpr() (in module safeScope), 106
interrupt() (in module __entrypoint_io__), 114
isBroken() (in module safeScope), 103
isDeepFrozen() (in module safeScope), 103
isEmpty() (safeScope.Bytes method), 90
isEventual() (in module safeScope), 103
isFar() (in module safeScope), 103
isNear() (in module safeScope), 103
isResolved() (in module safeScope), 103
isSelfish() (in module safeScope), 103
isSelfless() (in module safeScope), 103
isSettled() (in module safeScope), 103
isSettled() (in module safeScope), 103
isSettled() (in module safeScope), 110

isZero() (safeScope.Double method), 87
isZero() (safeScope.Int method), 88

J
join() (safeScope.Bytes method), 90
join() (safeScope.List method), 91
join() (safeScope.Str method), 84

K
Kernel-Monte, 66

L
last() (safeScope.List method), 91
lastIndexOf() (safeScope.Bytes method), 90
lastIndexOf() (safeScope.Str method), 84
leq() (in module safeScope), 110
lessThan() (in module safeScope), 110
lexical scoping, 67
List (class in safeScope), 90
ListComprehensionExpr() (in module safeScope), 106
ListExpr() (in module safeScope), 106
ListPattern() (in module safeScope), 106
LiteralExpr() (in module safeScope), 106
log() (safeScope.Double method), 87

M
M (in module safeScope), 101
makeAuditorKit() (in module safeScope), 104
makeBrandPair (in module safeScope), 100
makeEmptyRegion() (safeScope.Bytes static method), 89
makeEmptyRegion() (safeScope.Char static method), 85
makeEmptyRegion() (safeScope.Double static method), 86
makeEmptyRegion() (safeScope.Int static method), 87
makeEmptyRegion() (safeScope.Str static method), 83
makeFileResource (in module __entrypoint_io__), 113
makeLazySlot (in module safeScope), 99
makePair() (in module safeScope), 111
makeProcess (in module __entrypoint_io__), 114
makeProxy() (in module safeScope), 103
makeRegion() (safeScope.Bytes static method), 89
makeRegion() (safeScope.Char static method), 85
makeRegion() (safeScope.Double static method), 86
makeRegion() (safeScope.Int static method), 87
makeRegion() (safeScope.Str static method), 83
makeStdErr (in module __entrypoint_io__), 112
makeStdIn (in module __entrypoint_io__), 112
makeStdOut (in module __entrypoint_io__), 112
makeTCP4ClientEndpoint (in module __entrypoint_io__), 113
makeTCP4ServerEndpoint (in module __entrypoint_io__), 113
makeTraversalKey() (in module safeScope), 110
Map (class in safeScope), 92
MapComprehensionExpr() (in module safeScope), 106
MapExpr() (in module safeScope), 106
MapExprAssoc() (in module safeScope), 106
MapExprExport() (in module safeScope), 106
MapPattern() (in module safeScope), 106
MapPatternAssoc() (in module safeScope), 106
MapPatternImport() (in module safeScope), 106
Matcher() (in module safeScope), 106
matchMaker() (in module safeScope), 100, 101
max() (safeScope.Char method), 85
max() (safeScope.Int method), 88
message, 69, 118
MessageDesc() (in module safeScope), 107
MetaContextExpr() (in module safeScope), 107
MetaStateExpr() (in module safeScope), 107
Method() (in module safeScope), 107
MethodCallExpr() (in module safeScope), 107
min() (safeScope.Char method), 85
min() (safeScope.Int method), 88
MismatchExpr() (in module safeScope), 107
mod() (safeScope.Int method), 88
modPow() (safeScope.Int method), 88
Module() (in module safeScope), 107
multiply() (safeScope.Bytes method), 90
multiply() (safeScope.Double method), 87
multiply() (safeScope.Int method), 88
multiply() (safeScope.List method), 91
multiply() (safeScope.Str method), 84

N
name, 67
NamedArg() (in module safeScope), 107
NamedArgExport() (in module safeScope), 107
NamedParam() (in module safeScope), 107
NamedParamImport() (in module safeScope), 107
NaN (in module safeScope), 97
Near (in module safeScope), 104
negate() (safeScope.Double method), 87
negate() (safeScope.Int method), 88
next() (in module safeScope), 102
next() (safeScope.Char method), 85
next() (safeScope.Int method), 89
noun, 67
NounExpr() (in module safeScope), 107
null (in module safeScope), 97
NullOk (in module safeScope), 96

O
object, 66
ObjectExpr() (in module safeScope), 107
op__cmp() (safeScope.Bytes method), 90
op__cmp() (safeScope.Bytes static method), 89
op__cmp() (safeScope.Char method), 85
op__cmp() (safeScope.Char static method), 85
op__cmp() (safeScope.Double method), 87
op__cmp() (safeScope.Double static method), 86
op__cmp() (safeScope.Int method), 89
op__cmp() (safeScope.Int static method), 88
op__cmp() (safeScope.List method), 91
op__cmp() (safeScope.Set method), 94
op__cmp() (safeScope.Str method), 84
op__cmp() (safeScope.Str static method), 83
op__thru() (in module safeScope), 98
op__till() (in module safeScope), 98
optProblem() (in module safeScope), 103
optSame() (in module safeScope), 110
or() (safeScope.Int method), 89
or() (safeScope.Map method), 93
or() (safeScope.Set method), 94
OrExpr() (in module safeScope), 107
Pair (in module safeScope), 94
ParamDesc() (in module safeScope), 107
passes() (in module safeScope), 104
pattern, 66
patternHole() (in module safeScope), 100, 101
PatternHoleExpr() (in module safeScope), 107
PatternHolePattern() (in module safeScope), 107
pow() (safeScope.Double method), 87
pow() (safeScope.Int method), 89
PrefixExpr() (in module safeScope), 107
previous() (safeScope.Char method), 86
previous() (safeScope.Int method), 89
prize : Coercion, 118
promise() (in module safeScope), 103
promiseAllFulfilled (in module safeScope), 103
put() (safeScope.List method), 91
Q
QL, 118
QP, 118
QuasiExprHole() (in module safeScope), 107
quasiliteral, 118
quasiparser, 118
QuasiParserExpr() (in module safeScope), 107
QuasiParserPattern() (in module safeScope), 107
QuasiPatternHole() (in module safeScope), 108
QuasiText() (in module safeScope), 108
quote() (safeScope.Char method), 86
quote() (safeScope.Str method), 84
R
RangeExpr() (in module safeScope), 108
Ref (in module safeScope), 102
replace() (safeScope.Bytes method), 90
replace() (safeScope.Str method), 84
reverse() (safeScope.List method), 92
reverse() (safeScope.Map method), 93
run() (in module __entrypoint_io__), 112–114
run() (in module safeScope), 97–100, 102, 103, 109–111
S
safeScope (module), 83
Same (in module safeScope), 96
sameEver() (in module safeScope), 110
SameExpr() (in module safeScope), 108
SamePattern() (in module safeScope), 108
sameYet() (in module safeScope), 110
scope, 66
Script() (in module safeScope), 108
Selfless (in module safeScope), 104
semantics, 66
send() (in module safeScope), 101
SendExpr() (in module safeScope), 108
sendOnly() (in module safeScope), 102
sendTimestamp() (in module __entrypoint_io__), 112
SeqExpr() (in module safeScope), 108
Set (class in safeScope), 93
shiftLeft() (safeScope.Int method), 89
shiftRight() (safeScope.Int method), 89
sin() (safeScope.Double method), 87
size() (safeScope.Bytes method), 90
size() (safeScope.List method), 92
size() (safeScope.Map method), 93
size() (safeScope.Set method), 94
size() (safeScope.Str method), 84
slice() (safeScope.Bytes method), 90
slice() (safeScope.List method), 92
slice() (safeScope.Map method), 93
slice() (safeScope.Set method), 94
slice() (safeScope.Str method), 84
slot, 63, 67
slot object, 63
SlotExpr() (in module safeScope), 108
SlotPattern() (in module safeScope), 108
snapshot() (safeScope.List method), 92
snapshot() (safeScope.Map method), 93
snapshot() (safeScope.Set method), 94
sort() (safeScope.List method), 92
sortKeys() (safeScope.Map method), 93
sortValues() (safeScope.Map method), 93
spaceOfGuard() (in module safeScope), 99
spaceOfValue() (in module safeScope), 99
split() (safeScope.Bytes method), 90
split() (safeScope.Str method), 84
sqrt() (safeScope.Double method), 87  
stale stack frames, 67  
startsWith() (safeScope.Str method), 84  
state() (in module safeScope), 103  
static scope, 66  
stderr() (in module __entrypoint_io__), 112  
stdin() (in module __entrypoint_io__), 112  
stdio (in module __entrypoint_io__), 112  
stdout() (in module __entrypoint_io__), 112  
Str (class in safeScope), 83  
SubrangeGuard (in module safeScope), 97  
subtract() (safeScope.Bytes static method), 89  
subtract() (safeScope.Char method), 86  
subtract() (safeScope.Char static method), 85  
subtract() (safeScope.Double method), 87  
subtract() (safeScope.Double static method), 86  
subtract() (safeScope.Int method), 89  
subtract() (safeScope.Int static method), 88  
subtract() (safeScope.Set method), 94  
subtract() (safeScope.Str static method), 83  
SuchThatPattern() (in module safeScope), 108  
supersetOff() (in module safeScope), 104  
supersetOff() (safeScope.Any static method), 96  
supersetOff() (safeScope.Binding static method), 104  
supersetOff() (safeScope.Bool static method), 83  
supersetOff() (safeScope.FinalSlot static method), 95  
supersetOff() (safeScope.VarSlot static method), 95  
supersetOff() (safeScope.Void static method), 96  
SwitchExpr() (in module safeScope), 108  
syntactic expansion, 66  

tan() (safeScope.Double method), 87  
TempNounExpr() (in module safeScope), 108  
throw (in module safeScope), 102  
Timer (in module __entrypoint_io__), 112  
To() (in module safeScope), 108  
toBytes() (safeScope.Bytes method), 87  
toLowerCase() (safeScope.Bytes method), 90  
toLowerCase() (safeScope.Str method), 84  
toQuote() (in module safeScope), 102  
toString() (in module safeScope), 102  
toUpperCase() (safeScope.Bytes method), 90  
toUpperCase() (safeScope.Str method), 85  
trace (in module safeScope), 99  
traceln (in module safeScope), 99  
Transparent (in module safeScope), 104  
trim() (safeScope.Bytes method), 90  
trim() (safeScope.Str method), 85  
true (in module safeScope), 97  
TryExpr() (in module safeScope), 108  

U  
unification, 72  
unsafeNow() (in module __entrypoint_io__), 112  
unsealException (in module __entrypoint_io__), 113  

V  
valueHole() (in module safeScope), 100, 101  
ValueHoleExpr() (in module safeScope), 108  
ValueHolePattern() (in module safeScope), 108  
valueMaker() (in module safeScope), 100, 101  
VarPattern() (in module safeScope), 108  
VarSlot (class in safeScope), 95  
verb, 119  
VerbAssignExpr() (in module safeScope), 108  
ViaPattern() (in module safeScope), 108  
Void (class in safeScope), 96  
Vow (in module safeScope), 96  

W  
whenBroken() (in module safeScope), 103  
whenBrokenOnly() (in module safeScope), 103  
WhenExpr() (in module safeScope), 108  
whenResolved() (in module safeScope), 103  
whenResolvedOnly() (in module safeScope), 103  
WhileExpr() (in module safeScope), 109  
with() (safeScope.Bytes method), 90  
with() (safeScope.List method), 92  
with() (safeScope.Map method), 93  
with() (safeScope.Set method), 94  
with() (safeScope.Str method), 85  
withDefault() (in module safeScope), 110  
without() (safeScope.Map method), 93  
without() (safeScope.Set method), 94  
withRadix() (in module safeScope), 98  

X  
xor() (safeScope.Int method), 89