

The Mellifera Programming Language

Commit 9d7db45aa2df8d4b668c374a49c552bcc3a80c61
Commit Date 2026-09-01

Contents

README.md	3
overview.mf	13
GRAMMAR	29
LICENSE	33
Makefile	35
go.mod	37
mellifera.go	39
cmd/mf/mf.go	215
cmd/wasm/wasm.go	229
bin/mf-test	247
examples/basic-types-and-values.mf	249
examples/creating-references.mf	251
examples/depth-first-search.mf	253
examples/exceptions.mf	255
examples/explicit-references.mf	257
examples/fibonacci-iterative.mf	259
examples/fibonacci-recursive-memoized.mf	261
examples/fibonacci-recursive.mf	263
examples/fizzbuzz.mf	265
examples/greeter.mf	267
examples/hello.mf	269
examples/program-arguments.mf	271
examples/regular-expressions.mf	273
examples/sieve-of-eratosthenes.mf	275
examples/user-defined-iterators.mf	277
examples/user-defined-types.mf	279
examples/value-semantics-and-structural-equality.mf	281
examples/word-count-simple.mf	283
examples/word-count.mf	285

README.md

```
1  ![The Mellifera Programming Language](https://ashn.dev/mellifera/mellifera-logo.svg)
2
3  Mellifera is a scripting language featuring mutable value semantics with
4  explicit references, structural equality, copy-on-write data sharing, strong
5  dynamic typing, and a lightweight nominal type system with structural
6  protocols.
7
8  ```
9  println("hello 🐝");
10 ```
11
12 ```
13 $ mf examples/hello.mf
14 hello 🐝
15 ```
16
17 Mellifera aims to be an enjoyable language for writing small scripts and CLI
18 utilities. Here is an example word counting program in Mellifera showcasing
19 some of the syntax and semantics of the language.
20
21 ```
22 #!/usr/bin/env mf
23 # usage: cat FILE | word-count-simple.mf
24 let words = re::split(input(), r`\s+`)
25   .map(function(word) {
26     return re::replace(word.to_lower(), r`[^\\w!]'`, "");
27   })
28   .filter(function(word) {
29     return word.count() != 0;
30   });
31
32 let counts = Map{};
33 for word in words {
34   counts[word] = counts.get(word, 0) + 1;
35 }
36
37 let ordered = counts
38   .pairs()
39   .sorted_by(function(lhs, rhs) {
40     return rhs.value - lhs.value;
41   });
42 for pair in ordered {
43   println("${pair.key} {pair.value}");
44 }
45 ```
46
47 ```
48 $ cat /tmp/peter-piper.txt
49 Peter Piper picked a peck of pickled peppers,
50 A peck of pickled peppers Peter Piper picked;
51 If Peter Piper picked a peck of pickled peppers,
52 Where's the peck of pickled peppers Peter Piper picked?
53
54 $ cat /tmp/peter-piper.txt | mf examples/word-count-simple.mf
55 peter 4
56 piper 4
57 picked 4
58 peck 4
59 of 4
```

```

60 pickled 4
61 peppers 4
62 a 3
63 if 1
64 where's 1
65 the 1
66 ```
67
68 Mellifera features value semantics, meaning assignment operations copy the
69 contents of a value when executed. After an assignment statement such as `a =
70 b`, `a` and `b` will contain separate copies of the same value. Mellifera also
71 performs equality comparisons based on structural equality, so two values are
72 considered to be equal if they have the same contents.
73
74 ```
75 let x = ["foo", {"bar": 123}, "baz"];
76 let y = x; # x is assigned to y by copy
77 println(`${x} is {x}`);
78 println(`${y} is {y}`);
79 # x and y are separate values that are structurally equal
80 println(`${x} == y is {x == y}`);
81
82 print("\n");
83
84 # updates to x and y do not affect each other, because they are separate values
85 x[0] = "abc";
86 y[1]["bar"] = "xyz";
87 println(`${x} is {x}`);
88 println(`${y} is {y}`);
89 # x and y are no longer structurally equal as their contents now differ
90 println(`${x} == y is {x == y}`);
91
92 print("\n");
93
94 let z = ["foo", {"bar": "xyz"}, "baz"];
95 println(`${z} is {z}`);
96 # y and z are separate values that are structurally equal
97 println(`${y} == z is {y == z}`);
98 ```
99
100 ```
101 $ mf examples/value-semantics-and-structural-equality.mf
102 x is ["foo", {"bar": 123}, "baz"]
103 y is ["foo", {"bar": 123}, "baz"]
104 x == y is true
105
106 x is ["abc", {"bar": 123}, "baz"]
107 y is ["foo", {"bar": "xyz"}, "baz"]
108 x == y is false
109
110 z is ["foo", {"bar": "xyz"}, "baz"]
111 y == z is true
112 ```
113
114 Unlike most scripting languages, in which reference semantics are the default
115 way of assigning and passing around composite data, Mellifera uses mutable
116 value semantics with explicit references. One may obtain a reference to a value
117 using the postfix `.&` operator, then later dereference that reference-value to
118 get the original referenced value using the postfix `.*` operator.
119
120 ```
121 let pass_by_value = function(val) {
122   println(`[inside pass_by_value] val starts as {val}`);
123
124   # Here val is a copy, so the mutation performed by `val.push` will not
125   # change the original value outside of this function's lexical scope.
126   val.push(123);

```

```

127     println($"[inside pass_by_value] val ends as {val}");
128 };
129
130
131 let pass_by_reference = function(ref) {
132     println($"[inside pass_by_reference] ref starts as {ref} with referenced value
    ↪ {ref.*}");
133
134     # Here ref is a reference to the original value outside of this function's
135     # lexical scope, so mutation performed by `ref.*.push` will change that
136     # original value.
137     ref.*.push(123); # .* is the postfix dereference operator
138
139     println($"[inside pass_by_reference] ref ends as {ref} with referenced value
    ↪ {ref.*}");
140 };
141
142 let x = ["foo", "bar", "baz"];
143 println($"[outside pass_by_value] x starts as {x}");
144 pass_by_value(x);
145 println($"[outside pass_by_value] x ends as {x}");
146
147 print("\n");
148
149 let y = ["abc", "def", "ghi"];
150 println($"[outside pass_by_reference] y starts as {y}");
151 pass_by_reference(y.&); # .& is the postfix reference operator
152 println($"[outside pass_by_reference] y ends as {y}");
153 ```
154
155 ```
156 $ mf examples/explicit-references.mf
157 [outside pass_by_value] x starts as ["foo", "bar", "baz"]
158 [inside pass_by_value] val starts as ["foo", "bar", "baz"]
159 [inside pass_by_value] val ends as ["foo", "bar", "baz", 123]
160 [outside pass_by_value] x ends as ["foo", "bar", "baz"]
161
162 [outside pass_by_reference] y starts as ["abc", "def", "ghi"]
163 [inside pass_by_reference] ref starts as reference@0x372ec01e3390 with referenced value
    ↪ ["abc", "def", "ghi"]
164 [inside pass_by_reference] ref ends as reference@0x372ec01e3390 with referenced value
    ↪ ["abc", "def", "ghi", 123]
165 [outside pass_by_reference] y ends as ["abc", "def", "ghi", 123]
166 ```
167
168 Mellifera's mutable value semantics mean that references can only be created at
169 specific points in a program: using the `.&` operator on a function argument,
170 implicitly for `self` receiver arguments in methods declared with
171 `function.&(self, ...)` , or when declaring loop variables with a postfix `.&`.
172
173 ```
174 # Reference created by using the reference operator .& on a function argument.
175 let f = function(x) {
176     println($"x is {x} of type {typename(x)}");
177     println($"x.* is {x.*} of type {typename(x.*)}");
178 };
179 let v = ["foo", "bar", "baz"];
180 f(v.&);
181
182 print("\n");
183
184 # Reference created referring to each vector element in a for loop.
185 let inventory = [
186     {"item": "apples", "quantity": 12},
187     {"item": "bananas", "quantity": 30},
188     {"item": "cherries", "quantity": 8},
189 ];

```

```

190 println($`inventory starts as {inventory}`);
191 for stock.& in inventory {
192     println($`Doubling the quantity of {stock.*.item}`);
193     stock.*.quantity = stock.*.quantity * 2;
194 }
195 println($`inventory ends as {inventory}`);
196
197 print("\n");
198
199 # Reference created referring to each map value in a for loop.
200 let bank_accounts = {
201     "Ashley": {"checking": 350, "savings": 1500},
202     "Joseph": {"checking": 120, "savings": 45},
203 };
204 println($`bank_accounts starts as {bank_accounts}`);
205 for person, account.& in bank_accounts {
206     println($`Adding $100 to {person}'s savings account`);
207     account.*.savings = account.*.savings + 100;
208 }
209 println($`bank_accounts ends as {bank_accounts}`);
210
211 print("\n");
212
213 # Reference implicitly created and passed as the first argument when a method
214 # declared with function.& is invoked with the dot operator.
215 let person = type {
216     .init = function(name, age) {
217         return new person { .name = name, .age = age };
218     },
219     .birthday = function.&(self) {
220         println($`Happy birthday {self.name}! 🎂`);
221         self.*.age = self.*.age + 1;
222     },
223 };
224 let ashley = person::init("Ashley", 21);
225 println($`ashley starts as {ashley}`);
226 ashley.birthday(); # ashley.& is implicitly passed as self to person::birthday()
227 println($`ashley ends as {ashley}`);
228 ```
229
230 ...
231 $ mf examples/creating-references.mf
232 x is reference@0x4d9c24eb58a0 of type reference
233 x.* is ["foo", "bar", "baz"] of type vector
234
235 inventory starts as [{"item": "apples", "quantity": 12}, {"item": "bananas", "quantity":
236   ↪ 30}, {"item": "cherries", "quantity": 8}]
237 Doubling the quantity of apples
238 Doubling the quantity of bananas
239 Doubling the quantity of cherries
240 inventory ends as [{"item": "apples", "quantity": 24}, {"item": "bananas", "quantity":
241   ↪ 60}, {"item": "cherries", "quantity": 16}]
242
243 bank_accounts starts as {"Ashley": {"checking": 350, "savings": 1500}, "Joseph":
244   ↪ {"checking": 120, "savings": 45}}
245 Adding $100 to Ashley's savings account
246 Adding $100 to Joseph's savings account
247 bank_accounts ends as {"Ashley": {"checking": 350, "savings": 1600}, "Joseph":
248   ↪ {"checking": 120, "savings": 145}}
249
250 ashley starts as {"name": "Ashley", "age": 21}
251 Happy birthday Ashley! 🎂
252 ashley ends as {"name": "Ashley", "age": 22}
253 ```
254
255 References cannot be assigned to a new variable, stored in a container, closed
256 over by a function, or propagated up the call stack with return or error

```

```

253 statements. Because of this, the language guarantees that a reference can never
254 outlive the scope in which it was created, without requiring escape analysis.
255
256 ```
257 $ mf -c 'let x = 123; let f = function(ref) { let y = ref; }; f(x.&);'
258 [<command>, line 1] error: attempted assignment statement with type reference to number
259 ...within f@ [<command>, line 1] called from <command>, line 1
260
261 $ mf -c 'let x = 123; let f = function(ref) { let y = [ref]; }; f(x.&);'
262 [<command>, line 1] error: invalid vector construction with element
263   ↳ reference@0x24739de8d410
264 ...within f@ [<command>, line 1] called from <command>, line 1
265
266 $ mf -c 'let x = 123; let f = function(ref) { let y = {"foo": ref}; }; f(x.&);'
267 [<command>, line 1] error: invalid map construction with key "foo" and value
268   ↳ reference@0x193422f0d410
269 ...within f@ [<command>, line 1] called from <command>, line 1
270
271 $ mf -c 'let x = 123; let f = function(ref) { let y = {ref}; }; f(x.&);'
272 [<command>, line 1] error: invalid set construction with element reference@0x7173f42db410
273 ...within f@ [<command>, line 1] called from <command>, line 1
274
275 $ mf -c 'let x = 123; let f = function(ref) { function() {}; }; f(x.&);'
276 [<command>, line 1] error: {"message": "function closes over a reference", "references":
277   ↳ [{"name": "ref", "location": {"file": "<command>", "line": 1}]}
278 ...within f@ [<command>, line 1] called from <command>, line 1
279
280 $ mf -c 'let x = 123; let f = function(ref) { return ref; }; f(x.&);'
281 [<command>, line 1] error: attempted return statement with type reference to number
282 ...within f@ [<command>, line 1] called from <command>, line 1
283
284 $ mf -c 'let x = 123; let f = function(ref) { error ref; }; f(x.&);'
285 [<command>, line 1] error: attempted error statement with type reference to number
286 ...within f@ [<command>, line 1] called from <command>, line 1
287
288 ```
289
290 Mellifera contains a small number of core built-in types with first-class
291 language support: null, boolean, number, string, regular expression, vector,
292 map, set, and function, as well as a second-class reference type. These
293 built-in types are sufficient for most simple programs, but when the need
294 arises for more structured data, users have the option to define custom types
295 with specific behavior.
296
297 ```
298
299 let vec2 = type {
300   .fixed = function(self, ndigits) {
301     return new vec2 {
302       .x = self.x.fixed(ndigits),
303       .y = self.y.fixed(ndigits),
304     };
305   },
306   .magnitude = function(self) {
307     return math::sqrt(self.x * self.x + self.y * self.y);
308   },
309 };
310
311 let va = new vec2 {.x = 3, .y = 4};
312 println(`${va} is {va} with type {typename(va)}`);
313 println(`${va.magnitude()} is {va.magnitude()}`);
314
315 print("\n");
316
317 let vb = new vec2 {.x = math::e, .y = -math::pi};
318 println(`${vb} is {vb} with type {typename(vb)}`);
319 println(`${vb.magnitude()} is {vb.magnitude()}`);
320 println(`${vb.fixed(3)} is {vb.fixed(3)}`);
321
322 ```

```

```

317   ``
318   ``
319   $ mf examples/user-defined-types.mf
320   va is {"x": 3, "y": 4} with type vec2
321   va.magnitude() is 5
322   ``
323   vb is {"x": 2.718281828459045, "y": -3.141592653589793} with type vec2
324   vb.magnitude() is 4.154354402313313
325   vb.fixed(3) is {"x": 2.718, "y": -3.142}
326   ``
327   ``
328   ``
329   let fizzbuzz = type extends iterator {
330     .init = function(n, max) {
331       return new fizzbuzz {"n": n, "max": max};
332     },
333     .next = function.&(self) {
334       let n = self.n;
335       if n > self.max {
336         error null; # error null signals end-of-iteration
337       }
338       self.n = self.n + 1;
339       if n % 3 == 0 and n % 5 == 0 {
340         return "fizzbuzz";
341       }
342       if n % 3 == 0 {
343         return "fizz";
344       }
345       if n % 5 == 0 {
346         return "buzz";
347       }
348       return n;
349     },
350   };
351   ``
352   let fb = fizzbuzz::init(1, 15);
353   for x in fb {
354     println(x);
355   }
356   ``
357   ``
358   ``
359   $ mf examples/user-defined-iterators.mf
360   1
361   2
362   fizz
363   4
364   buzz
365   fizz
366   7
367   8
368   fizz
369   buzz
370   11
371   fizz
372   13
373   14
374   fizzbuzz
375   ``
376   ``
377   Mellifera is intended to be a practical language with reasonable
378   exception-based error handling and pleasant top-level error traces for when
379   things go wrong.
380   ``
381   ``
382   try {
383     fs::read("/path/to/file/that/does/not/exist.txt");

```

```

384 }
385 catch err {
386     eprintln($"error: {err}");
387 }
388
389 eprint("\n");
390
391 let g = function() {
392     let f = function() {
393         error "oopsie";
394     };
395     f();
396 };
397 function() {
398     let h = function() {
399         g();
400     };
401     h();
402 }();
403 ```
404
405 ```
406 $ mf examples/exceptions.mf
407 error: failed to read file "/path/to/file/that/does/not/exist.txt" (file not found)
408
409 [examples/exceptions.mf, line 12] error: oopsie
410 ...within f@[examples/exceptions.mf, line 11] called from examples/exceptions.mf, line 14
411 ...within g@[examples/exceptions.mf, line 10] called from examples/exceptions.mf, line 18
412 ...within h@[examples/exceptions.mf, line 17] called from examples/exceptions.mf, line 20
413 ...within function@[examples/exceptions.mf, line 16] called from examples/exceptions.mf,
414 ↪ line 21
415 ```
416
417 A brief language overview can be found in `overview.mf`, and additional example
418 programs can be found under the `examples` directory.
419
420 ## Installing
421
422 Build the Mellifera interpreter + associated tooling and install Mellifera to
423 the location specified by `MELLIFERA_HOME` (default `$HOME/.mellifera`):
424
425 ```
426 make build install # Install to $HOME/.mellifera
427 make build install MELLIFERA_HOME=/opt/mellifera # Install to /opt/mellifera
428 ```
429
430 Then, add the following snippet to your `.bashrc` (or equivalent shell
431 configuration file), replacing `$HOME/.mellifera` with your chosen
432 `MELLIFERA_HOME` directory if installing to a non-default `MELLIFERA_HOME`
433 location:
434
435 ```
436 export MELLIFERA_HOME="$HOME/.mellifera"
437 export PATH="$MELLIFERA_HOME/bin:$PATH"
438 ```
439
440 Finally, open a new interactive shell and verify the Mellifera interpreter was
441 successfully installed with:
442
443 ```
444 $ mf --command 'println("hello 🐝");'
445 hello 🐝
446 ```
447
448 ## Development
449
450 Mellifera currently has two implementations: a library & interpreter written in

```

```

450 Go (`mellifera.go` and `cmd/mf/mf.go`), which serves as the primary language
451 implementation, and an older reference interpreter written in Python (`mf.py`),
452 which served as the original language implementation during prototyping. In
453 order to ensure that Mellifera does not depend on a particular host language,
454 both implementations are updated to support the same core set of language
455 features and built-in functions. Most users will want to use the Go
456 implementation, and most `make` targets will default to the Go version of that
457 target. However, specific `*-go` and `*-py` versions of targets should be used
458 when hacking on either the Go implementation or Python implementation,
459 respectively.
460
461 ### General Development (Default Go Implementation)
462
463 ```
464 make build    # build standalone interpreter executable
465 make check    # run interpreter golden tests
466 make format   # format sources
467 make install  # install standalone Mellifera tooling
468 ```
469
470 ### Development on the Go Library & Interpreter
471
472 ```
473 make build-go  # build standalone interpreter executable
474 make wasm-go   # build Wasm module for embedding in the browser
475 make check-go  # run interpreter golden tests
476 make format-go # format sources using go fmt
477 make install   # install standalone Mellifera tooling
478 ```
479
480 The Go Mellifera implementation has support for running Mellifera programs in
481 the browser via [WebAssembly](https://webassembly.org). In a terminal, run the
482 following command:
483
484 ```
485 make wasm-go && python3 -m http.server
486 ```
487
488 Then, navigate to `http://localhost:8000/mellifera.html` in your web browser to
489 interact with a minimal web build of the interpreter.
490
491 ### Development on the Python Reference Interpreter
492
493 ```
494 python3 -m venv .venv-mellifera
495 . .venv-mellifera/bin/activate
496 python3 -m pip install -r requirements.txt
497
498 make check-py  # run interpreter golden tests
499 make lint-py   # lint with mypy and flake8
500 make format-py # format using black
501 ```
502
503 ### Development on Both Interpreters
504
505 There is a script, `tools/validate-compatibility.sh`, that will validate that
506 the token stream and abstract syntax tree representations produced by both
507 interpreters are identical for a large set of programs in the Mellifera
508 repository, including all example and test programs:
509
510 ```
511 sh tools/validate-compatibility.sh
512 ```
513
514 To build, format, lint, run tests, and validate compatibility between both
515 implementations, use the following one-liner:
516

```

```

517   ```
518   make clean format-go format-py lint-py build-go wasm-go check-go check-py && sh
519   ↪ tools/validate-compatibility.sh
520   ```
521   ## Contributing
522
523   Contributions are welcomed and appreciated! 🐝
524
525   One of the best ways to help Mellifera is to use the language for your own
526   projects and let me know what works and what needs improvement. Is Mellifera
527   missing a critical built-in function? Is there a misleading error message that
528   could use some attention? Did you find a bug in the interpreter? Let me know by
529   creating a [new issue](https://github.com/ashn-dot-dev/mellifera/issues/new)!
530
531   Want to assist the project through code or documentation contributions? Take a
532   look at open issues marked with the
533   [help-wanted](https://github.com/ashn-dot-dev/mellifera/issues?q=state%3Aopen%20label%3A%22help-wanted%22)
534   label for areas where your help would be particularly appreciated.
535
536   We aim for every commit in the Mellifera repository to be complete and
537   self-contained. Before submitting a pull request, please rebase and squash your
538   commits into logical units of change. For any non-trivial update, your commit
539   messages should contain information about *what* the commit does as well as
540   *why* the commit was needed. Examples of good commit messages can be found
541   [here](https://github.com/ashn-dot-dev/mellifera/commit/84dd674e2cad88c1b07ef8486b0b2cdb837b9c87)
542   and
543   [here](https://github.com/ashn-dot-dev/mellifera/commit/d89231eeabc867cdf2be3f102614255fe217e093).
544
545   ### Strict No LLM / No AI Policy
546
547   Mellifera is a language developed by humans, for humans. 🐝❤️
548
549   Use of LLMs for issues, comments, and pull requests is not permitted.
550
551   ## License and Credits
552
553   All content in this repository, unless otherwise noted, is licensed under the
554   Zero-Clause BSD license.
555
556   See LICENSE for more information.
557
558   The Mellifera logo and Melli character were designed by Natalie Jara.

```


overview.mf

```
1  # This overview will provide a brief introduction to the Mellifera programming
2  # language. You can execute this overview as a Mellifera program by running:
3  #
4  #      $ mf overview.mf
5  #
6  # from the command line with Mellifera installed, or by running:
7  #
8  #      $ ./bin/mf overview.mf
9  #
10 # from the Mellifera project root after running `make build`.
11
12 println("=====");
13 println("== Basic Types and Values ==");
14 println("=====");
15
16 # Mellifera has a single number type, which is an IEEE-754 double-precision
17 # float. Numbers can be written in decimal notation, or in hexadecimal notation
18 # if the number is an integer.
19 3.14159;
20 0xdeadbeef;
21
22 # Arithmetic operators + - * / % work with numbers how you would expect.
23 println($"123 + 456 is {123 + 456}`"); # 123 + 456 is 579
24 println($"123 - 456 is {123 - 456}`"); # 123 - 456 is -333
25 println($"123 * 456 is {123 * 456}`"); # 123 * 456 is 56088
26 println($"123 / 456 is {123 / 456}`"); # 123 / 456 is 0.26973684210526316
27 println($"123 % 456 is {123 % 456}`"); # 123 % 456 is 123
28
29 # The result of the remainder operator % takes the sign of the dividend,
30 # similar to C's remainder operator.
31 println($"+7 % +3 is {+7 % +3}`"); # +7 % +3 is 1
32 println($"+7 % -3 is {+7 % -3}`"); # +7 % -3 is 1
33 println($"-7 % +3 is {-7 % +3}`"); # -7 % +3 is -1
34 println($"-7 % -3 is {-7 % -3}`"); # -7 % -3 is -1
35
36 # Comparison operators == != <= >= < and > also work with numbers how you would
37 # expect.
38 println($"123 == 456 is {123 == 456}`"); # 123 == 456 is false
39 println($"123 != 456 is {123 != 456}`"); # 123 != 456 is true
40 println($"123 <= 456 is {123 <= 456}`"); # 123 <= 456 is true
41 println($"123 >= 456 is {123 >= 456}`"); # 123 >= 456 is false
42 println($"123 < 456 is {123 < 456}`"); # 123 < 456 is true
43 println($"123 > 456 is {123 > 456}`"); # 123 > 456 is false
44
45 # The special number values NaN and Inf represent the IEEE-754 not-a-number,
46 # and infinity, respectively.
47 NaN;
48 Inf;
49 -Inf;
50
51 # Mellifera has a string type consisting of a sequence of bytes, which usually
52 # represent UTF-8 encoded text.
53 "hello 🍌";
54
55 # Strings enclosed in double quotes may contain the following escape sequences:
56 #
57 #   \t => tab
58 #   \n => newline
59 #   \" => double quote
```

```

60 #  \\ => backslash
61 #  \xMN => hexadecimal byte 0xMN
62 println("hello\tworld"); # "hello  world"
63 println("hello\x5Fworld"); # "hello_world"
64 println("hello\x0\x9f\x90\x9dworld"); # "hello🐍world"
65
66 # Strings enclosed in backticks, called raw strings, interpret their contents
67 # verbatim without escaping.
68 println(`hello\tworld`); # hello\tworld
69
70 # Multi-line raw strings are enclosed in three backticks.
71 println(``line 1
72 line 2
73 line 3```);
74
75 # You can achieve markdown-like multi-line string literals by trimming leading
76 # and trailing whitespace from the string with the `trim` method.
77 println(``
78 line 1
79 line 2
80 line 3
81 ``.trim());
82
83 # Template strings are written by placing a $ character at the start of a
84 # string literal. These strings will evaluate the contents of expressions
85 # enclosed in { } characters and insert the stringified result into the string.
86 let name = "Ashley";
87 println($"Hello there, {name}!"); # Hello there, Ashley!
88
89 # Template strings can be used with either escaped or raw string literals, and
90 # can contain any expression. We have used template strings a number of times
91 # in this overview as a way to show the output of certain operations.
92 println($"The square root of 1764 is {math::sqrt(1764)}"); # The square root of 1764 is
93 ↪ 42
94
95 # If an evaluated expression in a template string is itself a string, then it
96 # will be printed without quotes. If you would like the quotes to be displayed,
97 # then wrap the evaluated expression in a call to the `repr()` function.
98 println($"`foo` in a template string turns into {repr("foo")}"); # "`foo` in a template string
99 ↪ turns into foo
100 println($"`repr("foo") in a template string turns into {repr(repr("foo"))}`); # `repr("foo") in a
101 ↪ template string turns into "foo"
102
103 # Strings can be concatenated with the + operator, and do not support any other
104 # arithmetic operators.
105 println("foo" + "bar"); # foobar
106 try {
107     "foo" - "bar";
108 }
109 catch err {
110     println($"`foo` - `bar` produced the error {repr(err)}"); # "`foo` - `bar` produced
111     ↪ the error "attempted - operation with types `string` and `string`"
112 }
113
114 # Comparison operators == != <= >= < and > will compare strings
115 # lexicographically, e.g. "abc" is less than "def".
116 println($"`abc` == `def` is {repr("abc" == "def")}");
117 println($"`abc` != `def` is {repr("abc" != "def")}");
118 println($"`abc` <= `def` is {repr("abc" <= "def")}");
119 println($"`abc` >= `def` is {repr("abc" >= "def")}");
120 println($"`abc` < `def` is {repr("abc" < "def")}");
121 println($"`abc` > `def` is {repr("abc" > "def")}");
122
123 # Mellifera has booleans written using the keywords true and false.
124 true;
125 false;

```

```

123 # The unary `not` operator will negate a boolean.
124 println(`${not true is {not true}}`); # not true is false
125 println(`${not false is {not false}}`); # not false is true
126
127 # The binary `and` operator produces a true value when both operands are true.
128 println(`${true and true is {true and true}}`); # true and true is true
129 println(`${true and false is {true and false}}`); # true and false is false
130 println(`${false and true is {false and true}}`); # false and true is false
131 println(`${false and false is {false and false}}`); # false and false is false
132
133 # The binary `or` operator produces a true value when either operand is true.
134 println(`${true or true is {true or true}}`); # true or true is true
135 println(`${true or false is {true or false}}`); # true or false is true
136 println(`${false or true is {false or true}}`); # false or true is true
137 println(`${false or false is {false or false}}`); # false or false is false
138
139 # Mellifera does not have a concept of truthy values seen in other scripting
140 # languages. Statements with conditional expressions in Mellifera such as
141 # if-statements and while-loops *must* be provided a boolean value.
142 if not false {
143     println("not false is true, so the block of the if-statement is executed");
144 }
145
146 try {
147     while 1 + 1 { }
148 }
149 catch err {
150     println(`${while-loop with the conditional 1 + 1 produced the error {repr(err)}}`);
151 }
152
153 # Mellifera has the null type and value, which represents the absence of
154 # another meaningful value, similar to null in JavaScript or None in Python.
155 null;
156
157 # Null is commonly returned by functions indicating that nothing was found.
158 println(`${[123, 456, 789].find(456) returns {[123, 456, 789].find(456)}}`); # [123, 456,
    ↪ 789].find(456) returns 1
159 println(`${[123, 456, 789].find("nope") returns {[123, 456, 789].find("nope")}}`); # [123,
    ↪ 456, 789].find("nope") returns null
160
161 # Mellifera supports regular expression matching using the =~ and !~ operators.
162 # These are always written as STRING =~ REGEXP or STRING !~ REGEXP with regular
163 # expression values written as strings prefixed with an r character, similar to
164 # how template strings are prefixed with a $ character.
165 if "foobar" =~ r`foo(bar|baz)(qux)?` {
166     # Regular expression capture groups are accessed with $n where n is the nth
167     # capture group in the most recently executed regular expression in scope.
168     # The result of $n will be the matched string capture if the nth capture
169     # group successfully matched, or null if that capture group did not match.
170     # The result of $0 is the entire matched string.
171     dumpLn($0); # "foobar"
172     dumpLn($1); # "bar"
173     dumpLn($2); # null
174 }
175
176 # Mellifera has three aggregate data types: vector, map, and set. We will first
177 # learn about vector, which is an ordered sequence of values, also known as a
178 # dynamic array.
179 []; # empty vector
180 [123, 456, 789];
181
182 # Vectors may contain elements of any type, including other aggregate types.
183 [123, ["foo", "bar"], null];
184
185 # Elements of a vector are accessed using the postfix [] operator, which uses
186 # zero-based indexing.
187 println(`${[123, 456, 789][1] is {[123, 456, 789][1]}}`); # [123, 456, 789][1] is 456

```

```

188
189 # You can get the number of elements in a vector using the .count() method.
190 let v = [123, 456, 789];
191 println($`v is {v}`); # v is [123, 456, 789]
192 println($`v.count() is {v.count()}`); # v.count() is 3
193
194 # You can insert an element at the end of a vector with the `push` method.
195 v.push("foo");
196 println($`v after v.push("foo") is {v}`); # v after v.push("foo") is [123, 456, 789,
    ↪ "foo"]
197
198 # You can remove an element from the end of a vector with the `pop` method.
199 let popped = v.pop();
200 println($`v after v.pop() is {v} and the popped element is {repr(popped)}`); # v after
    ↪ v.pop() is [123, 456, 789] and the popped element is "foo"
201
202 # Iteration over the elements of a vector is performed with a for statement.
203 for element in v {
204     println($`element is {repr(element)}`); # element is 123
205                                           # element is 456
206                                           # element is 789
207 }
208
209 # Vectors can be concatenated with the + operator, and do not support any other
210 # arithmetic operators.
211 println(["foo"] + ["bar"]); # ["foo", "bar"]
212 try {
213     ["foo"] - ["bar"];
214 }
215 catch err {
216     println($`["foo"] - ["bar"] produced the error {repr(err)}`); # ["foo"] - ["bar"]
    ↪ produced the error "attempted - operation with types `vector` and `vector`"
217 }
218
219 # The next aggregate data type is the map, which is an ordered collection of
220 # key-value pairs with unique keys, also known as an associative array.
221 # Key-value pairs can be written using either the syntax:
222 #
223 #   key: value
224 #
225 # or using the syntax:
226 #
227 #   .name = value
228 #
229 # where `.name` is transformed into the key "name".
230 #
231 # Maps may optionally be prefixed with the Map keyword to disambiguate them
232 # from sets, which we will see in just a moment. The empty map must be
233 # written as Map{}.
234 Map{}; # empty map
235 {"name": "Ashley", "age": 21};
236 { .name = "Ashley", .age = 21 }; # equivalent to the map above
237 Map{"name": "Ashley", "age": 21}; # equivalent to the map above
238
239 # Maps may contain keys and values of any type, including other aggregate
240 # types. Use of non-string keys is supported and actively encouraged when doing
241 # so would fit your problem domain.
242 {
243     [123, 456, 789]: [135, 246],
244     { .abc = 123 }: "you and me",
245     .worries = null,
246 };
247
248 # Elements of a map are accessed using the postfix [] operator.
249 println($`{{.name = "Ashley", .age = 21}}["name"] is {repr({.name = "Ashley", .age =
    ↪ 21}["name"])}`); # {.name = "Ashley", .age = 21}["name"] is "Ashley"
250

```

```

251 # You can get the number of elements in a map using the .count() method.
252 let m = {.name = "Ashley", .age = 21};
253 println($`m is {m}`); # m is {"name": "Ashley", "age": 21}
254 println($`m.count() is {m.count()}`); # m.count() is 2
255
256 # You can insert an element into a map with the `insert` method.
257 m.insert("profession", "student");
258 println($`m after m.insert("profession", "student") is {m}`); # m after
    ↪ m.insert("profession", "student") is {"name": "Ashley", "age": 21, "profession":
    ↪ "student"}
259
260 # You can remove an element from a map with the `remove` method.
261 let removed = m.remove("age");
262 println($`m after m.remove("age") is {m} and the removed value is {repr(removed)}`); # m
    ↪ after m.remove("age") is {"name": "Ashley", "profession": "student"} and the removed
    ↪ value is 21
263
264 # Iteration over the key-value pairs of a map is performed with a for
265 # statement. Maps are *ordered* based on the insertion order of their elements,
266 # so iterating over a map is deterministic.
267 for key, value in m {
268     println($`key is {repr(key)} and value is {repr(value)}`); # key is "name" and value
    ↪ is "Ashley"
269                                     # key is "profession" and
    ↪ value is "student"
270 }
271
272 # The final aggregate data type is the set, which is an ordered collection of
273 # unique elements. Unlike vectors, attempting to construct a set with
274 # duplicates of the same element will result in only one of those elements
275 # appearing in the set.
276 #
277 # Sets may optionally be prefixed with the Set keyword to disambiguate them
278 # from maps. The empty set *must* be written as Set{}.
279 Set{}; # empty set
280 {"foo", "bar", "baz"};
281 Set{"foo", "bar", "baz"}; # equivalent to the set above
282
283 # Sets may contain elements of any type, including other aggregate types.
284 {123, ["foo", "bar"], null};
285
286 # Duplicate elements are only represented once in a set.
287 dump($`{{123, 456, 123}} is {repr({123, 456, 123})}`);
288
289 # You can get the number of elements in a set using the .count() method.
290 let s = {123, 456, 789};
291 println($`s is {s}`); # s is {123, 456, 789}
292 println($`s.count() is {s.count()}`); # s.count() is 3
293
294 # You can insert an element into a set with the `insert` method.
295 s.insert("foo");
296 println($`s after s.insert("foo") is {s}`); # s after s.insert("foo") is {123, 456, 789,
    ↪ "foo"}
297
298 # You can remove an element from a set with the `remove` method.
299 let removed = s.remove(456);
300 println($`s after s.remove(456) is {s} and the removed value is {repr(removed)}`); # s
    ↪ after s.remove(456) is {123, 789, "foo"} and the removed value is 456
301
302 # Iteration over the elements of a set is performed with a for statement. Sets
303 # are *ordered* based on the insertion order of their elements, so iterating
304 # over a set is deterministic.
305 for element in s {
306     println($`element is {repr(element)}`); # element is 123
307                                           # element is 789
308                                           # element is "foo"
309 }

```

```

310
311 # Mellifera supports first-class functions, defined using the function keyword.
312 function(a, b) { return a + b; };
313
314 # A function declared with function.&() will expect its first parameter to be
315 # passed by reference (relevant for method calls discussed later).
316 let inc = function.&(obj) { obj.n = obj.n + 1; };
317 let foo = {"n": 0};
318 println($`foo is {foo}`); # foo is {"n": 0}
319 inc(foo.&);
320 println($`foo after being passed to {inc} by reference is {foo}`); # foo after being
    ↪ passed to inc@[overview.mf, line XXX] by reference is {"n": 1}
321 # Functions close over their environment.
322 let n = 3;
323 let addn = function(x) {
324     return x + n;
325 };
326 println($`addn(5) is {addn(5)}`); # addn(5) is 8
327
328 # If a function does not return a value with a return statement, then the
329 # result of that function is null.
330 let do_nothing = function() { };
331 println($`do_nothing() is {do_nothing()}`); # do_nothing() is null
332
333 # Mellifera has an explicit reference type representing the location of a value
334 # in memory. References are useful when you want to access a specific value,
335 # such as a vector or a map, without creating a separate copy of that value.
336 #
337 # References are obtained by using the postfix .& operator on a value in a
338 # function call argument list, or within a for-loop variable declaration
339 # (discussed in more detail later), and are dereferenced using the postfix .*
340 # operator.
341 let f = function(ref) {
342     println($`passed in reference is {ref}`);
343     ref.*["foo"] = "baz"; # mutate the referenced value
344 };
345 let m = {"foo": "bar"};
346 println($`m before being passed to f() by reference is {m}`);
347 f(m.&); # pass in a reference to the {"foo": "bar"}
348 println($`m after being passed to f() by reference is {m}`);
349
350 # For convenience, accessing elements of a referenced value with:
351 #
352 #   reference.*[index]
353 #
354 # is equivalent to the following syntax, known as auto-dereferencing:
355 #
356 #   reference[index]
357 let f = function(ref) {
358     println($`passed in reference is {ref}`);
359     ref["foo"] = "baz"; # mutate the referenced value with auto dereferencing
360 };
361 let m = {"foo": "bar"};
362 println($`m before being passed to f() by reference is {m}`);
363 f(m.&); # pass in a reference to the {"foo": "bar"}
364 println($`m after being passed to f() by reference is {m}`);
365
366 println("=====");
367 println("== Value Semantics and Structural Equality ==");
368 println("=====");
369
370 # Mellifera features value semantics, meaning assignment operations copy the
371 # contents of a value when executed. After a variable declaration or assignment
372 # statement such as:
373 #
374 #   let a = b;
375 #

```

```

376 # or:
377 #
378 #   a = b;
379 #
380 # `a` and `b` will contain separate copies of the same value. Mellifera also
381 # performs equality comparisons based on structural equality, so two values
382 # are considered to be equal if they have the same contents.
383
384 let x = ["foo", {"bar": 123}, "baz"];
385 println($`x is {x}`); # x is ["foo", {"bar": 123}, "baz"]
386
387 let y = x; # x is assigned to y by copy
388 println($`y is {y}`); # y is ["foo", {"bar": 123}, "baz"]
389
390 # x and y are separate values that are structurally equal.
391 println($`x == y is {x == y}`); # x == y is true
392
393 # Updates to x and y do not affect each other, because they are separate values
394 # with separate addresses.
395 x[0] = "abc";
396 y[1]["bar"] = "xyz";
397 println($`x is {x}`); # x is ["abc", {"bar": 123}, "baz"]
398 println($`y is {y}`); # y is ["foo", {"bar": "xyz"}, "baz"]
399
400 # x and y are no longer structurally equal as their contents now differ.
401 println($`x == y is {x == y}`); # x == y is false
402
403 let z = ["foo", {"bar": "xyz"}, "baz"];
404 println($`z is {z}`); # z is ["foo", {"bar": "xyz"}, "baz"]
405 # y and z are separate values that are structurally equal.
406 println($`y == z is {y == z}`); # y == z is true
407
408 println("=====");
409 println("== If, Elif, and Else Statements ==");
410 println("=====");
411
412 # Arms of an if statement are evaluated in order until a condition evaluates as
413 # true, at which point the body of that conditional is executed.
414 if true {
415     println("A (if)");
416 }
417
418 if false {
419     println("X");
420 }
421 elif true {
422     println("B (elif)");
423 }
424
425 if false {
426     println("X");
427 }
428 elif false {
429     println("X");
430 }
431
432 if false {
433     println("X");
434 }
435 elif false {
436     println("X");
437 }
438 elif true {
439     println("C (second-elif)");
440 }
441
442 if false {

```

```

443     println("X");
444 }
445 elif false {
446     println("X");
447 }
448 else {
449     println("D (else)");
450 }
451
452 println("=====");
453 println("== Loop Statements ==");
454 println("=====");
455
456 # Integer loops of the form:
457 #
458 #   for i in n {
459 #       # body
460 #   }
461 #
462 # iterate over integers in the half-open range [0, n).
463 for x in 3 {
464     println("$x is {x}");
465 }
466
467 # As seen previously, you can loop over the elements of a vector with a loop of
468 # the form:
469 #
470 #   for element in vector {
471 #       # body
472 #   }
473 for y in ["a", "b", "c"] {
474     println("$y is {y}");
475 }
476
477 # You can also directly update the elements of a vector by iterating over the
478 # elements of the vector with special syntax that takes a reference to the
479 # iterated-over element.
480 let v = [{"n": 123}, {"n": 456}, {"n": 789}];
481 println("$v is {v}"); # v is [{"n": 123}, {"n": 456}, {"n": 789}]
482 for y.& in v {
483     y.n = y.n + 1;
484 }
485 println("$v after update loop is {v}"); # v after update loop is [{"n": 124}, {"n": 457},
↳ [{"n": 790}]
486
487 # As seen previously, you can loop over the key-value pairs of a map with a
488 # loop of the form:
489 #
490 #   for key, value in map {
491 #       # body
492 #   }
493 for k, v in {"foo": {"n": 123}, "bar": {"n": 456}, "baz": {"n": 789}} {
494     println("$k is {k}, v is {v}");
495 }
496
497 # You can also directly update the elements of a map by iterating over the
498 # key-value pairs of the map with special syntax that takes a reference to the
499 # iterated-over value.
500 let m = {"foo": {"n": 123}, "bar": {"n": 456}, "baz": {"n": 789}};
501 println("$m is {m}"); # m is {"foo": {"n": 123}, "bar": {"n": 456}, "baz": {"n": 789}}
502 for k, v.& in m {
503     v.n = v.n + 1;
504 }
505 println("$m after update loop is {m}"); # m after update loop is {"foo": {"n": 124},
↳ "bar": {"n": 457}, "baz": {"n": 790}}
506
507 # To iterate over just the keys of a map, omit the value variable.

```

```

508 for k in {"foo": {"n": 123}, "bar": {"n": 456}, "baz": {"n": 789}} {
509     println($"k is {k}");
510 }
511
512 # As seen previously, you can loop over the elements of a set with a loop of
513 # the form:
514 #
515 #     for element in set {
516 #         # body
517 #     }
518 for z in {"a", "b", "c"} {
519     println($"z is {z}");
520 }
521
522 # You can iterate over values of custom types that define a `next` method which
523 # produces the next iterated value when called, erroring with a null value when
524 # an end-of-iteration condition is reached. Defining a custom iterator this way
525 # uses several concepts that have not yet been touched on in this overview, so
526 # for now just know that Mellifera supports custom user-defined iterators.
527 let fizzbuzz = type extends iterator {
528     .init = function(n, max) {
529         return new fizzbuzz {"n": n, "max": max};
530     },
531     .next = function.&(self) {
532         let n = self.n;
533         if n > self.max {
534             error null; # error null signals end-of-iteration
535         }
536         self.n = self.n + 1;
537         if n % 3 == 0 and n % 5 == 0 {
538             return "fizzbuzz";
539         }
540         if n % 3 == 0 {
541             return "fizz";
542         }
543         if n % 5 == 0 {
544             return "buzz";
545         }
546         return n;
547     },
548 };
549 let fb = fizzbuzz::init(1, 15);
550 println($"Executing fizzbuzz with custom iterator type {typename(fb)}...`");
551 for x in fb {
552     println(x);
553 }
554
555 # While-loops execute the body of the loop every time the while-condition
556 # evaluates as true.
557 let n = -3;
558 while n <= 3 {
559     println($"n <= 3 is true for n == {n}");
560     n = n + 1;
561 }
562
563 # The break statement will immediately jump out of the inner-most for or while
564 # loop and continue execution after that loop.
565 for x in [123, 456, 789] {
566     println($"x is {repr(x)} before the break statement`");
567     break; # break out of the loop before an error occurs
568     error "oopsie";
569 }
570
571 # The continue statement will immediately jump back to the loop conditional and
572 # continue execution with the next iteration of that loop.
573 for x in ["abc", "def", "hij"] {
574     println($"x is {repr(x)} before the continue statement`");

```

```

575     continue; # continue with the next iteration of the loop before an error occurs
576     error "oopsie";
577 }
578
579 println("=====");
580 println("== Lexical Scoping, Functions, and Closures ==");
581 println("=====");
582
583 # Variables are declared with a let statement.
584 let color = "maroon"; # declaration and assignment
585
586 # Executing the let statement above creates a variable in the current lexical
587 # scope named `color` and gives it a copy of the value "maroon" on the right
588 # hand side of the assignment operator. Once the variable has been declared,
589 # it can be re-assigned using similar syntax without the let keyword.
590 color = "teal"; # assignment without declaration
591
592 # It is an error to attempt to assign a value to a variable that has not been
593 # declared.
594 try {
595     shirt = "flannel";
596 }
597 catch err {
598     println(`${shirt = "flannel" produced the error {repr(err)}`); # shirt = "flannel"
599     ↪ produced the error "identifier `shirt` is not defined"
600 }
601
602 # Re-declaring the same variable with another let statement is explicitly
603 # allowed and encouraged. When you see `let` in a Mellifera program, think
604 # of it as a way of saying "this variable means something new".
605 let foo = 123;
606 let foo = "bar";
607
608 # Any time you see curly braces { }, a new variable scope has been declared.
609 # Within this new scope you can access variables from outer scopes, but new
610 # variables declared within that scope are not accessible from outside that
611 # scope. Declarations of new variables with the same name as a variable in an
612 # outer scope will shadow the variable in the outer scope.
613 let b = 123;
614 let c = "foo";
615 println(`${b is {repr(b)}}`); # b is 123
616 println(`${c is {repr(c)}}`); # c is "foo"
617 if true {
618     let a = 456; # a is only accessible inside this if-body scope
619     let b = 789; # b shadows the outer variable with the same name
620     c = "bar"; # c updates the variable in the outer scope
621     println(`${a inside the new scope is {repr(a)}}`); # a inside the new scope is 456
622     println(`${b inside the new scope is {repr(b)}}`); # b inside the new scope is 789
623     println(`${c inside the new scope is {repr(c)}}`); # c inside the new scope is "bar"
624 }
625 println(`${b outside that new scope is still {repr(b)}}`); # b outside that new scope is
626 ↪ still 123
627 println(`${c outside that new scope is {repr(c)}}`); # c outside that new scope is "bar"
628
629 # As we saw before, functions close over their environment, retaining access to
630 # outer scopes even after program control has left that scope.
631 let adder = function(x) {
632     # x is introduced into this scope by the function parameter list
633     return function(y) {
634         # x is still in scope and can be accessed here.
635         # y is introduced into this scope by the function parameter list
636         return x + y;
637     };
638 };
639 let add1 = adder(1);
640 println(`${add1(2) is {add1(2)}}`); # add1(2) is 3

```

```

640 println("=====");
641 println("== Try, Catch, and Error ==");
642 println("=====");
643
644 # Errors are a way of reporting unexpected conditions or invalid state within
645 # a Mellifera program. An error is generated using an error statement, and will
646 # propagate up the call stack until it is caught by the catch block of a
647 # try-catch statement.
648 try {
649     error "oopsie";
650 }
651 catch err {
652     println($"caught error {repr(err)}"); # caught error "oopsie"
653 }
654
655 # Errors can also be generated by the Mellifera runtime when an invalid
656 # operation occurs, e.g. from an invalid index out-of-bounds operation.
657 try {
658     let v = [123, 456, 789]; # vector with three elements
659     v[100]; # attempt to access a vector element with an invalid index
660 }
661 catch err {
662     println($"caught error {repr(err)}"); # caught error "invalid vector access with
663     ↪ index 100"
664 }
665
666 # Most errors generated by the Mellifera runtime and user code are strings
667 # describing the caught error. However, an error statement can use any value,
668 # including structured data in the form of a map.
669 let is_even = function(n) {
670     if not ty::is_number(n) or not n.is_integer() {
671         error {
672             .kind = "invalid argument",
673             .message = $"expected a number, received {typename(n)} with value {repr(n)}",
674         };
675     }
676     return n % 2 == 0;
677 };
678 try {
679     println($"is_even(4) is {is_even(4)}"); # is_even(4) is true
680     println($"is_even(\"foo\") is {is_even(\"foo\")}"); # ERROR!!!
681 }
682 catch err {
683     println($"caught error {repr(err)}"); # caught error {"kind": "invalid argument",
684     ↪ "message": "expected a number, received string with value \"foo\""}
685 }
686
687 # The error capture variable in a catch block can be omitted if you do not
688 # intend to use the error value.
689 try {
690     let bytes = fs::read("/path/to/file/that/does/not/exist");
691     println(bytes);
692 }
693 catch {
694     # We failed to read the file, but that may be expected behavior, so ignore
695     # the error and move on...
696 }
697
698 # If an error propagates all the way up the call stack without being caught
699 # then the error will automatically be reported alongside a stack trace.
700 # Letting an error propagate this way will terminate a Mellifera program, which
701 # would end this overview, so an example top-level error trace will be shown
702 # below without actually generating that error trace.
703 let g = function() {
704     let f = function() {
705         error "oopsie";
706     };
707 }

```

```

705     f();
706 };
707 let boom = function() {
708     let h = function() {
709         g();
710     };
711     h();
712 };
713 # boom();
714 # ^^^^^^^ Executing this line would generate an error trace similar to:
715 # [example.mf, line 3] error: oopsie
716 # ...within f@[example.mf, line 2] called from example.mf, line 5
717 # ...within g@[example.mf, line 1] called from example.mf, line 9
718 # ...within h@[example.mf, line 8] called from example.mf, line 11
719 # ...within boom@[example.mf, line 7] called from example.mf, line 13
720
721 println("=====");
722 println("== Built-in Variables and Functions ==");
723 println("=====");
724
725 # Mellifera aims to be a "batteries-included" scripting language, shipping with
726 # out-of-the-box support for common algorithms and data formats. Adding this
727 # out-of-the-box support to the core language is a work-in-progress and will
728 # improve as the language develops further.
729
730 # In this overview, you have already seen a few of the most frequently used
731 # built-in functions including `println` and `dumpln`, which will write the
732 # stringified representation of a value to the standard output stream. For
733 # string values, the `println` function will write the string data itself,
734 # whereas `dumpln` will write the escaped string data in double quotes as it
735 # would appear in Mellifera source code.
736 println("hello\nworld"); # hello
737                          # world
738 dumpln("hello\nworld"); # "hello\nworld"
739
740 # The values of NaN and Inf, representing the not-a-number and infinity values
741 # of the number type, are actually just built-in variables.
742 NaN; # this is the built-in NaN defined by the interpreter
743 let NaN = 0 / 0; # but it could have just as easily been defined in user-code
744
745 # In addition to the top-level built-ins seen so far, there are a number of
746 # built-in variables and functions grouped into categories like "math" and
747 # "random".
748 math::sin(-math::pi); # sine trigonometric function with input - $\pi$ 
749 random::integer(1, 10); # generate a random integer in the range [1, 10]
750 json::encode({
751     .name = "Ashley",
752     .profession = "student",
753     .allergies = ["seasonal (winter)", "lavender"],
754 });
755
756 println("=====");
757 println("== Custom User-Defined Types ==");
758 println("=====");
759
760 # In addition to the core Mellifera types, users can define custom types using
761 # the type keyword. In the example below, we define a new type called `vec2`
762 # representing a 2D vector, similar to what one might see in a video game
763 # development framework.
764 let vec2 = type {
765     .fixed = function(self, ndigits) {
766         return new vec2 {
767             .x = self.x.fixed(ndigits),
768             .y = self.y.fixed(ndigits),
769         };
770     },
771     .magnitude = function(self) {

```

```

772         return math::sqrt(self.x * self.x + self.y * self.y);
773     },
774 };
775
776 # Custom types are instantiated using the new keyword. Our vec2 expects to
777 # operate on a map with number fields x and y, so we instantiate a vec2
778 # instance using a map literal with those x and y fields.
779 let v2 = new vec2 {.x = 3, .y = 4};
780
781 # You can invoke vec2 methods on this newly created vec2 instance, as the type
782 # of the value we created is vec2 instead of map.
783 println($`v2.magnitude() is {v2.magnitude()}`);
784
785 # You can get the type of a value using the built-in `typeof` function, and the
786 # name of that type using the built-in `typename` function.
787 println($`v2 is a {typename(v2)} => {typeof(v2)}`); # v2 is a vec2 => {"fixed":
    ↪ vec2::fixed@[overview.mf, line XXX], "magnitude": vec2::magnitude@[overview.mf, line
    ↪ YYY]}
788
789 # Custom types will commonly be defined with an `init` method which acts as the
790 # canonical constructor for that type wrapping the actual new expression.
791 let vec2 = type {
792     .init = function(x, y) {
793         return new vec2 {.x = x, .y = y};
794     },
795     .fixed = function(self, ndigits) {
796         return new vec2 {
797             .x = self.x.fixed(ndigits),
798             .y = self.y.fixed(ndigits),
799         };
800     },
801     .magnitude = function(self) {
802         return math::sqrt(self.x * self.x + self.y * self.y);
803     },
804 };
805 let v2 = vec2::init(3, 4);
806 println($`v2 created with vec2::init is {v2} of type {typename(v2)}`); # v2 created with
    ↪ vec2::init is {"x": 3, "y": 4} of type vec2
807
808 # You may have noticed that sometimes methods are accessed using the ::
809 # operator, such as in the case of our `vec2::init` method invocation:
810 #
811 #     vec2::init(3, 4);
812 #
813 # and that other times methods are accessed using the . operator, such as in
814 # the case of our `v2.magnitude()` invocation. What is up with that?
815 #
816 # Melliifera has syntax sugar intended to make property access and method
817 # invocation more convenient. Since types are just fancy maps, you can access
818 # any method of a type using the same [] operator used to access the field of
819 # any other map.
820 println($`vec2["init"] is {vec2["init"]}`); # vec2["init"] is vec2::init@[overview.mf,
    ↪ line XXX]
821
822 # If we wanted to invoke our init method, we could theoretically do so using
823 # the [] operator like so.
824 println($`vec2["init"](3, 4) is {vec2["init"](3, 4)} with type {typename(vec2["init"](3,
    ↪ 4))}`); # vec2["init"](3, 4) is {"x": 3, "y": 4} with type vec2
825
826 # That is a lot of characters for a relatively common operation, so as syntax
827 # sugar the :: operator will look up a value in a map without needing to
828 # include the square brackets and quotes.
829 println($`vec2::init is {vec2::init}`); # vec2::init is vec2::init@[overview.mf, line
    ↪ XXX]
830
831 # So `vec2::init(3, 4)` is exactly equivalent to `vec2["init"](3, 4)`, just
832 # with a little bit nicer syntax.

```

```

833 println(`vec2::init(3, 4) is {vec2::init(3, 4)} with type {typename(vec2::init(3,
    ↪ 4))}`); # vec2::init(3, 4) is {"x": 3, "y": 4} with type vec2
834
835 # The . operator can also be used to access the fields of a map, similar to the
836 # :: operator.
837 println(`vec2.init is {vec2.init}`); # vec2.init is vec2::init@[overview.mf, line XXX]
838
839 # However, there is one major difference between the :: operator and the .
840 # operator: when invoking a function call on a function accessed using the .
841 # operator, a reference to the value on the left hand side of the . operator is
842 # implicitly passed to the invoked function if the function was defined to take
843 # a self parameter by reference with the function.&() syntax.
844 try {
845     vec2.init(3, 4); # arguments passed to the init function are vec2.&, 3, and 4
846 }
847 catch err {
848     println(`encountered the error {repr(err)}`); # encountered the error "invalid
    ↪ function argument count (expected 2, received 3)"
849 }
850
851 # You may ask why you would want a reference to the value on the left hand side
852 # of the . operator to be implicitly passed to the invoked function. Well, by
853 # implicitly passing a reference to that value, it is possible to write code
854 # where functions can easily act on and modify values on the left hand side of
855 # the . operator. A great example of this is with the `vector::push` function.
856 # If we wanted to push an element onto the back of a vector, we could do so
857 # using the :: operator syntax, but that would require manually taking a
858 # reference to the vector that needs to be modified.
859 let v = [123, 456, 789];
860 println(`v before vector::push(v.&, "foo") is {v}`); # v before vector::push(v.&, "foo")
    ↪ is [123, 456, 789]
861 vector::push(v.&, "foo");
862 println(`v after vector::push(v.&, "foo") is {v}`); # v after vector::push(v.&, "foo")
    ↪ is [123, 456, 789, "foo"]
863
864 # Similar to how it is much more convenient to write `vec2::init` than it is to
865 # write `vec2["init"]` everywhere, it is much more convenient to write
866 # `v.push(value)` than it is to write `vector::push(v.&, value)` everywhere.
867 let v = [123, 456, 789];
868 println(`v before v.push("foo") is {v}`); # v before v.push("foo") is [123, 456, 789]
869 v.push("foo");
870 println(`v after v.push("foo") is {v}`); # v after v.push("foo") is [123, 456, 789,
    ↪ "foo"]
871
872 # Additionally, when using the . syntax, the programmer does not need to know
873 # the exact type of a value when invoking methods. In the example above, we
874 # happen to know that `v` was a vector, but what if it was some unknown type?
875 # Maybe we have a generic function that works on any value with a `call`
876 # method. Without this implicit reference passing, we would have to sprinkle
877 # `typeof` everywhere in order to actually get to the generic functions we want
878 # to call.
879 let print_count = function(value) {
880     let count = typeof(value)::count(value); # YUCK!!!
881     println(`{repr(value)} has an element count of {count}`);
882 };
883 print_count(["foo", "bar", "baz"]); # ["foo", "bar", "baz"] has an element count of 3
884 print_count({.weather = "rainy", .temperature = "46°F"}); # {"weather": "rainy",
    ↪ "temperature": "46°F"} has an element count of 2
885
886 let print_count = function(value) {
887     let count = value.count(); # MUCH NICER!!!
888     println(`{repr(value)} has an element count of {count}`);
889 };
890 print_count(["foo", "bar", "baz"]); # ["foo", "bar", "baz"] has an element count of 3
891 print_count({.weather = "rainy", .temperature = "46°F"}); # {"weather": "rainy",
    ↪ "temperature": "46°F"} has an element count of 2
892

```

```

893 # When not invoking a function, the . operator behaves similar to the ::
894 # operator, accessing the properties of a value by name.
895 let m = {"foo": 123, "bar": 456};
896 m.foo = 789;
897 math::log10(m.bar);
898
899 # This leads to the general advice that you should use the :: operator when
900 # accessing the properties of a type and you should use the . operator when
901 # accessing the properties on an instance of a type.
902 let v2 = vec2::init(3, 4); # vec2 is a type => use ::
903 v2.x; # v2 is an instance of type vec2 => use .
904 v2.magnitude(); # v2 is an instance of type vec2 => use .
905
906 # Previously in this overview, it was mentioned that it is possible to create
907 # custom types that act as iterators by defining a type with a `next` method.
908 # The easiest way to define one of these types is by extending the built-in
909 # iterator type with an `extends` type definition. Note that Mellifera uses
910 # structural typing, so `extends` is not creating an is-a relationship that
911 # one would find in a language with inheritance. Rather, the `extends` keyword
912 # is a convenient way of saying "take all of the methods from this existing
913 # type and override some of them with any additional methods I define here".
914 let count_by_threes = type extends iterator {
915     .init = function(begin, end) {
916         return new count_by_threes {
917             .cur = begin,
918             .end = end,
919         };
920     },
921     .next = function.&(self) {
922         if self.cur >= self.end {
923             error null; # error null signals end-of-iteration
924         }
925         self.cur = self.cur + 3;
926         return self.cur - 3;
927     },
928 };
929
930 # Since our count_by_threes instance acts as an iterator, we can iterate over
931 # that instance with a for-loop.
932 let iter = count_by_threes::init(10, 32);
933 for x in iter {
934     println(x); # 10
935                 # 13
936                 # 16
937                 # 19
938                 # 22
939                 # 25
940                 # 28
941                 # 31
942 }
943
944 # Additionally, since the count_by_threes type is extended from iterator, we
945 # can use all of the iterator methods on our count_by_threes instance.
946 let iter = count_by_threes::init(10, 32);
947 let even = iter.filter(function(x) { return x % 2 == 0; }).into_vector();
948 println(`${even} is {even}`); # even is [10, 16, 22, 28]
949
950 println("=====");
951 println("== Example Code ==");
952 println("=====");
953
954 # Example recursive fibonacci implementation.
955 let fibonacci = function(x) {
956     if x == 0 or x == 1 {
957         return x;
958     }
959     return fibonacci(x - 1) + fibonacci(x - 2);

```

```

960 };
961
962 println($"fibonacci(10) is {fibonacci(10)}");
963
964 # Example merge sort implementation.
965 let sort = function(x) {
966     if x.count() <= 1 {
967         return x;
968     }
969     let mid = (x.count() / 2).trunc();
970     let lo = sort(x.slice(0, mid));
971     let hi = sort(x.slice(mid, x.count()));
972     let lo_index = 0;
973     let hi_index = 0;
974     let result = [];
975     for _ in x.count() {
976         if lo_index == lo.count() {
977             result.push(hi[hi_index]);
978             hi_index = hi_index + 1;
979         }
980         elif hi_index == hi.count() {
981             result.push(lo[lo_index]);
982             lo_index = lo_index + 1;
983         }
984         elif lo[lo_index] < hi[hi_index] {
985             result.push(lo[lo_index]);
986             lo_index = lo_index + 1;
987         }
988         else {
989             result.push(hi[hi_index]);
990             hi_index = hi_index + 1;
991         }
992     }
993     return result;
994 };
995
996 let vec = [6, 8, 3, 1, 7, 2, 4, 5, 9];
997 println($"unsorted vector is {vec}");
998 println($"sorted vector is {sort(vec)}");
999
1000 # Example implementation counting the frequency of words in a string. Returns a
1001 # map of individual words to that word's occurrence count.
1002 let word_count = function(s) {
1003     let words = re::split(s, r`\s+`)
1004     .map(function(word) {
1005         return re::replace(word.to_lower(), r`[^\\w]`, "");
1006     })
1007     .filter(function(word) {
1008         return word.count() != 0;
1009     });
1010
1011     let counts = Map{};
1012     for word in words {
1013         try { counts[word] = counts[word] + 1; }
1014         catch { counts[word] = 1; }
1015     }
1016
1017     return counts;
1018 };
1019
1020 let text = ``
1021 Peter Piper picked a peck of pickled peppers,
1022 A peck of pickled peppers Peter Piper picked;
1023 If Peter Piper picked a peck of pickled peppers,
1024 Where's the peck of pickled peppers Peter Piper picked?
1025 ``;
1026 println($"word count of the input text: {word_count(text)}");

```

GRAMMAR

```
1 <program> ::= <statement>*
2
3 <statement> ::= <statement-let>
4               | <statement-if-elif-else>
5               | <statement-for>
6               | <statement-while>
7               | <statement-break>
8               | <statement-continue>
9               | <statement-try>
10              | <statement-error>
11              | <statement-return>
12              | <statement-assign>
13              | <statement-expression>
14
15 <statement-let> ::= "let" <identifier> "=" <expression> ";"
16
17 <statement-if-elif-else> ::= "if" <expression> <block> ("elif" <expression> <block>)*
18   ↳ ("else" <block>)?
19
20 <statement-for> ::= "for" <identifier> "&"? ("," <identifier> "&")? "in" <expression>
21   ↳ <block>
22
23 <statement-while> ::= "while" <expression> <block>
24
25 <statement-break> ::= "break" ";"
26
27 <statement-continue> ::= "continue" ";"
28
29 <statement-try> ::= "try" <block> "catch" <identifier>? <block>
30
31 <statement-error> ::= "error" <expression> ";"
32
33 <statement-return> ::= "return" <expression>? ";"
34
35 <statement-assign> ::= <expression> "=" <expression> ";"
36
37 <statement-expression> ::= <expression> ";"
38
39 <block> ::= "{" <statement>* "}"
40
41 <expression> ::= <null>
42               | <boolean>
43               | <number>
44               | <string>
45               | <regex>
46               | <vector>
47               | <map>
48               | <set>
49               | <function>
50               | <template>
51               | <identifier>
52               | "freeze" <expression>
53               | "type" ("extends" <expression>)? <expression>
54               | "new" <expression> <expression>
55               | "(" <expression> ")"
56               | "+" <expression>
57               | "-" <expression>
58               | "not" <expression>
59               | <expression> "and" <expression>
```

```

58         | <expression> "or" <expression>
59         | <expression> "==" <expression>
60         | <expression> "!=" <expression>
61         | <expression> "<=" <expression>
62         | <expression> ">=" <expression>
63         | <expression> "<" <expression>
64         | <expression> ">" <expression>
65         | <expression> "=~" <expression>
66         | <expression> "!~" <expression>
67         | <expression> "+" <expression>
68         | <expression> "-" <expression>
69         | <expression> "*" <expression>
70         | <expression> "/" <expression>
71         | <expression> "%" <expression>
72         | <expression> "(" (<expression> ("," <expression>)*)? (","? ")"
73         | <expression> "[" <expression> "]"
74         | <expression> "." <identifier>
75         | <expression> "::" <identifier>
76         | <expression> "&"
77         | <expression> ".*"
78         | <regex-group>
79
80 <template-element> ::= "{"
81                     | "}"
82                     | "{" <expression> "}"
83                     | <character>
84
85 <template> ::= "$" "'" <template-element>* "'"
86             | "$" "\"" <template-element>* "\""
87             | "$" "`" <template-element>* "`"
88
89 <identifier> ::= /[a-zA-Z_]\w*/
90
91 <function> ::= "function" "&"? "(" (<identifier> ("," <identifier>)*)? ")" <block>
92
93 <set> ::= "Set" "{" "}"
94         | "Set"? "{" (<expression> ("," <expression>)* (","?))? "}"
95
96 <map-element> = <expression> ":" <expression>
97               | "." <identifier> "=" <expression>
98
99 <map> ::= "Map" "{" "}"
100         | "Map"? "{" (<map-element> ("," <map-element>)* (","?))? "}"
101
102 <vector> ::= "[" (<expression> ("," <expression>)* (","?))? "]"
103
104 <regex> ::= 'r' ? escaped re2 regular expression ? "'"
105           | "r" ? literal re2 regular expression ? "\""
106
107 <regex-group> ::= "$" /\d+/
108
109 <string> ::= "'" <character>* "'"
110           | "\"" <printable>* "\""
111           | "`" <printable>* "`"
112
113 <character> ::= "\t" # tab
114             | "\n" # newline
115             | "\"" # double quote
116             | "\\" # backslash
117             | /\[x[0-9a-fA-F][0-9a-fA-F]/ # hex byte, e.g. \xFF -> 0xFF
118             | (<printable> - ("'" | "\""))
119
120 <printable> ::= ? printable Unicode character ?
121
122 <line-feed> ::= ? Unicode character U+000A ?
123
124 <number> ::= /\d[0-9a-fA-F]+/

```

```
125         | /\d+(\.\d+)?/
126
127 <boolean> ::= "true"
128           | "false"
129
130 <null> ::= "null"
```


LICENSE

```
1 Copyright (c) 2025 ashn <me@ashn.dev>
2
3 Permission to use, copy, modify, and/or distribute this software for any
4 purpose with or without fee is hereby granted.
5
6 THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
7 REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
8 AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
9 INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
10 LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
11 OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
12 PERFORMANCE OF THIS SOFTWARE.
```


Makefile

```
1 .POSIX:
2 .PHONY: \
3     default all-go all-py \
4     build build-go \
5     wasm wasm-go \
6     install \
7     check check-go check-py \
8     lint-py \
9     format format-go format-py \
10    clean
11
12 MELLIFERA_HOME = $$HOME/.mellifera
13
14 default: build-go
15
16 all-go: build-go wasm-go format-go check-go
17
18 all-py: format-py lint-py check-py
19
20 bin/mf: build-go
21
22 build: build-go
23
24 build-go:
25     go build -o=bin/ cmd/mf/mf.go
26
27 wasm: wasm-go
28
29 wasm-go:
30     GOARCH=wasm GOOS=js go build -o mellifera.wasm cmd/wasm/wasm.go
31     cp "$$(go env GOROOT)/lib/wasm/wasm_exec.js" .
32
33 install: bin/mf
34     mkdir -p "$(MELLIFERA_HOME)/bin"
35     mkdir -p "$(MELLIFERA_HOME)/lib"
36     cp -r bin "$(MELLIFERA_HOME)"
37     cp -r lib "$(MELLIFERA_HOME)"
38     cp mf.py "$(MELLIFERA_HOME)"
39
40 check: check-go
41
42 check-go: bin/mf
43     MELLIFERA_HOME="$(realpath .)" sh bin/mf-test
44
45 check-py:
46     MELLIFERA_HOME="$(realpath .)" sh bin/mf-test --py
47
48 # Flake8 Ignored Errors:
49 #   E203 - Conflicts with Black.
50 #   E221 - Disabled for manual vertically-aligned code.
51 #   E241 - Disabled for manual vertically-aligned code.
52 #   E501 - Conflicts with Black.
53 #   W503 - Conflicts with Black.
54 lint-py:
55     python3 -m mypy --check-untyped-defs mf.py
56     python3 -m flake8 --ignore=E203,E221,E241,E501,W503 mf.py
57
58 format: format-go
59
```

Makefile

```
60 format-go:
61     go fmt
62     go fmt cmd/mf/*.go
63     go fmt cmd/wasm/*.go
64
65 format-py:
66     python3 -m black mf.py
67
68 clean:
69     rm -rf __pycache__
70     rm -rf .mypy_cache
71     rm -f bin/mf
72     rm -f mellifera.wasm
73     rm -f wasm_exec.js
```

go.mod

```
1 module ashn.dev/mellifera
2
3 go 1.26
```

go.mod

mellifera.go

```

1 package mellifera
2
3 import (
4     "encoding/json"
5     "errors"
6     "fmt"
7     "io"
8     "iter"
9     "math"
10    "math/rand/v2"
11    "os"
12    "reflect"
13    "regexp"
14    "slices"
15    "strconv"
16    "strings"
17    "sync"
18    "unicode"
19    "unicode/utf8"
20 )
21
22 const (
23     MAX_INDEX_INTEGER = 2147483647 // 2**31 - 1
24     MAX_SAFE_INTEGER  = +9007199254740991 // +(2**53 - 1)
25     MIN_SAFE_INTEGER  = -9007199254740991 // -(2**53 - 1)
26 )
27
28 func escape(s string) string {
29     result := ""
30     index := 0
31
32     for {
33         r, size := utf8.DecodeRuneInString(s[index:])
34
35         if r == utf8.RuneError && size == 0 {
36             break
37         }
38         if r == utf8.RuneError && size == 1 {
39             result += fmt.Sprintf("\\x%02x", s[index])
40             index += 1
41             continue
42         }
43
44         index += size
45         if r == '\\t' {
46             result += "\\t"
47             continue
48         }
49         if r == '\\n' {
50             result += "\\n"
51             continue
52         }
53         if r == '"' {
54             result += "\\\""
55             continue
56         }
57         if r == '\\\\' {
58             result += "\\\\"
59             continue
60         }
61     }
62
63     return result
64 }

```

```

60     }
61     result += string(r)
62 }
63
64 return result
65 }
66
67 func quote(s string) string {
68     if strings.Contains(s, "\"") {
69         return fmt.Sprintf(`"%s"`, s)
70     }
71     return fmt.Sprintf("`%s`", s)
72 }
73
74 // FNV-1a
75 func fnv1a(s string) uint64 {
76     var hash uint64 = 14695981039346656037 // FNV_offset_basis
77     for i := 0; i < len(s); i += 1 {
78         hash ^= uint64(s[i])
79         hash *= 1099511628211 // FNV_prime
80     }
81     return hash
82 }
83
84 // MurmurHash3 64-bit finalization mix
85 func fmix64(k uint64) uint64 {
86     k ^= k >> 33
87     k *= 0xff51afd7ed558ccd
88     k ^= k >> 33
89     k *= 0xc4ceb9fe1a85ec53
90     k ^= k >> 33
91     return k
92 }
93
94 type Value interface {
95     Typename() string
96     String() string
97     Meta(ctx *Context) *Map
98     Take() Value
99     Copy() Value
100    CopyOnWrite()
101    Freeze() Value
102    IsImmutable() bool
103    Hash() uint64
104    Equal(Value) bool
105    CombEncode(e *CombEncoder) error
106 }
107
108 func MetaFunction(ctx *Context, value Value, name Value) (Value, bool) {
109     meta := value.Meta(ctx)
110     if meta == nil {
111         return nil, false
112     }
113     function, ok := meta.Lookup(name)
114     if !ok {
115         return nil, false
116     }
117     if _, ok := function.(*Function); ok {
118         return function, true
119     }
120     if _, ok := function.(*Builtin); ok {
121         return function, true
122     }
123     return nil, false
124 }
125
126 func ValueAsSafeInteger(value Value) (int64, error) {

```

```

127     number, ok := value.(*Number)
128     if !ok {
129         return 0, fmt.Errorf("cannot convert %s into an integer", value.TypeName())
130     }
131     if math.IsInf(number.data, 0) || math.IsNaN(number.data) {
132         return 0, fmt.Errorf("cannot convert %v into an integer", value)
133     }
134     truncated := math.Trunc(number.data)
135     if truncated != number.data {
136         return 0, fmt.Errorf("cannot convert %v into an integer without truncation",
↪ value)
137     }
138     if truncated < MIN_SAFE_INTEGER || truncated > MAX_SAFE_INTEGER {
139         return 0, fmt.Errorf("integer %v is outside the safe integer range", value)
140     }
141     return int64(truncated), nil
142 }
143
144 func ValueAsIndex(value Value) (int, error) {
145     number, ok := value.(*Number)
146     if !ok {
147         return 0, fmt.Errorf("cannot convert %s into an integer", value.TypeName())
148     }
149     if math.IsInf(number.data, 0) || math.IsNaN(number.data) {
150         return 0, fmt.Errorf("cannot convert %v into an integer", value)
151     }
152     truncated := math.Trunc(number.data)
153     if truncated != number.data {
154         return 0, fmt.Errorf("cannot convert %v into an integer without truncation",
↪ value)
155     }
156     if truncated < 0 || truncated > MAX_INDEX_INTEGER {
157         return 0, fmt.Errorf("integer %v is outside the indexable integer range", value)
158     }
159     return int(truncated), nil
160 }
161
162 // Internal optimization allowing for a freshly-evaluated rvalue to be directly
163 // moved into a store (a la std::move) without requiring an extra Copy()
164 // operation:
165 //
166 // let x = ["foo", "bar", "baz"]; # rvalue stored in the environment
167 // v[0] = {"abc": 123};           # rvalue stored in a container
168 // m["foo"] = bar;                # lvalue copied into a container
169 //
170 // If the provided argument is a free value, then that value is returned
171 // verbatim ready to have its ownership transferred via a Take() operation.
172 //
173 // If the provided argument is an owned value then it is copied, ensuring that
174 // the invariant of at most one owner per value is upheld.
175 //
176 // Callers must guarantee that the provided argument is relinquished as part of
177 // the function invocation. Since it is unknown whether the value returned is
178 // the same as the input argument or a copy, invoking moveOrCopy() MUST be
179 // treated strictly as if the value was moved and not copied, otherwise there
180 // is the possibility that the original value and the moveOrCopy()-ed result
181 // could alias.
182 func moveOrCopy(value Value) Value {
183     switch v := value.(type) {
184     case *Vector:
185         if v.state == containerStateFree && (v.data == nil || v.data.refs == 0) {
186             return value // move
187         }
188     case *Map:
189         if v.state == containerStateFree && (v.data == nil || v.data.refs == 0) {
190             return value // move
191         }

```

```

192     case *Set:
193         if v.state == containerStateFree && (v.data == nil || v.data.refs == 0) {
194             return value // move
195         }
196     }
197
198     // immutable values => Copy() returning immutable identity
199     // mutable values => Copy() returning mutable value with unique data
200     return value.Copy()
201 }
202
203 // Decrement the shared use count of the underlying shared data held by this
204 // value. This operation is only valid when it is known that the marked
205 // value is *guaranteed* to be currently unreachable.
206 func unmarkUsed(value Value) {
207     switch v := value.(type) {
208     case *Vector:
209         if v.data != nil && !v.IsImmutable() {
210             v.data.uses -= 1
211         }
212     case *Map:
213         if v.data != nil && !v.IsImmutable() {
214             v.data.uses -= 1
215         }
216     case *Set:
217         if v.data != nil && !v.IsImmutable() {
218             v.data.uses -= 1
219         }
220     }
221 }
222
223 // Increment the reference count of the underlying shared data held by this
224 // value to signal that there is an access path to that data through a
225 // reference.
226 func markReferenced(value Value) {
227     switch v := value.(type) {
228     case *Vector:
229         if v.data == nil {
230             v.data = &vectorData{uses: 1}
231         }
232         v.data.refs += 1
233     case *Map:
234         if v.data == nil {
235             v.data = &mapData{uses: 1}
236         }
237         v.data.refs += 1
238     case *Set:
239         if v.data == nil {
240             v.data = &setData{uses: 1}
241         }
242         v.data.refs += 1
243     }
244 }
245
246 // Decrement the reference count of the underlying shared data held by this
247 // value. This operation is only valid when it is known that the referenced
248 // value was previously marked with markReferenced(), and that all references
249 // to the value are *guaranteed* to be currently unreachable.
250 func unmarkReferenced(value Value) {
251     switch v := value.(type) {
252     case *Vector:
253         if v.data != nil {
254             v.data.refs -= 1
255         }
256     case *Map:
257         if v.data != nil {
258             v.data.refs -= 1

```

```

259     }
260     case *Set:
261         if v.data != nil {
262             v.data.refs -= 1
263         }
264     }
265 }
266
267 // Reports whether a callable's implicit `self` argument is passed by reference
268 // in function call with a dot access expression as the left hand side of that
269 // expression.
270 //
271 // function(self) { ... } # pass by value
272 // function.&(self) { ... } # pass by reference
273 func callableSelfIsPassedByReference(function Value) bool {
274     switch f := function.(type) {
275     case *Function:
276         return f.ast.SelfByReference
277     case *Builtin:
278         return f.byRef
279     default:
280         return false
281     }
282 }
283
284 // Invokes the provided metafunction, passing an implicit copy or reference
285 // `self` to the metafunction. Effectively a utility function for the common
286 // pattern of calling value.into_string() or a similarly-flavored callable.
287 func (ctx *Context) callMetaWithSelf(location *SourceLocation, metaFunction Value, self
    ↪ Value) (Value, error) {
288     if !callableSelfIsPassedByReference(metaFunction) {
289         return call(ctx, location, metaFunction, []Value{self.Copy()})
290     }
291     result, err := call(ctx, location, metaFunction, []Value{ctx.newReference(self)})
292     unmarkReferenced(self)
293     return result, err
294 }
295
296 type CombEncoder struct {
297     w          io.Writer
298     err        error // Internal sticky error.
299     indentLevel int // Number of times the indent text is written before the main text
    ↪ per line.
300
301     // Indent before the first element on a new line.
302     // Optional: nil implies single-line default formatting.
303     // [
304     //     "foo",
305     //     "bar",
306     //     "baz"
307     // ]
308     // ^^^^
309     // \\\\_ indent of " "
310     Indent *string
311     // Space separator between elements.
312     //
313     // ["foo", "bar", "baz"]
314     //      ^      ^
315     // \_____\ separator of " "
316     // {"foo": 123, "bar": 456}
317     //      ^      ^      ^
318     // \_____\ separator of " "
319     Separator string
320 }
321
322 func NewCombEncoder(w io.Writer) *CombEncoder {
323     return &CombEncoder{

```

```

324         w:          w,
325         err:         nil,
326         indentLevel: 0,
327
328         Indent:      nil,
329         Separator: "",
330     }
331 }
332
333 func (e *CombEncoder) writeString(s string) error {
334     if e.err != nil {
335         return e.err
336     }
337
338     _, e.err = e.w.Write([]byte(s))
339
340     return e.err
341 }
342
343 func (e *CombEncoder) writeIndent(s string) error {
344     if e.Indent != nil {
345         for range e.indentLevel {
346             e.writeString(*e.Indent)
347         }
348     }
349
350     e.writeString(s)
351
352     return e.err
353 }
354
355 func (e *CombEncoder) writeEndOfLine() error {
356     if e.Indent != nil {
357         e.writeString("\n")
358     } else {
359         e.writeString(e.Separator)
360     }
361
362     return e.err
363 }
364
365 type Context struct {
366     Stdin io.Reader
367     Stdout io.Writer
368     Stderr io.Writer
369
370     // Builtin Type Metamaps
371     functionMeta *Map
372     booleanMeta  *Map
373     numberMeta   *Map
374     stringMeta   *Map
375     regexpMeta   *Map
376     vectorMeta   *Map
377     mapMeta      *Map
378     setMeta      *Map
379     referenceMeta *Map
380
381     // Null Singleton
382     Null *Null
383
384     // Boolean Singletons
385     True  *Boolean
386     False *Boolean
387
388     // Base Environment
389     BaseEnvironment *Environment
390

```

```

391 // Context-Specific Random Number Generation State
392 rng *rand.Rand
393
394 // Miscellaneous State and Definitions
395 reIntegerDec      *regexp.Regexp
396 reIntegerDecFullmatch *regexp.Regexp
397 reNumberDec       *regexp.Regexp
398 reNumberDecFullmatch *regexp.Regexp
399 reNumberHex       *regexp.Regexp
400 reNumberHexFullmatch *regexp.Regexp
401 importFileCache   map[string]Value
402 identifierCache    map[string]*String
403 constStringIntoString *String
404 constStringNext     *String
405 callDepth          int
406 }
407
408 func NewContext() *Context {
409     ctx := &Context{}
410
411     ctx.Stdin = os.Stdin
412     ctx.Stdout = os.Stdout
413     ctx.Stderr = os.Stderr
414
415     ctx.functionMeta = ctx.NewMetaMapOrPanic("function", nil)
416     ctx.booleanMeta = ctx.NewMetaMapOrPanic("boolean", []MapPair{
417         {ctx.NewString("init"), BuiltinBooleanInit(ctx)},
418     })
419     ctx.numberMeta = ctx.NewMetaMapOrPanic("number", []MapPair{
420         {ctx.NewString("MAX_SAFE_INTEGER"), ctx.NewNumber(MAX_SAFE_INTEGER)},
421         {ctx.NewString("MIN_SAFE_INTEGER"), ctx.NewNumber(MIN_SAFE_INTEGER)},
422         {ctx.NewString("init"), BuiltinNumberInit(ctx)},
423         {ctx.NewString("is_nan"), BuiltinNumberIsNan(ctx)},
424         {ctx.NewString("is_inf"), BuiltinNumberIsInf(ctx)},
425         {ctx.NewString("is_finite"), BuiltinNumberIsFinite(ctx)},
426         {ctx.NewString("is_integer"), BuiltinNumberIsInteger(ctx)},
427         {ctx.NewString("format"), BuiltinNumberFormat(ctx)},
428         {ctx.NewString("fixed"), BuiltinNumberFixed(ctx)},
429         {ctx.NewString("trunc"), BuiltinNumberTrunc(ctx)},
430         {ctx.NewString("round"), BuiltinNumberRound(ctx)},
431         {ctx.NewString("floor"), BuiltinNumberFloor(ctx)},
432         {ctx.NewString("ceil"), BuiltinNumberCeil(ctx)},
433     })
434     ctx.stringMeta = ctx.NewMetaMapOrPanic("string", []MapPair{
435         {ctx.NewString("init"), BuiltinStringInit(ctx)},
436         {ctx.NewString("bytes"), BuiltinStringBytes(ctx)},
437         {ctx.NewString("runes"), BuiltinStringRunes(ctx)},
438         {ctx.NewString("count"), BuiltinStringCount(ctx)},
439         {ctx.NewString("is_empty"), BuiltinStringIsEmpty(ctx)},
440         {ctx.NewString("contains"), BuiltinStringContains(ctx)},
441         {ctx.NewString("starts_with"), BuiltinStringStartsWith(ctx)},
442         {ctx.NewString("ends_with"), BuiltinStringEndsWith(ctx)},
443         {ctx.NewString("trim"), BuiltinStringTrim(ctx)},
444         {ctx.NewString("find"), BuiltinStringFind(ctx)},
445         {ctx.NewString("rfind"), BuiltinStringRfind(ctx)},
446         {ctx.NewString("slice"), BuiltinStringSlice(ctx)},
447         {ctx.NewString("split"), BuiltinStringSplit(ctx)},
448         {ctx.NewString("join"), BuiltinStringJoin(ctx)},
449         {ctx.NewString("cut"), BuiltinStringCut(ctx)},
450         {ctx.NewString("replace"), BuiltinStringReplace(ctx)},
451         {ctx.NewString("to_title"), BuiltinStringToTitle(ctx)},
452         {ctx.NewString("to_upper"), BuiltinStringToUpper(ctx)},
453         {ctx.NewString("to_lower"), BuiltinStringToLower(ctx)},
454     })
455     ctx.regexpMeta = ctx.NewMetaMapOrPanic("regexp", []MapPair{
456         {ctx.NewString("init"), BuiltinRegexpInit(ctx)},
457         {ctx.NewString("split"), BuiltinRegexpSplit(ctx)},

```

```

458         {ctx.NewString("replace"), BuiltinRegexpReplace(ctx)},
459     })
460     ctx.vectorMeta = ctx.NewMetaMapOrPanic("vector", []MapPair{
461         {ctx.NewString("init"), BuiltinVectorInit(ctx)},
462         {ctx.NewString("count"), BuiltinVectorCount(ctx)},
463         {ctx.NewString("is_empty"), BuiltinVectorIsEmpty(ctx)},
464         {ctx.NewString("contains"), BuiltinVectorContains(ctx)},
465         {ctx.NewString("insert"), BuiltinVectorInsert(ctx)},
466         {ctx.NewString("remove"), BuiltinVectorRemove(ctx)},
467         {ctx.NewString("push"), BuiltinVectorPush(ctx)},
468         {ctx.NewString("pop"), BuiltinVectorPop(ctx)},
469         {ctx.NewString("any"), BuiltinVectorAny(ctx)},
470         {ctx.NewString("all"), BuiltinVectorAll(ctx)},
471         {ctx.NewString("map"), BuiltinVectorMap(ctx)},
472         {ctx.NewString("filter"), BuiltinVectorFilter(ctx)},
473         {ctx.NewString("reduce"), BuiltinVectorReduce(ctx)},
474         {ctx.NewString("find"), BuiltinVectorFind(ctx)},
475         {ctx.NewString("rfind"), BuiltinVectorRfind(ctx)},
476         {ctx.NewString("slice"), BuiltinVectorSlice(ctx)},
477         {ctx.NewString("reversed"), BuiltinVectorReversed(ctx)},
478         {ctx.NewString("sorted"), ctx.NewNull()}, // deferred instantiation
479         {ctx.NewString("sorted_by"), ctx.NewNull()}, // deferred instantiation
480         {ctx.NewString("into_iterator"), ctx.NewNull()}, // deferred instantiation
481     })
482     ctx.mapMeta = ctx.NewMetaMapOrPanic("map", []MapPair{
483         {ctx.NewString("init"), BuiltinMapInit(ctx)},
484         {ctx.NewString("count"), BuiltinMapCount(ctx)},
485         {ctx.NewString("is_empty"), BuiltinMapIsEmpty(ctx)},
486         {ctx.NewString("contains"), BuiltinMapContains(ctx)},
487         {ctx.NewString("insert"), BuiltinMapInsert(ctx)},
488         {ctx.NewString("remove"), BuiltinMapRemove(ctx)},
489         {ctx.NewString("get"), BuiltinMapGet(ctx)},
490         {ctx.NewString("keys"), BuiltinMapKeys(ctx)},
491         {ctx.NewString("values"), BuiltinMapValues(ctx)},
492         {ctx.NewString("pairs"), BuiltinMapPairs(ctx)},
493         {ctx.NewString("union"), ctx.NewNull()}, // deferred instantiation
494         {ctx.NewString("into_iterator"), ctx.NewNull()}, // deferred instantiation
495     })
496     ctx.setMeta = ctx.NewMetaMapOrPanic("set", []MapPair{
497         {ctx.NewString("init"), BuiltinSetInit(ctx)},
498         {ctx.NewString("count"), BuiltinSetCount(ctx)},
499         {ctx.NewString("is_empty"), BuiltinSetIsEmpty(ctx)},
500         {ctx.NewString("contains"), BuiltinSetContains(ctx)},
501         {ctx.NewString("insert"), BuiltinSetInsert(ctx)},
502         {ctx.NewString("remove"), BuiltinSetRemove(ctx)},
503         {ctx.NewString("union"), ctx.NewNull()}, // deferred instantiation
504         {ctx.NewString("intersection"), ctx.NewNull()}, // deferred instantiation
505         {ctx.NewString("difference"), ctx.NewNull()}, // deferred instantiation
506         {ctx.NewString("into_iterator"), ctx.NewNull()}, // deferred instantiation
507     })
508     ctx.referenceMeta = ctx.NewMetaMapOrPanic("reference", nil)
509
510     ctx.Null = &Null{}
511
512     ctx.True = &Boolean{true}
513     ctx.False = &Boolean{false}
514
515     ctx.BaseEnvironment = NewEnvironment(nil)
516
517     seed := rand.Uint64() // random initial rng seed
518     ctx.rng = rand.New(rand.NewPCG(seed, seed))
519
520     ctx.reIntegerDec = regexp.MustCompile(`^\d+`)
521     ctx.reIntegerDecFullmatch = regexp.MustCompile(`^\d+$`)
522     ctx.reNumberDec = regexp.MustCompile(`^\d+(\.\d+)?`)
523     ctx.reNumberDecFullmatch = regexp.MustCompile(`^\d+(\.\d+)?$`)
524     ctx.reNumberHex = regexp.MustCompile(`^0x[0-9a-fA-F]+`)

```

```

525 ctx.reNumberHexFullmatch = regexp.MustCompile(`^0x[0-9a-fA-F]+$`)
526 ctx.importFileCache = map[string]Value{}
527 ctx.identifierCache = map[string]*String{}
528 ctx.constStringIntoString = ctx.NewString("into_string")
529 ctx.constStringNext = ctx.NewString("next")
530 ctx.callDepth = 0
531
532 ctx.BaseEnvironment.Let("boolean", ctx.booleanMeta)
533 ctx.BaseEnvironment.Let("number", ctx.numberMeta)
534 ctx.BaseEnvironment.Let("string", ctx.stringMeta)
535 ctx.BaseEnvironment.Let("regexp", ctx.regexpMeta)
536 ctx.BaseEnvironment.Let("vector", ctx.vectorMeta)
537 ctx.BaseEnvironment.Let("map", ctx.mapMeta)
538 ctx.BaseEnvironment.Let("set", ctx.setMeta)
539 ctx.BaseEnvironment.Let("reference", ctx.referenceMeta)
540 ctx.BaseEnvironment.Let("iterator", ctx.NewValueFromSourceOrPanic("",
↵ _ITERATOR_SOURCE))
541 ctx.BaseEnvironment.Let("NaN", ctx.NewNumber(math.NaN()))
542 ctx.BaseEnvironment.Let("Inf", ctx.NewNumber(math.Inf(+1)))
543 ctx.BaseEnvironment.Let("assert", BuiltinAssert(ctx))
544 ctx.BaseEnvironment.Let("typeof", BuiltinTypeof(ctx))
545 ctx.BaseEnvironment.Let("typename", BuiltinTypename(ctx))
546 ctx.BaseEnvironment.Let("repr", BuiltinRepr(ctx))
547 ctx.BaseEnvironment.Let("dump", BuiltinDump(ctx))
548 ctx.BaseEnvironment.Let("dumpln", BuiltinDumpln(ctx))
549 ctx.BaseEnvironment.Let("print", BuiltinPrint(ctx))
550 ctx.BaseEnvironment.Let("println", BuiltinPrintln(ctx))
551 ctx.BaseEnvironment.Let("eprint", BuiltinEprint(ctx))
552 ctx.BaseEnvironment.Let("eprintln", BuiltinEprintln(ctx))
553 ctx.BaseEnvironment.Let("range", ctx.NewNull()) // deferred instantiation
554 ctx.BaseEnvironment.Let("min", BuiltinMin(ctx))
555 ctx.BaseEnvironment.Let("max", BuiltinMax(ctx))
556 ctx.BaseEnvironment.Let("comb", ctx.NewMapOrPanic([]MapPair{
557     {ctx.NewString("decode"), BuiltinCombDecode(ctx)},
558     {ctx.NewString("encode"), BuiltinCombEncode(ctx)},
559     {ctx.NewString("encode_ex"), BuiltinCombEncodeEx(ctx)},
560 },).Freeze())
561 ctx.BaseEnvironment.Let("html", ctx.NewMapOrPanic([]MapPair{
562     {ctx.NewString("escape"), BuiltinHtmlEscape(ctx)},
563 },).Freeze())
564 ctx.BaseEnvironment.Let("json", ctx.NewMapOrPanic([]MapPair{
565     {ctx.NewString("decode"), BuiltinJsonDecode(ctx)},
566     {ctx.NewString("encode"), BuiltinJsonEncode(ctx)},
567 },).Freeze())
568 ctx.BaseEnvironment.Let("math", ctx.NewMapOrPanic([]MapPair{
569     {ctx.NewString("e"), ctx.NewNumber(math.E)},
570     {ctx.NewString("pi"), ctx.NewNumber(math.Pi)},
571     {ctx.NewString("is_nan"), BuiltinMathIsNaN(ctx)},
572     {ctx.NewString("is_inf"), BuiltinMathIsInf(ctx)},
573     {ctx.NewString("is_finite"), BuiltinMathIsFinite(ctx)},
574     {ctx.NewString("is_integer"), BuiltinMathIsInteger(ctx)},
575     {ctx.NewString("sign"), BuiltinMathSign(ctx)},
576     {ctx.NewString("copy_sign"), BuiltinMathCopySign(ctx)},
577     {ctx.NewString("trunc"), BuiltinMathTrunc(ctx)},
578     {ctx.NewString("round"), BuiltinMathRound(ctx)},
579     {ctx.NewString("floor"), BuiltinMathFloor(ctx)},
580     {ctx.NewString("ceil"), BuiltinMathCeil(ctx)},
581     {ctx.NewString("abs"), BuiltinMathAbs(ctx)},
582     {ctx.NewString("mod"), BuiltinMathMod(ctx)},
583     {ctx.NewString("rem"), BuiltinMathRem(ctx)},
584     {ctx.NewString("exp"), BuiltinMathExp(ctx)},
585     {ctx.NewString("exp2"), BuiltinMathExp2(ctx)},
586     {ctx.NewString("exp10"), BuiltinMathExp10(ctx)},
587     {ctx.NewString("log"), BuiltinMathLog(ctx)},
588     {ctx.NewString("log2"), BuiltinMathLog2(ctx)},
589     {ctx.NewString("log10"), BuiltinMathLog10(ctx)},
590     {ctx.NewString("pow"), BuiltinMathPow(ctx)},

```

```

591         {ctx.NewString("sqrt"), BuiltinMathSqrt(ctx)},
592         {ctx.NewString("cbrt"), BuiltinMathCbrt(ctx)},
593         {ctx.NewString("clamp"), BuiltinMathClamp(ctx)},
594         {ctx.NewString("sin"), BuiltinMathSin(ctx)},
595         {ctx.NewString("cos"), BuiltinMathCos(ctx)},
596         {ctx.NewString("tan"), BuiltinMathTan(ctx)},
597         {ctx.NewString("asin"), BuiltinMathAsin(ctx)},
598         {ctx.NewString("acos"), BuiltinMathAcos(ctx)},
599         {ctx.NewString("atan"), BuiltinMathAtan(ctx)},
600         {ctx.NewString("atan2"), BuiltinMathAtan2(ctx)},
601         {ctx.NewString("sinh"), BuiltinMathSinh(ctx)},
602         {ctx.NewString("cosh"), BuiltinMathCosh(ctx)},
603         {ctx.NewString("tanh"), BuiltinMathTanh(ctx)},
604         {ctx.NewString("asinh"), BuiltinMathAsinh(ctx)},
605         {ctx.NewString("acosh"), BuiltinMathAcosh(ctx)},
606         {ctx.NewString("atanh"), BuiltinMathAtanh(ctx)},
607         {ctx.NewString("sum"), BuiltinMathSum(ctx)},
608         {ctx.NewString("product"), BuiltinMathProduct(ctx)},
609     }).Freeze()
610     ctx.BaseEnvironment.Let("module", ctx.NewMapOrPanic([]MapPair{
611         {ctx.NewString("path"), ctx.NewNull()},
612         {ctx.NewString("file"), ctx.NewNull()},
613         {ctx.NewString("directory"), ctx.NewNull()}},
614     }).Freeze()
615     ctx.BaseEnvironment.Let("random", ctx.NewMapOrPanic([]MapPair{
616         {ctx.NewString("seed"), BuiltinRandomSeed(ctx)},
617         {ctx.NewString("integer"), BuiltinRandomInteger(ctx)},
618     }).Freeze()
619     ctx.BaseEnvironment.Let("re", ctx.NewMapOrPanic([]MapPair{
620         {ctx.NewString("split"), BuiltinReSplit(ctx)},
621         {ctx.NewString("replace"), BuiltinReReplace(ctx)},
622     }).Freeze()
623     ctx.BaseEnvironment.Let("ty", ctx.NewMapOrPanic([]MapPair{
624         {ctx.NewString("is"), BuiltinTyIs(ctx)},
625         {ctx.NewString("is_null"), BuiltinTyIsNull(ctx)},
626         {ctx.NewString("is_boolean"), BuiltinTyIsBoolean(ctx)},
627         {ctx.NewString("is_number"), BuiltinTyIsNumber(ctx)},
628         {ctx.NewString("is_string"), BuiltinTyIsString(ctx)},
629         {ctx.NewString("is_regexp"), BuiltinTyIsRegexp(ctx)},
630         {ctx.NewString("is_vector"), BuiltinTyIsVector(ctx)},
631         {ctx.NewString("is_map"), BuiltinTyIsMap(ctx)},
632         {ctx.NewString("is_set"), BuiltinTyIsSet(ctx)},
633         {ctx.NewString("is_reference"), BuiltinTyIsReference(ctx)},
634         {ctx.NewString("is_function"), BuiltinTyIsFunction(ctx)},
635     }).Freeze()
636     ctx.BaseEnvironment.Let("utf8", ctx.NewMapOrPanic([]MapPair{
637         {ctx.NewString("is_valid"), BuiltinUTF8IsValid(ctx)},
638         {ctx.NewString("code_point"), BuiltinUTF8CodePoint(ctx)},
639         {ctx.NewString("code_points"), BuiltinUTF8CodePoints(ctx)},
640         {ctx.NewString("string"), BuiltinUTF8String(ctx)},
641     }).Freeze()
642
643     // Initialize builtins from source with deferred instantiation.
644     ctx.vectorMeta.data.hmap.insert(ctx.NewString("sorted"), BuiltinVectorSorted(ctx))
645     ctx.vectorMeta.data.hmap.insert(ctx.NewString("sorted_by"),
646 ↪ BuiltinVectorSortedBy(ctx))
647     ctx.vectorMeta.data.hmap.insert(ctx.NewString("into_iterator"),
648 ↪ BuiltinVectorIntoIterator(ctx))
649     ctx.mapMeta.data.hmap.insert(ctx.NewString("union"), BuiltinMapUnion(ctx))
650     ctx.mapMeta.data.hmap.insert(ctx.NewString("into_iterator"),
651 ↪ BuiltinMapIntoIterator(ctx))
652     ctx.setMeta.data.hmap.insert(ctx.NewString("union"), BuiltinSetUnion(ctx))
653     ctx.setMeta.data.hmap.insert(ctx.NewString("intersection"),
654 ↪ BuiltinSetIntersection(ctx))
655     ctx.setMeta.data.hmap.insert(ctx.NewString("difference"), BuiltinSetDifference(ctx))
656     ctx.setMeta.data.hmap.insert(ctx.NewString("into_iterator"),
657 ↪ BuiltinSetIntoIterator(ctx))

```

```

653     _ = ctx.BaseEnvironment.Set("range", BuiltinRange(ctx))
654
655     return ctx
656 }
657
658 func (ctx *Context) NewNull() *Null {
659     return ctx.Null
660 }
661
662 func (ctx *Context) NewBoolean(data bool) *Boolean {
663     if data {
664         return ctx.True
665     }
666     return ctx.False
667 }
668
669 func (ctx *Context) NewNumber(data float64) *Number {
670     return &Number{data}
671 }
672
673 func (ctx *Context) NewString(data string) *String {
674     return &String{data}
675 }
676
677 func (ctx *Context) NewStringf(format string, args ...any) *String {
678     return &String{fmt.Sprintf(format, args...)}
679 }
680
681 func (ctx *Context) NewRegexp(text string) (*Regexp, error) {
682     data, err := regexp.Compile(text)
683     if err != nil {
684         return nil, fmt.Errorf("invalid regular expression \"%s\"", escape(text))
685     }
686     return &Regexp{data}, nil
687 }
688
689 func (ctx *Context) NewRegexpOrPanic(text string) *Regexp {
690     result, err := ctx.NewRegexp(text)
691     if err != nil {
692         panic(err.Error())
693     }
694     return result
695 }
696
697 func (ctx *Context) NewVector(elements []Value) (*Vector, error) {
698     result := &Vector{data: nil}
699     for _, element := range elements {
700         if err := result.Push(element); err != nil {
701             return nil, fmt.Errorf("invalid vector construction with element %v",
702 ↪ element)
703         }
704     }
705     return result, nil
706 }
707
708 func (ctx *Context) NewVectorOrPanic(elements []Value) *Vector {
709     result, err := ctx.NewVector(elements)
710     if err != nil {
711         panic(err.Error())
712     }
713     return result
714 }
715
716 func (ctx *Context) NewMap(elements []MapPair) (*Map, error) {
717     if elements == nil || len(elements) == 0 {
718         return &Map{
719             data: nil,

```

```

719         meta: ctx.mapMeta,
720         name: nil,
721     }, nil
722 }
723
724 result := &Map{
725     data: nil,
726     meta: ctx.mapMeta,
727     name: nil,
728 }
729 for _, element := range elements {
730     err := result.Insert(element.Key, element.Value)
731     if err != nil {
732         return nil, fmt.Errorf("invalid map construction with key %v and value %v",
↪ element.Key, element.Value)
733     }
734 }
735 return result, nil
736 }
737
738 func (ctx *Context) NewMapOrPanic(elements []MapPair) *Map {
739     result, err := ctx.NewMap(elements)
740     if err != nil {
741         panic(err.Error())
742     }
743     return result
744 }
745
746 func (ctx *Context) NewMapWithType(meta *Map, elements []MapPair) (*Map, error) {
747     if meta.name == nil {
748         return nil, errors.New("meta argument is not a metamap")
749     }
750
751     result, err := ctx.NewMap(elements)
752     if err != nil {
753         return nil, err
754     }
755     result.meta = meta.Copy().(*Map)
756
757     return result, nil
758 }
759
760 func (ctx *Context) NewMapWithTypeOrPanic(meta *Map, elements []MapPair) *Map {
761     result, err := ctx.NewMapWithType(meta, elements)
762     if err != nil {
763         panic(err.Error())
764     }
765     return result
766 }
767
768 func (ctx *Context) NewMetaMap(name string, elements []MapPair) (*Map, error) {
769     result, err := ctx.NewMap(elements)
770     if err != nil {
771         return nil, err
772     }
773     result.name = &name
774     return result.Freeze().(*Map), nil
775 }
776
777 func (ctx *Context) NewMetaMapOrPanic(name string, elements []MapPair) *Map {
778     result, err := ctx.NewMetaMap(name, elements)
779     if err != nil {
780         panic(err.Error())
781     }
782     return result
783 }
784

```

```

785 func (ctx *Context) NewSet(elements []Value) (*Set, error) {
786     if elements == nil || len(elements) == 0 {
787         return &Set{data: nil}, nil
788     }
789
790     result := &Set{data: nil}
791     for _, element := range elements {
792         err := result.Insert(element)
793         if err != nil {
794             return nil, fmt.Errorf("invalid set construction with element %v", element)
795         }
796     }
797     return result, nil
798 }
799
800 func (ctx *Context) NewSetOrPanic(elements []Value) *Set {
801     result, err := ctx.NewSet(elements)
802     if err != nil {
803         panic(err.Error())
804     }
805     return result
806 }
807
808 func (ctx *Context) newReference(value Value) *Reference {
809     value.CopyOnWrite()
810     markReferenced(value)
811     return &Reference{value}
812 }
813
814 func (ctx *Context) newFunction(ast *AstExpressionFunction, env *Environment) *Function {
815     return &Function{ast, env}
816 }
817
818 func (ctx *Context) NewBuiltin(name string, types []Type, impl func(*Context, []Value)
819     ↪ (Value, error)) *Builtin {
820     return &Builtin{name, types, impl, false}
821 }
822
823 // Takes an implicit `self` argument by reference. This `self` argument *must*
824 // not escape, as the Mellifera runtime treats the self reference as temporary.
825 func (ctx *Context) NewBuiltinWithSelfByReference(name string, types []Type, impl
826     ↪ func(*Context, []Value) (Value, error)) *Builtin {
827     return &Builtin{name, types, impl, true}
828 }
829
830 func (ctx *Context) NewExternal(data any) *External {
831     return &External{data, nil}
832 }
833
834 func (ctx *Context) NewExternalWithType(meta *Map, data any) *External {
835     if meta.name == nil {
836         panic("meta argument is not a metamap")
837     }
838
839     return &External{data, meta}
840 }
841
842 func (ctx *Context) NewValueFromSource(name string, source string) (Value, error) {
843     var location *SourceLocation = nil
844     if name != "" {
845         location = &SourceLocation{fmt.Sprintf("%s@builtin", name), 1}
846     }
847
848     lexer := NewLexer(ctx, source, location)
849     parser, err := NewParser(&lexer)
850     if err != nil {
851         return nil, err
852     }

```

```

850     }
851     program, err := parser.ParseProgram()
852     if err != nil {
853         return nil, err
854     }
855     env := NewEnvironment(ctx.BaseEnvironment)
856     return program.Eval(ctx, env)
857 }
858
859 func (ctx *Context) NewValueFromSourceOrPanic(name string, source string) Value {
860     value, err := ctx.NewValueFromSource(name, source)
861     if err != nil {
862         panic(err.Error())
863     }
864     return value
865 }
866
867 func (ctx *Context) GetModule(path string) (Value, bool) {
868     cached, ok := ctx.importFileCache[path]
869     if !ok {
870         return nil, false
871     }
872     return cached.Copy(), true
873 }
874
875 func (ctx *Context) SetModule(path string, value Value) {
876     ctx.importFileCache[path] = value.Copy()
877 }
878
879 type Null struct {
880 }
881
882 func (self *Null) Typename() string {
883     return "null"
884 }
885
886 func (self *Null) String() string {
887     return "null"
888 }
889
890 func (self *Null) Meta(ctx *Context) *Map {
891     return nil
892 }
893
894 func (self *Null) Take() Value {
895     return self // immutable value
896 }
897
898 func (self *Null) Copy() Value {
899     return self // immutable value
900 }
901
902 func (self *Null) CopyOnWrite() {
903     // immutable value
904 }
905
906 func (self *Null) Freeze() Value {
907     return self // immutable value
908 }
909
910 func (self *Null) IsImmutable() bool {
911     return true
912 }
913
914 func (self *Null) Hash() uint64 {
915     return 0
916 }

```

```

917
918 func (self *Null) Equal(other Value) bool {
919     _, ok := other.(*Null)
920     return ok
921 }
922
923 func (self *Null) CombEncode(e *CombEncoder) error {
924     return e.writeString(self.String())
925 }
926
927 type Boolean struct {
928     data bool
929 }
930
931 func (self *Boolean) Data() bool {
932     return self.data
933 }
934
935 func (self *Boolean) Typename() string {
936     return "boolean"
937 }
938
939 func (self *Boolean) String() string {
940     if self.data {
941         return "true"
942     }
943     return "false"
944 }
945
946 func (self *Boolean) Meta(ctx *Context) *Map {
947     return ctx.booleanMeta
948 }
949
950 func (self *Boolean) Take() Value {
951     return self // immutable value
952 }
953
954 func (self *Boolean) Copy() Value {
955     return self // immutable value
956 }
957
958 func (self *Boolean) CopyOnWrite() {
959     // immutable value
960 }
961
962 func (self *Boolean) Freeze() Value {
963     return self // immutable value
964 }
965
966 func (self *Boolean) IsImmutable() bool {
967     return true
968 }
969
970 func (self *Boolean) Hash() uint64 {
971     if self.data {
972         return 1
973     }
974     return 0
975 }
976
977 func (self *Boolean) Equal(other Value) bool {
978     othr, ok := other.(*Boolean)
979     if !ok {
980         return false
981     }
982     return self.data == othr.data
983 }

```

```

984
985 func (self *Boolean) CombEncode(e *CombEncoder) error {
986     return e.writeString(self.String())
987 }
988
989 type Number struct {
990     data float64
991 }
992
993 func (self *Number) Data() float64 {
994     return self.data
995 }
996
997 func (self *Number) Typename() string {
998     return "number"
999 }
1000
1001 func (self *Number) String() string {
1002     if math.IsNaN(self.data) {
1003         return "NaN"
1004     }
1005     if self.data == math.Inf(+1) {
1006         return "Inf"
1007     }
1008     if self.data == math.Inf(-1) {
1009         return "-Inf"
1010     }
1011     if self.data == 0 {
1012         // Special case for IEEE-754 +0 and -0, which are distinct in their bit
1013         // representation, but should always be stringified as "0".
1014         return "0"
1015     }
1016     if math.Trunc(self.data) == self.data {
1017         return strconv.FormatFloat(self.data, 'f', 0, 64)
1018     }
1019     return strconv.FormatFloat(self.data, 'f', -1, 64)
1020 }
1021
1022 func (self *Number) Meta(ctx *Context) *Map {
1023     return ctx.numberMeta
1024 }
1025
1026 func (self *Number) Take() Value {
1027     return self // immutable value
1028 }
1029
1030 func (self *Number) Copy() Value {
1031     return self // immutable value
1032 }
1033
1034 func (self *Number) CopyOnWrite() {
1035     // immutable value
1036 }
1037
1038 func (self *Number) Freeze() Value {
1039     return self // immutable value
1040 }
1041
1042 func (self *Number) IsImmutable() bool {
1043     return true
1044 }
1045
1046 func (self *Number) Hash() uint64 {
1047     if self.data == 0 {
1048         // Special case for IEEE-754 +0 and -0, which are distinct in their bit
1049         // representation, but should still compare equal as map keys.
1050         return 0

```

```

1051     }
1052     return fmix64(math.Float64bits(self.data))
1053 }
1054
1055 func (self *Number) Equal(other Value) bool {
1056     othr, ok := other.(*Number)
1057     if !ok {
1058         return false
1059     }
1060     return self.data == othr.data
1061 }
1062
1063 func (self *Number) CombEncode(e *CombEncoder) error {
1064     if e.err == nil && (math.IsInf(self.data, 0) || math.IsNaN(self.data)) {
1065         e.err = fmt.Errorf("invalid comb value %s", self.String())
1066         return e.err
1067     }
1068     return e.writeString(self.String())
1069 }
1070
1071 type String struct {
1072     data string
1073 }
1074
1075 func (self *String) Data() string {
1076     return self.data
1077 }
1078
1079 func (self *String) Typename() string {
1080     return "string"
1081 }
1082
1083 func (self *String) String() string {
1084     return fmt.Sprintf("%s", escape(self.data))
1085 }
1086
1087 func (self *String) Meta(ctx *Context) *Map {
1088     return ctx.stringMeta
1089 }
1090
1091 func (self *String) Take() Value {
1092     return self // immutable value
1093 }
1094
1095 func (self *String) Copy() Value {
1096     return self // immutable value
1097 }
1098
1099 func (self *String) CopyOnWrite() {
1100     // immutable value
1101 }
1102
1103 func (self *String) Freeze() Value {
1104     return self // immutable value
1105 }
1106
1107 func (self *String) IsImmutable() bool {
1108     return true
1109 }
1110
1111 func (self *String) Hash() uint64 {
1112     return fnv1a(self.data)
1113 }
1114
1115 func (self *String) Equal(other Value) bool {
1116     othr, ok := other.(*String)
1117     if !ok {

```

```
1118         return false
1119     }
1120     return self.data == othr.data
1121 }
1122
1123 func (self *String) CombEncode(e *CombEncoder) error {
1124     return e.writeString(self.String())
1125 }
1126
1127 type Regexp struct {
1128     data *regexp.Regexp
1129 }
1130
1131 func (self *Regexp) Data() *regexp.Regexp {
1132     return self.data
1133 }
1134
1135 func (self *Regexp) Typename() string {
1136     return "regexp"
1137 }
1138
1139 func (self *Regexp) String() string {
1140     return fmt.Sprintf("r\"%s\"", escape(self.data.String()))
1141 }
1142
1143 func (self *Regexp) Meta(ctx *Context) *Map {
1144     return ctx.regexpMeta
1145 }
1146
1147 func (self *Regexp) Take() Value {
1148     return self // immutable value
1149 }
1150
1151 func (self *Regexp) Copy() Value {
1152     return self // immutable value
1153 }
1154
1155 func (self *Regexp) CopyOnWrite() {
1156     // immutable value
1157 }
1158
1159 func (self *Regexp) Freeze() Value {
1160     return self // immutable value
1161 }
1162
1163 func (self *Regexp) IsImmutable() bool {
1164     return true
1165 }
1166
1167 func (self *Regexp) Hash() uint64 {
1168     return fnv1a(self.data.String())
1169 }
1170
1171 func (self *Regexp) Equal(other Value) bool {
1172     othr, ok := other.(*Regexp)
1173     if !ok {
1174         return false
1175     }
1176     return self.data.String() == othr.data.String()
1177 }
1178
1179 func (self *Regexp) CombEncode(e *CombEncoder) error {
1180     if e.err == nil {
1181         e.err = fmt.Errorf("invalid comb value %s", self.String())
1182     }
1183     return e.err
1184 }
```

```

1185
1186 type containerState int
1187
1188 const (
1189     // The container is currently free to be taken and owned by another entity.
1190     containerStateFree = iota
1191     // The container is currently owned and must be given before it can be
1192     // owned by another entity.
1193     containerStateOwned
1194     // The container is frozen and immutable, so giving or taking it will have
1195     // no effect.
1196     containerStateFrozen
1197 )
1198
1199 type vectorData struct {
1200     elements []Value
1201     uses     int
1202     refs     int
1203 }
1204
1205 type Vector struct {
1206     data *vectorData
1207     state containerState
1208 }
1209
1210 func (self *Vector) Typename() string {
1211     return "vector"
1212 }
1213
1214 func (self *Vector) String() string {
1215     if self.data == nil || len(self.data.elements) == 0 {
1216         return "[]"
1217     }
1218
1219     s := make([]string, len(self.data.elements))
1220     for i, element := range self.data.elements {
1221         s[i] = element.String()
1222     }
1223     return fmt.Sprintf("[%s]", strings.Join(s, ", "))
1224 }
1225
1226 func (self *Vector) Meta(ctx *Context) *Map {
1227     return ctx.vectorMeta
1228 }
1229
1230 func (self *Vector) Take() Value {
1231     switch self.state {
1232     case containerStateFree:
1233         // free -> owned
1234         self.state = containerStateOwned
1235     case containerStateOwned:
1236         panic("attempting to take owned vector")
1237     case containerStateFrozen:
1238         // immutable value
1239     default:
1240         panic("unreachable")
1241     }
1242     return self
1243 }
1244
1245 func (self *Vector) Copy() Value {
1246     if self.IsImmutable() {
1247         return self // immutable value
1248     }
1249
1250     if self.data == nil {
1251         return &Vector{data: nil}

```

```

1252     }
1253
1254     if self.data.refs > 0 {
1255         // Deep-copy elements to ensure that shared data containing at least
1256         // one reference does not allow for shared mutation of semantically
1257         // separate values created via copy.
1258         elements := make([]Value, len(self.data.elements))
1259         for i, element := range self.data.elements {
1260             elements[i] = element.Copy().Take()
1261         }
1262         return &Vector{
1263             data: &vectorData{
1264                 elements: elements,
1265                 uses:    1,
1266             },
1267         }
1268     }
1269
1270     self.data.uses += 1
1271     return &Vector{data: self.data}
1272 }
1273
1274 func (self *Vector) CopyOnWrite() {
1275     if self.data != nil && self.data.uses > 1 {
1276         self.data.uses -= 1
1277         elements := make([]Value, len(self.data.elements))
1278         for i, element := range self.data.elements {
1279             elements[i] = element.Copy().Take()
1280         }
1281         self.data = &vectorData{
1282             elements: elements,
1283             uses:    1,
1284         }
1285     }
1286 }
1287
1288 func (self *Vector) Freeze() Value {
1289     if self.IsImmutable() {
1290         return self
1291     }
1292
1293     value := self.Copy().(*Vector)
1294     value.CopyOnWrite()
1295     if value.data != nil {
1296         for i := range value.data.elements {
1297             value.data.elements[i] = value.data.elements[i].Freeze()
1298         }
1299     }
1300     value.state = containerStateFrozen
1301     return value
1302 }
1303
1304 func (self *Vector) IsImmutable() bool {
1305     return self.state == containerStateFrozen
1306 }
1307
1308 func (self *Vector) Hash() uint64 {
1309     var hash uint64 = 0
1310     if self.data != nil {
1311         for _, element := range self.data.elements {
1312             hash += element.Hash()
1313         }
1314     }
1315     return hash
1316 }
1317
1318 func (self *Vector) Equal(other Value) bool {

```

```

1319     othr, ok := other.(*Vector)
1320     if !ok {
1321         return false
1322     }
1323
1324     if self.Count() != othr.Count() {
1325         return false
1326     }
1327
1328     if self.data != nil && othr.data != nil {
1329         for i := range self.data.elements {
1330             if !self.data.elements[i].Equal(othr.data.elements[i]) {
1331                 return false
1332             }
1333         }
1334     }
1335     return true
1336 }
1337
1338 func (self *Vector) CombEncode(e *CombEncoder) error {
1339     if self.data == nil || len(self.data.elements) == 0 {
1340         e.writeString("[]")
1341         return e.err
1342     }
1343
1344     e.writeString("[")
1345     if e.Indent != nil {
1346         e.writeEndOfLine()
1347     }
1348     e.indentLevel += 1
1349
1350     for i, element := range self.data.elements {
1351         e.writeIndent("")
1352         element.CombEncode(e)
1353
1354         if i != len(self.data.elements)-1 {
1355             e.writeString(",")
1356             e.writeEndOfLine()
1357         } else if e.Indent != nil {
1358             e.writeEndOfLine()
1359         }
1360     }
1361
1362     e.indentLevel -= 1
1363     e.writeIndent("]")
1364
1365     return e.err
1366 }
1367
1368 func (self *Vector) Count() int {
1369     if self.data == nil {
1370         return 0
1371     }
1372
1373     return len(self.data.elements)
1374 }
1375
1376 func (self *Vector) Elements() []Value {
1377     if self.data == nil {
1378         return []Value{}
1379     }
1380     return self.data.elements
1381 }
1382
1383 func (self *Vector) Get(index int) Value {
1384     return self.data.elements[index]
1385 }

```

```

1386
1387 func (self *Vector) Set(index int, value Value) error {
1388     if self.IsImmutable() {
1389         return fmt.Errorf("attempted to modify immutable vector %v", self)
1390     }
1391
1392     if valueReference, ok := value.(*Reference); ok {
1393         return fmt.Errorf("attempted vector element assignment with element type
↪ reference to %v", valueReference.data.TypeName())
1394     }
1395
1396     self.CopyOnWrite()
1397     self.data.elements[index] = value.Take()
1398     return nil
1399 }
1400
1401 func (self *Vector) Insert(index int, value Value) error {
1402     if self.IsImmutable() {
1403         return fmt.Errorf("attempted to modify immutable vector %v", self)
1404     }
1405
1406     if valueReference, ok := value.(*Reference); ok {
1407         return fmt.Errorf("attempted vector element insert with element type reference to
↪ %v", valueReference.data.TypeName())
1408     }
1409
1410     self.CopyOnWrite()
1411
1412     if self.data == nil {
1413         self.data = &vectorData{
1414             elements: nil,
1415             uses:     1,
1416         }
1417     }
1418
1419     self.data.elements = slices.Insert(self.data.elements, index, value.Take())
1420     return nil
1421 }
1422
1423 func (self *Vector) Remove(index int) (Value, error) {
1424     if self.IsImmutable() {
1425         return nil, fmt.Errorf("attempted to modify immutable vector %v", self)
1426     }
1427
1428     self.CopyOnWrite()
1429     if self.data == nil || len(self.data.elements) == 0 {
1430         return nil, fmt.Errorf("attempted to remove from an empty vector")
1431     }
1432
1433     element := self.data.elements[index]
1434     self.data.elements = slices.Delete(self.data.elements, index, index+1)
1435     return element.Copy(), nil
1436 }
1437
1438 func (self *Vector) Push(value Value) error {
1439     if self.IsImmutable() {
1440         return fmt.Errorf("attempted to modify immutable vector %v", self)
1441     }
1442
1443     if valueReference, ok := value.(*Reference); ok {
1444         return fmt.Errorf("attempted vector element push with element type reference to
↪ %v", valueReference.data.TypeName())
1445     }
1446
1447     self.CopyOnWrite()
1448     if self.data == nil {
1449         self.data = &vectorData{

```

```

1450         elements: nil,
1451         uses:      1,
1452     }
1453 }
1454 self.data.elements = append(self.data.elements, value.Take())
1455 return nil
1456 }
1457
1458 // Returns nil when popping from an empty vector.
1459 func (self *Vector) Pop() (Value, error) {
1460     if self.IsImmutable() {
1461         return nil, fmt.Errorf("attempted to modify immutable vector %v", self)
1462     }
1463
1464     self.CopyOnWrite()
1465     if self.data == nil || len(self.data.elements) == 0 {
1466         return nil, nil
1467     }
1468
1469     element := self.data.elements[len(self.data.elements)-1]
1470     self.data.elements = self.data.elements[:len(self.data.elements)-1]
1471     return element.Copy(), nil
1472 }
1473
1474 type MapPair struct {
1475     Key    Value
1476     Value  Value
1477 }
1478
1479 const (
1480     orderedHashMapSmallCap = 8
1481     orderedHashMapSlotFree = -1
1482     orderedHashMapSlotDead = -2
1483 )
1484
1485 type orderedHashMapSmallEntry struct {
1486     hash uint64
1487     key   Value
1488     value Value
1489 }
1490
1491 type orderedHashMapLargeEntry struct {
1492     hash uint64
1493     key   Value
1494     value Value
1495     dead  bool
1496 }
1497
1498 type orderedHashMap struct {
1499     // Small Mode: Entries inline within small[:live] in insertion order.
1500     // Large Mode: Backing array small[:] is zeroed.
1501     small [orderedHashMapSmallCap]orderedHashMapSmallEntry
1502     // Small Mode: Nil slice.
1503     // Large Mode: Non-nil slice (including dead entries) in insertion order.
1504     large []orderedHashMapLargeEntry
1505     // Open-addressing index with len(index) as a power of two, where elements
1506     // are entry indices that in the free, live, or dead state.
1507     index []int32
1508     live  int // Count of live entries.
1509     dead  int // Count of dead entries (large mode only).
1510     used  int // Count of non-free index slots (large mode only).
1511 }
1512
1513 func (self *orderedHashMap) count() int {
1514     return self.live
1515 }
1516

```

```

1517 // SMALL MODE ONLY
1518 // Returns the index of the provided key with the provided hash in the small
1519 // entry buffer. Returns -1 if no entry is found.
1520 func (self *orderedHashMap) findSmall(key Value, hash uint64) int {
1521     for i := 0; i < self.live; i += 1 {
1522         if self.small[i].hash == hash && self.small[i].key.Equal(key) {
1523             return i
1524         }
1525     }
1526     return -1
1527 }
1528
1529 // LARGE MODE ONLY
1530 // Returns the index of the provided key with the provided hash in the large
1531 // entry buffer. Returns -1 if no entry is found.
1532 func (self *orderedHashMap) findLarge(key Value, hash uint64) int {
1533     mask := uint64(len(self.index) - 1)
1534     i := hash & mask
1535     for {
1536         slot := self.index[i]
1537         if slot == orderedHashMapSlotFree {
1538             return -1
1539         }
1540         if slot >= 0 {
1541             e := &self.large[slot]
1542             if e.hash == hash && e.key.Equal(key) {
1543                 return int(slot)
1544             }
1545         }
1546         i = (i + 1) & mask
1547     }
1548 }
1549
1550 func (self *orderedHashMap) lookup(key Value) (Value, bool) {
1551     hash := key.Hash()
1552     if self.large == nil {
1553         if i := self.findSmall(key, hash); i != -1 {
1554             return self.small[i].value, true
1555         }
1556         return nil, false
1557     }
1558     if i := self.findLarge(key, hash); i != -1 {
1559         return self.large[i].value, true
1560     }
1561     return nil, false
1562 }
1563
1564 func (self *orderedHashMap) lookupEntry(key Value) (Value, Value, bool) {
1565     hash := key.Hash()
1566     if self.large == nil {
1567         if i := self.findSmall(key, hash); i != -1 {
1568             return self.small[i].key, self.small[i].value, true
1569         }
1570     } else if i := self.findLarge(key, hash); i != -1 {
1571         return self.large[i].key, self.large[i].value, true
1572     }
1573     return nil, nil, false
1574 }
1575
1576 func (self *orderedHashMap) insert(key, value Value) {
1577     hash := key.Hash()
1578
1579     // Replace an existing entry if it already exists in the map.
1580     if self.large == nil {
1581         if i := self.findSmall(key, hash); i != -1 {
1582             self.small[i].key = key
1583             self.small[i].value = value

```

```

1584         return
1585     }
1586 } else if i := self.findLarge(key, hash); i != -1 {
1587     self.large[i].key = key
1588     self.large[i].value = value
1589     return
1590 }
1591
1592 if self.large == nil {
1593     if self.live < orderedHashMapSmallCap {
1594         self.small[self.live] = orderedHashMapSmallEntry{
1595             hash: hash,
1596             key: key,
1597             value: value,
1598         }
1599         self.live += 1
1600         return
1601     }
1602
1603     // Not enough storage in the small buffer. Transform from small mode to
1604     // large mode by migrating inline entries into a heap-allocated slice.
1605     self.large = make([]orderedHashMapLargeEntry, self.live,
1606 ↪ 2*orderedHashMapSmallCap)
1607     for i := 0; i < self.live; i += 1 {
1608         self.large[i] = orderedHashMapLargeEntry{
1609             hash: self.small[i].hash,
1610             key: self.small[i].key,
1611             value: self.small[i].value,
1612         }
1613     }
1614     // Clear memory of previously used small entries.
1615     self.small = [orderedHashMapSmallCap]orderedHashMapSmallEntry{}
1616
1617     // Build initial index.
1618     self.index = make([]int32, 4*orderedHashMapSmallCap)
1619     for i := range self.index {
1620         self.index[i] = orderedHashMapSlotFree
1621     }
1622     mask := uint64(len(self.index) - 1)
1623     for n := range self.large {
1624         i := self.large[n].hash & mask
1625         for self.index[i] != orderedHashMapSlotFree {
1626             i = (i + 1) & mask
1627         }
1628         self.index[i] = int32(n)
1629     }
1630     self.used = self.live
1631 }
1632
1633 // Rebuild threshold sits at a 3/4 load factor somewhat arbitrarily.
1634 if (self.used+1)*4 > len(self.index)*3 {
1635     self.rebuild()
1636 }
1637
1638 // Probe for a free or dead slot in the index.
1639 mask := uint64(len(self.index) - 1)
1640 i := hash & mask
1641 for {
1642     slot := self.index[i]
1643     if slot == orderedHashMapSlotFree {
1644         self.used += 1
1645         break
1646     }
1647     if slot == orderedHashMapSlotDead {
1648         break
1649     }
1650     i = (i + 1) & mask

```

```

1650     }
1651
1652     self.large = append(self.large, orderedHashMapLargeEntry{
1653         hash: hash,
1654         key:   key,
1655         value: value,
1656     })
1657     self.index[i] = int32(len(self.large) - 1)
1658     self.live += 1
1659 }
1660
1661 func (self *orderedHashMap) rebuild() {
1662     out := 0
1663     for i := range self.large {
1664         if self.large[i].dead {
1665             continue
1666         }
1667         if out != i {
1668             self.large[out] = self.large[i]
1669         }
1670         out += 1
1671     }
1672     clear(self.large[out:])
1673     self.large = self.large[:out]
1674     self.dead = 0
1675
1676     capacity := 4 * orderedHashMapSmallCap
1677     // Keep load at or below a 3/8 load factor, chosen somewhat arbitrarily.
1678     for capacity*3 < (self.live+1)*8 {
1679         capacity *= 2
1680     }
1681     if capacity == len(self.index) {
1682         for i := range self.index {
1683             self.index[i] = orderedHashMapSlotFree
1684         }
1685     } else {
1686         self.index = make([]int32, capacity)
1687         for i := range self.index {
1688             self.index[i] = orderedHashMapSlotFree
1689         }
1690     }
1691
1692     mask := uint64(len(self.index) - 1)
1693     for n := range self.large {
1694         i := self.large[n].hash & mask
1695         for self.index[i] != orderedHashMapSlotFree {
1696             i = (i + 1) & mask
1697         }
1698         self.index[i] = int32(n)
1699     }
1700     self.used = out
1701 }
1702
1703 func (self *orderedHashMap) remove(key Value) {
1704     hash := key.Hash()
1705     if self.large == nil {
1706         i := self.findSmall(key, hash)
1707         if i == -1 {
1708             return
1709         }
1710         copy(self.small[i:self.live-1], self.small[i+1:self.live])
1711         // Clear memory of previously used small entry.
1712         self.small[self.live-1] = orderedHashMapSmallEntry{}
1713         self.live -= 1
1714         return
1715     }
1716

```

```

1717     mask := uint64(len(self.index) - 1)
1718     i := hash & mask
1719     for {
1720         slot := self.index[i]
1721         if slot == orderedHashMapSlotFree {
1722             return
1723         }
1724         if slot >= 0 {
1725             e := &self.large[slot]
1726             if e.hash == hash && e.key.Equal(key) {
1727                 self.index[i] = orderedHashMapSlotDead
1728                 // Clear memory of previous entry, while setting a tombstone.
1729                 *e = orderedHashMapLargeEntry{dead: true}
1730                 self.live -= 1
1731                 self.dead += 1
1732                 // Keep iteration 0(live) in the case of many removals.
1733                 if self.dead > self.live {
1734                     self.rebuild()
1735                 }
1736                 return
1737             }
1738         }
1739         i = (i + 1) & mask
1740     }
1741 }
1742
1743 // Iterate over key-value pairs in insertion order.
1744 // The receiver ordered hash map must not be modified during iteration.
1745 func (self *orderedHashMap) entries() iter.Seq2[Value, Value] {
1746     return func(yield func(Value, Value) bool) {
1747         if self.large == nil {
1748             for i := 0; i < self.live; i += 1 {
1749                 if !yield(self.small[i].key, self.small[i].value) {
1750                     return
1751                 }
1752             }
1753             return
1754         }
1755         for i := range self.large {
1756             e := &self.large[i]
1757             if e.dead {
1758                 continue
1759             }
1760             if !yield(e.key, e.value) {
1761                 return
1762             }
1763         }
1764     }
1765 }
1766
1767 // Iterate over keys in insertion order.
1768 // The receiver ordered hash map must not be modified during iteration.
1769 func (self *orderedHashMap) keys() iter.Seq[Value] {
1770     return func(yield func(Value) bool) {
1771         for k := range self.entries() {
1772             if !yield(k) {
1773                 return
1774             }
1775         }
1776     }
1777 }
1778
1779 // Iterate over values in insertion order.
1780 // The receiver ordered hash map must not be modified during iteration.
1781 func (self *orderedHashMap) values() iter.Seq[Value] {
1782     return func(yield func(Value) bool) {
1783         for _, v := range self.entries() {

```

```

1784         if !yield(v) {
1785             return
1786         }
1787     }
1788 }
1789 }
1790
1791 type mapData struct {
1792     hmap orderedHashMap
1793     uses int
1794     refs int
1795 }
1796
1797 type Map struct {
1798     data *mapData
1799     meta *Map // Optional (non-nil implies that this is a mutable map)
1800     name *string // Optional (non-nil implies that this is a frozen metamap)
1801     state containerState
1802 }
1803
1804 func (self *Map) Typename() string {
1805     if self.meta != nil && self.meta.name != nil {
1806         // Instance of a custom type that has its own name.
1807         return *self.meta.name
1808     }
1809     return "map"
1810 }
1811
1812 func (self *Map) String() string {
1813     if self.data == nil || self.data.hmap.count() == 0 {
1814         return "Map{}"
1815     }
1816
1817     s := make([]string, 0)
1818     for key, value := range self.data.hmap.entries() {
1819         s = append(s, fmt.Sprintf("%s: %s", key.String(), value.String()))
1820     }
1821     return fmt.Sprintf("{%s}", strings.Join(s, ", "))
1822 }
1823
1824 func (self *Map) Meta(ctx *Context) *Map {
1825     return self.meta
1826 }
1827
1828 func (self *Map) Take() Value {
1829     switch self.state {
1830     case containerStateFree:
1831         // free -> owned
1832         self.state = containerStateOwned
1833     case containerStateOwned:
1834         panic("attempting to take owned map")
1835     case containerStateFrozen:
1836         // immutable value
1837     default:
1838         panic("unreachable")
1839     }
1840     return self
1841 }
1842
1843 func (self *Map) Copy() Value {
1844     if self.IsImmutable() {
1845         return self // immutable value
1846     }
1847
1848     if self.data == nil {
1849         return &Map{
1850             data: nil,

```

```

1851         meta: self.meta,
1852         name: self.name,
1853     }
1854 }
1855
1856 if self.data.refs > 0 {
1857     // Deep-copy elements to ensure that shared data containing at least
1858     // one reference does not allow for shared mutation of semantically
1859     // separate values created via copy.
1860     data := &mapData{uses: 1}
1861     for key, value := range self.data.hmap.entries() {
1862         data.hmap.insert(key.Copy().Take(), value.Copy().Take())
1863     }
1864     return &Map{
1865         data: data,
1866         meta: self.meta,
1867         name: self.name,
1868     }
1869 }
1870
1871 self.data.uses += 1
1872 return &Map{
1873     data: self.data,
1874     meta: self.meta,
1875     name: self.name,
1876 }
1877 }
1878
1879 func (self *Map) CopyOnWrite() {
1880     if self.data != nil && self.data.uses > 1 {
1881         self.data.uses -= 1
1882         data := &mapData{
1883             uses: 1,
1884         }
1885
1886         for key, value := range self.data.hmap.entries() {
1887             data.hmap.insert(key.Copy().Take(), value.Copy().Take())
1888         }
1889         self.data = data
1890     }
1891 }
1892
1893 func (self *Map) Freeze() Value {
1894     if self.IsImmutable() {
1895         return self
1896     }
1897
1898     value := self.Copy().(*Map)
1899     value.CopyOnWrite()
1900     if value.data != nil {
1901         pairs := value.Pairs()
1902         for _, pair := range pairs {
1903             err := value.Insert(pair.Key.Freeze(), pair.Value.Freeze())
1904             if err != nil {
1905                 panic(err) // should never occur
1906             }
1907         }
1908     }
1909     value.state = containerStateFrozen
1910     return value
1911 }
1912
1913 func (self *Map) IsImmutable() bool {
1914     return self.state == containerStateFrozen
1915 }
1916
1917 func (self *Map) Hash() uint64 {

```

```

1918     var hash uint64 = 0
1919     if self.data != nil {
1920         for key, value := range self.data.hmap.entries() {
1921             hash += key.Hash() + value.Hash()
1922         }
1923     }
1924     return hash
1925 }
1926
1927 func (self *Map) Equal(other Value) bool {
1928     othr, ok := other.(*Map)
1929     if !ok {
1930         return false
1931     }
1932
1933     if self.Count() != othr.Count() {
1934         return false
1935     }
1936     if self.Count() == 0 {
1937         // Empty maps.
1938         return true
1939     }
1940
1941     // Non-empty maps - self and othr both have non-nil data.
1942     if self.data == othr.data {
1943         // Self and other share the same backing map data. Equality depends on
1944         // each key and value being equal to itself, which is itself dependent
1945         // on whether any nested key or value contains a NaN value.
1946         for key, value := range self.data.hmap.entries() {
1947             if !key.Equal(key) || !value.Equal(value) {
1948                 return false
1949             }
1950         }
1951         return true
1952     }
1953
1954     // Non-empty maps - self and othr both have separate non-nil data.
1955     for key, value := range self.data.hmap.entries() {
1956         othrValue, ok := othr.Lookup(key)
1957         if !ok {
1958             return false
1959         }
1960         if !value.Equal(othrValue) {
1961             return false
1962         }
1963     }
1964
1965     return true
1966 }
1967
1968 func (self *Map) CombEncode(e *CombEncoder) error {
1969     if self.data == nil || self.data.hmap.count() == 0 {
1970         e.writeString("Map{}")
1971         return e.err
1972     }
1973
1974     e.writeString("{")
1975     if e.Indent != nil {
1976         e.writeEndOfLine()
1977     }
1978     e.indentLevel += 1
1979
1980     remaining := self.data.hmap.count()
1981     for key, value := range self.data.hmap.entries() {
1982         e.writeIndent("")
1983         key.CombEncode(e)
1984         e.writeString(":")

```

```

1985         e.writeString(e.Separator)
1986         value.CombEncode(e)
1987
1988         remaining -= 1
1989         if remaining != 0 {
1990             e.writeString(",")
1991             e.writeEndOfLine()
1992         } else if e.Indent != nil {
1993             e.writeEndOfLine()
1994         }
1995     }
1996
1997     e.indentLevel -= 1
1998     e.writeIndent("}")
1999
2000     return e.err
2001 }
2002
2003 func (self *Map) Count() int {
2004     if self.data == nil {
2005         return 0
2006     }
2007
2008     return self.data.hmap.count()
2009 }
2010
2011 func (self *Map) Pairs() []MapPair {
2012     if self.data == nil {
2013         return []MapPair{}
2014     }
2015
2016     pairs := make([]MapPair, 0, self.data.hmap.count())
2017     for key, value := range self.data.hmap.entries() {
2018         pairs = append(pairs, MapPair{key, value})
2019     }
2020     return pairs
2021 }
2022
2023 // Returns nil on lookup failure.
2024 func (self *Map) Get(key Value) Value {
2025     lookup, _ := self.Lookup(key)
2026     return lookup
2027 }
2028
2029 func (self *Map) Lookup(key Value) (Value, bool) {
2030     if self.data == nil {
2031         return nil, false
2032     }
2033
2034     return self.data.hmap.lookup(key)
2035 }
2036
2037 func (self *Map) Insert(key, value Value) error {
2038     if self.IsImmutable() {
2039         return fmt.Errorf("attempted to modify immutable map %v", self)
2040     }
2041
2042     if keyReference, ok := key.(*Reference); ok {
2043         return fmt.Errorf("attempted map insert with key type reference to %v",
2044             ↪ keyReference.data.TypeName())
2045     }
2046
2047     if valueReference, ok := value.(*Reference); ok {
2048         return fmt.Errorf("attempted map insert with value type reference to %v",
2049             ↪ valueReference.data.TypeName())
2050     }

```

```

2050     if keyNumber, ok := key.(*Number); ok {
2051         if math.IsNaN(keyNumber.data) {
2052             return errors.New("invalid NaN map key")
2053         }
2054     }
2055
2056     self.CopyOnWrite()
2057     if self.data == nil {
2058         self.data = &mapData{
2059             uses: 1,
2060         }
2061     }
2062
2063     self.data.hmap.insert(key.Take(), value.Take())
2064     return nil
2065 }
2066
2067 func (self *Map) Remove(key Value) error {
2068     if self.IsImmutable() {
2069         return fmt.Errorf("attempted to modify immutable map %v", self)
2070     }
2071
2072     if keyReference, ok := key.(*Reference); ok {
2073         return fmt.Errorf("attempted map remove with key type reference to %v",
2074 ↪ keyReference.data.TypeName())
2075     }
2076
2077     if self.data == nil {
2078         return nil
2079     }
2080
2081     self.CopyOnWrite()
2082     self.data.hmap.remove(key)
2083     return nil
2084 }
2085
2086 type setData struct {
2087     hmap orderedHashMap // Entries of the form {key, nil}.
2088     uses int
2089     refs int
2090 }
2091
2092 type Set struct {
2093     data *setData
2094     state containerState
2095 }
2096
2097 func (self *Set) TypeName() string {
2098     return "set"
2099 }
2100
2101 func (self *Set) String() string {
2102     if self.data == nil || self.data.hmap.count() == 0 {
2103         return "Set{}"
2104     }
2105
2106     s := make([]string, 0)
2107     for element := range self.data.hmap.keys() {
2108         s = append(s, element.String())
2109     }
2110     return fmt.Sprintf("{%s}", strings.Join(s, ", "))
2111 }
2112
2113 func (self *Set) Meta(ctx *Context) *Map {
2114     return ctx.setMeta
2115 }

```

```

2116 func (self *Set) Take() Value {
2117     switch self.state {
2118     case containerStateFree:
2119         // free -> owned
2120         self.state = containerStateOwned
2121     case containerStateOwned:
2122         panic("attempting to take owned set")
2123     case containerStateFrozen:
2124         // immutable value
2125     default:
2126         panic("unreachable")
2127     }
2128     return self
2129 }
2130
2131 func (self *Set) Copy() Value {
2132     if self.IsImmutable() {
2133         return self // immutable value
2134     }
2135
2136     if self.data == nil {
2137         return &Set{data: nil}
2138     }
2139
2140     if self.data.refs > 0 {
2141         // Deep-copy elements to ensure that shared data containing at least
2142         // one reference does not allow for shared mutation of semantically
2143         // separate values created via copy.
2144         data := &setData{uses: 1}
2145         for element := range self.data.hmap.keys() {
2146             data.hmap.insert(element.Copy().Take(), nil)
2147         }
2148         return &Set{data: data}
2149     }
2150
2151     self.data.uses += 1
2152     return &Set{data: self.data}
2153 }
2154
2155 func (self *Set) CopyOnWrite() {
2156     if self.data != nil && self.data.uses > 1 {
2157         self.data.uses -= 1
2158         data := &setData{
2159             uses: 1,
2160         }
2161
2162         for element := range self.data.hmap.keys() {
2163             data.hmap.insert(element.Copy().Take(), nil)
2164         }
2165         self.data = data
2166     }
2167 }
2168
2169 func (self *Set) Freeze() Value {
2170     if self.IsImmutable() {
2171         return self
2172     }
2173
2174     value := self.Copy().(*Set)
2175     value.CopyOnWrite()
2176     if value.data != nil {
2177         for _, element := range value.Elements() {
2178             err := value.Insert(element.Freeze())
2179             if err != nil {
2180                 panic(err) // should never occur
2181             }
2182         }

```

```

2183     }
2184     value.state = containerStateFrozen
2185     return value
2186 }
2187
2188 func (self *Set) IsImmutable() bool {
2189     return self.state == containerStateFrozen
2190 }
2191
2192 func (self *Set) Hash() uint64 {
2193     var hash uint64 = 0
2194     if self.data != nil {
2195         for element := range self.data.hmap.keys() {
2196             hash += element.Hash()
2197         }
2198     }
2199     return hash
2200 }
2201
2202 func (self *Set) Equal(other Value) bool {
2203     othr, ok := other.(*Set)
2204     if !ok {
2205         return false
2206     }
2207
2208     if self.Count() != othr.Count() {
2209         return false
2210     }
2211     if self.Count() == 0 {
2212         // Empty sets.
2213         return true
2214     }
2215
2216     // Non-empty sets - self and othr both have non-nil data.
2217     if self.data == othr.data {
2218         // Self and othr share the same backing set data. Equality depends on
2219         // each element being equal to itself, which is itself dependent on
2220         // whether any nested elements contain a NaN value.
2221         for element := range self.data.hmap.keys() {
2222             if !element.Equal(element) {
2223                 return false
2224             }
2225         }
2226         return true
2227     }
2228
2229     // Non-empty sets - self and othr both have separate non-nil data.
2230     for element := range self.data.hmap.keys() {
2231         if _, ok := othr.Lookup(element); !ok {
2232             return false
2233         }
2234     }
2235
2236     return true
2237 }
2238
2239 func (self *Set) CombEncode(e *CombEncoder) error {
2240     if self.data == nil || self.data.hmap.count() == 0 {
2241         e.writeString("Set{}")
2242         return e.err
2243     }
2244
2245     e.writeString("{")
2246     if e.Indent != nil {
2247         e.writeEndOfLine()
2248     }
2249     e.indentLevel += 1

```

```

2250
2251     remaining := self.data.hmap.count()
2252     for element := range self.data.hmap.keys() {
2253         e.writeIndent("")
2254         element.CombEncode(e)
2255
2256         remaining -= 1
2257         if remaining != 0 {
2258             e.writeString(",")
2259             e.writeEndOfLine()
2260         } else if e.Indent != nil {
2261             e.writeEndOfLine()
2262         }
2263     }
2264
2265     e.indentLevel -= 1
2266     e.writeIndent("}")
2267
2268     return e.err
2269 }
2270
2271 func (self *Set) Count() int {
2272     if self.data == nil {
2273         return 0
2274     }
2275
2276     return self.data.hmap.count()
2277 }
2278
2279 func (self *Set) Elements() []Value {
2280     if self.data == nil {
2281         return []Value{}
2282     }
2283
2284     elements := make([]Value, 0, self.data.hmap.count())
2285     for element := range self.data.hmap.keys() {
2286         elements = append(elements, element)
2287     }
2288     return elements
2289 }
2290
2291 // Returns nil on lookup failure.
2292 func (self *Set) Get(value Value) Value {
2293     lookup, _ := self.Lookup(value)
2294     return lookup
2295 }
2296
2297 func (self *Set) Lookup(value Value) (Value, bool) {
2298     if self.data == nil {
2299         return nil, false
2300     }
2301
2302     element, _, ok := self.data.hmap.lookupEntry(value)
2303     if !ok {
2304         return nil, false
2305     }
2306
2307     return element, true
2308 }
2309
2310 func (self *Set) Insert(value Value) error {
2311     if self.IsImmutable() {
2312         return fmt.Errorf("attempted to modify immutable set %v", self)
2313     }
2314
2315     if valueReference, ok := value.(*Reference); ok {

```

```

2316         return fmt.Errorf("attempted set insert with element type reference to %v",
↪ valueReference.data.TypeName())
2317     }
2318
2319     if valueNumber, ok := value.(*Number); ok {
2320         if math.IsNaN(valueNumber.data) {
2321             return errors.New("invalid NaN set element")
2322         }
2323     }
2324
2325     self.CopyOnWrite()
2326     if self.data == nil {
2327         self.data = &setData{
2328             uses: 1,
2329         }
2330     }
2331
2332     self.data.hmap.insert(value.Take(), nil)
2333     return nil
2334 }
2335
2336 func (self *Set) Remove(value Value) error {
2337     if self.IsImmutable() {
2338         return fmt.Errorf("attempted to modify immutable set %v", self)
2339     }
2340
2341     if valueReference, ok := value.(*Reference); ok {
2342         ↪ return fmt.Errorf("attempted set remove with element type reference to %v",
↪ valueReference.data.TypeName())
2343     }
2344
2345     if self.data == nil {
2346         return nil
2347     }
2348
2349     self.CopyOnWrite()
2350     self.data.hmap.remove(value)
2351     return nil
2352 }
2353
2354 type Reference struct {
2355     data Value
2356 }
2357
2358 func (self *Reference) Data() Value {
2359     return self.data
2360 }
2361
2362 func (self *Reference) TypeName() string {
2363     return "reference"
2364 }
2365
2366 func (self *Reference) String() string {
2367     return fmt.Sprintf("reference@%p", self.data)
2368 }
2369
2370 func (self *Reference) Meta(ctx *Context) *Map {
2371     return ctx.referenceMeta
2372 }
2373
2374 func (self *Reference) Take() Value {
2375     return self // immutable value
2376 }
2377
2378 func (self *Reference) Copy() Value {
2379     return self // immutable value
2380 }

```

```

2381
2382 func (self *Reference) CopyOnWrite() {
2383     // immutable value
2384 }
2385
2386 func (self *Reference) Freeze() Value {
2387     return self // immutable value
2388 }
2389
2390 func (self *Reference) IsImmutable() bool {
2391     return true
2392 }
2393
2394 func (self *Reference) Hash() uint64 {
2395     return fmix64(uint64(reflect.ValueOf(self.data).Pointer()))
2396 }
2397
2398 func (self *Reference) Equal(other Value) bool {
2399     othr, ok := other.(*Reference)
2400     if !ok {
2401         return false
2402     }
2403     return reflect.ValueOf(self.data).Pointer() == reflect.ValueOf(othr.data).Pointer()
2404 }
2405
2406 func (self *Reference) CombEncode(e *CombEncoder) error {
2407     if e.err == nil {
2408         e.err = fmt.Errorf("invalid comb value %s", self.String())
2409     }
2410     return e.err
2411 }
2412
2413 type Function struct {
2414     ast *AstExpressionFunction
2415     env *Environment
2416 }
2417
2418 func (self *Function) Typename() string {
2419     return "function"
2420 }
2421
2422 func (self *Function) String() string {
2423     name := "function"
2424     if self.ast.Name != nil {
2425         name = self.ast.Name.data
2426     }
2427     for _, r := range name {
2428         if !(unicode.IsLetter(r) || unicode.IsDigit(r) || r == '_') {
2429             name = escape(name)
2430             break
2431         }
2432     }
2433     if self.ast.Location != nil {
2434         return fmt.Sprintf("%s@[%v, line %v]", name, self.ast.Location.File,
↵ self.ast.Location.Line)
2435     }
2436     return name
2437 }
2438
2439 func (self *Function) Meta(ctx *Context) *Map {
2440     return ctx.functionMeta
2441 }
2442
2443 func (self *Function) Take() Value {
2444     return self // immutable value
2445 }
2446

```

mellifera.go

```
2447 func (self *Function) Copy() Value {
2448     return self // immutable value
2449 }
2450
2451 func (self *Function) CopyOnWrite() {
2452     // immutable value
2453 }
2454
2455 func (self *Function) Freeze() Value {
2456     return self // immutable value
2457 }
2458
2459 func (self *Function) IsImmutable() bool {
2460     return true
2461 }
2462
2463 func (self *Function) Hash() uint64 {
2464     return fmix64(uint64(reflect.ValueOf(self.ast).Pointer() +
2465 ↪ reflect.ValueOf(self.env).Pointer()))
2466 }
2467
2468 func (self *Function) Equal(other Value) bool {
2469     othr, ok := other.(*Function)
2470     if !ok {
2471         return false
2472     }
2473     return reflect.ValueOf(self.ast).Pointer() == reflect.ValueOf(othr.ast).Pointer() &&
2474 ↪ reflect.ValueOf(self.env).Pointer() == reflect.ValueOf(othr.env).Pointer()
2475 }
2476
2477 func (self *Function) CombEncode(e *CombEncoder) error {
2478     if e.err == nil {
2479         e.err = fmt.Errorf("invalid comb value %s", self.String())
2480     }
2481     return e.err
2482 }
2483
2484 type Builtin struct {
2485     name string
2486     types []Type
2487     impl func(*Context, []Value) (Value, error)
2488     byRef bool
2489 }
2490
2491 func (self *Builtin) Typename() string {
2492     return "function"
2493 }
2494
2495 func (self *Builtin) String() string {
2496     return fmt.Sprintf("%s@builtin", self.name)
2497 }
2498
2499 func (self *Builtin) Meta(ctx *Context) *Map {
2500     return ctx.functionMeta
2501 }
2502
2503 func (self *Builtin) Take() Value {
2504     return self // immutable value
2505 }
2506
2507 func (self *Builtin) Copy() Value {
2508     return self // immutable value
2509 }
2510
2511 func (self *Builtin) CopyOnWrite() {
2512     // immutable value
2513 }
```

```

2512
2513 func (self *Builtin) Freeze() Value {
2514     return self // immutable value
2515 }
2516
2517 func (self *Builtin) IsImmutable() bool {
2518     return true
2519 }
2520
2521 func (self *Builtin) Hash() uint64 {
2522     return fnv1a(self.name)
2523 }
2524
2525 func (self *Builtin) Equal(other Value) bool {
2526     othr, ok := other.(*Builtin)
2527     if !ok {
2528         return false
2529     }
2530     return self.name == othr.name && reflect.ValueOf(self.impl).Pointer() ==
↪ reflect.ValueOf(othr.impl).Pointer()
2531 }
2532
2533 func (self *Builtin) CombEncode(e *CombEncoder) error {
2534     if e.err == nil {
2535         e.err = fmt.Errorf("invalid comb value %s", self.String())
2536     }
2537     return e.err
2538 }
2539
2540 type External struct {
2541     data any
2542     meta *Map // Optional
2543 }
2544
2545 func (self *External) Data() any {
2546     return self.data
2547 }
2548
2549 func (self *External) Typename() string {
2550     if self.meta != nil && self.meta.name != nil {
2551         // Instance of a custom type that has its own name.
2552         return *self.meta.name
2553     }
2554     return "external"
2555 }
2556
2557 func (self *External) String() string {
2558     return fmt.Sprintf("external(%v)", self.data)
2559 }
2560
2561 func (self *External) Meta(ctx *Context) *Map {
2562     return self.meta
2563 }
2564
2565 func (self *External) Take() Value {
2566     return self // immutable value
2567 }
2568
2569 func (self *External) Copy() Value {
2570     return self // immutable value
2571 }
2572
2573 func (self *External) CopyOnWrite() {
2574     // immutable value
2575 }
2576
2577 func (self *External) Freeze() Value {

```

mellifera.go

```
2578     return self // immutable value
2579 }
2580
2581 func (self *External) IsImmutable() bool {
2582     return true
2583 }
2584
2585 func (self *External) Hash() uint64 {
2586     return fnv1a(self.String())
2587 }
2588
2589 func (self *External) Equal(other Value) bool {
2590     othr, ok := other.(*External)
2591     if !ok {
2592         return false
2593     }
2594     return self.data == othr.data
2595 }
2596
2597 func (self *External) CombEncode(e *CombEncoder) error {
2598     if e.err == nil {
2599         e.err = fmt.Errorf("invalid comb value %s", self.String())
2600     }
2601     return e.err
2602 }
2603
2604 type SourceLocation struct {
2605     File string
2606     Line int
2607 }
2608
2609 func optionalSourceLocationIntoValue(ctx *Context, location *SourceLocation) Value {
2610     if location == nil {
2611         return ctx.Null
2612     }
2613     return ctx.NewMapOrPanic([]MapPair{
2614         {ctx.NewString("file"), ctx.NewString(location.File)},
2615         {ctx.NewString("line"), ctx.NewNumber(float64(location.Line))},
2616     })
2617 }
2618
2619 type ParseError struct {
2620     Location *SourceLocation // Optional
2621     why      string
2622 }
2623
2624 func (self ParseError) Error() string {
2625     return self.why
2626 }
2627
2628 // Token Kinds
2629 const (
2630     // Meta
2631     TOKEN_EOF = "end-of-file"
2632     // Identifiers and Literals
2633     TOKEN_IDENTIFIER = "identifier"
2634     TOKEN_TEMPLATE   = "template"
2635     TOKEN_NUMBER     = "number"
2636     TOKEN_STRING     = "string"
2637     TOKEN_REGEXP     = "regexp"
2638     TOKEN_REGEXP_GROUP = "regexp group"
2639     // Operators
2640     TOKEN_ADD = "+"
2641     TOKEN_SUB = "-"
2642     TOKEN_MUL = "*"
2643     TOKEN_DIV = "/"
2644     TOKEN_REM = "%"
```

```

2645     TOKEN_EQ      = "=="
2646     TOKEN_NE      = "!="
2647     TOKEN_LE      = "<="
2648     TOKEN_GE      = ">="
2649     TOKEN_LT      = "<"
2650     TOKEN_GT      = ">"
2651     TOKEN_EQ_RE    = "=~"
2652     TOKEN_NE_RE    = "!~"
2653     TOKEN_MKREF    = "&"
2654     TOKEN_DEREF    = ".*"
2655     TOKEN_DOT      = "."
2656     TOKEN_SCOPE    = "::"
2657     TOKEN_ASSIGN   = "="
2658     // Delimiters
2659     TOKEN_COMMA     = ","
2660     TOKEN_COLON     = ":"
2661     TOKEN_SEMICOLON = ";"
2662     TOKEN_LPAREN    = "("
2663     TOKEN_RPAREN    = ")"
2664     TOKEN_LBRACE    = "{"
2665     TOKEN_RBRACE    = "}"
2666     TOKEN_LBRACKET  = "["
2667     TOKEN_RBRACKET  = "]"
2668     // Keywords
2669     TOKEN_TYPE      = "type"
2670     TOKEN_EXTENDS   = "extends"
2671     TOKEN_NULL      = "null"
2672     TOKEN_TRUE      = "true"
2673     TOKEN_FALSE     = "false"
2674     TOKEN_MAP       = "Map"
2675     TOKEN_SET       = "Set"
2676     TOKEN_FUNCTION  = "function"
2677     TOKEN_NEW       = "new"
2678     TOKEN_NOT       = "not"
2679     TOKEN_AND       = "and"
2680     TOKEN_OR        = "or"
2681     TOKEN_LET       = "let"
2682     TOKEN_IF        = "if"
2683     TOKEN_ELIF      = "elif"
2684     TOKEN_ELSE      = "else"
2685     TOKEN_FOR       = "for"
2686     TOKEN_IN        = "in"
2687     TOKEN_WHILE     = "while"
2688     TOKEN_BREAK     = "break"
2689     TOKEN_CONTINUE  = "continue"
2690     TOKEN_TRY       = "try"
2691     TOKEN_CATCH     = "catch"
2692     TOKEN_ERROR     = "error"
2693     TOKEN_RETURN    = "return"
2694     TOKEN_FREEZE    = "freeze"
2695 )
2696
2697 type Token struct {
2698     Kind      string
2699     Literal   string
2700     Location  *SourceLocation // Optional
2701     Value     Value         // Optional
2702     Template  []AstExpression // Optional
2703     Group     int           // Optional
2704 }
2705
2706 func (self Token) String() string {
2707     if self.Literal != "" {
2708         return self.Literal
2709     }
2710     return self.Kind
2711 }

```

```

2712
2713 func (self Token) IntoValue(ctx *Context) Value {
2714     return ctx.NewMapOrPanic([]MapPair{
2715         {ctx.NewString("kind"), ctx.NewString(self.Kind)},
2716         {ctx.NewString("literal"), ctx.NewString(self.Literal)},
2717         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
2718     })
2719 }
2720
2721 type Lexer struct {
2722     ctx      *Context
2723     source   string
2724     location *SourceLocation // Optional
2725     position int
2726
2727     keywords map[string]string
2728 }
2729
2730 func NewLexer(ctx *Context, source string, location *SourceLocation) Lexer {
2731     keywords := map[string]string{
2732         TOKEN_NULL:    TOKEN_NULL,
2733         TOKEN_TRUE:    TOKEN_TRUE,
2734         TOKEN_FALSE:   TOKEN_FALSE,
2735         TOKEN_MAP:     TOKEN_MAP,
2736         TOKEN_SET:     TOKEN_SET,
2737         TOKEN_FUNCTION: TOKEN_FUNCTION,
2738         TOKEN_TYPE:    TOKEN_TYPE,
2739         TOKEN_EXTENDS: TOKEN_EXTENDS,
2740         TOKEN_NEW:     TOKEN_NEW,
2741         TOKEN_NOT:     TOKEN_NOT,
2742         TOKEN_AND:     TOKEN_AND,
2743         TOKEN_OR:      TOKEN_OR,
2744         TOKEN_LET:     TOKEN_LET,
2745         TOKEN_IF:      TOKEN_IF,
2746         TOKEN_ELIF:    TOKEN_ELIF,
2747         TOKEN_ELSE:    TOKEN_ELSE,
2748         TOKEN_FOR:     TOKEN_FOR,
2749         TOKEN_IN:      TOKEN_IN,
2750         TOKEN_WHILE:   TOKEN_WHILE,
2751         TOKEN_BREAK:   TOKEN_BREAK,
2752         TOKEN_CONTINUE: TOKEN_CONTINUE,
2753         TOKEN_TRY:     TOKEN_TRY,
2754         TOKEN_CATCH:   TOKEN_CATCH,
2755         TOKEN_ERROR:   TOKEN_ERROR,
2756         TOKEN_RETURN:  TOKEN_RETURN,
2757         TOKEN_FREEZE:  TOKEN_FREEZE,
2758     }
2759
2760     return Lexer{
2761         ctx:      ctx,
2762         source:   source,
2763         location: location,
2764         position: 0,
2765
2766         keywords: keywords,
2767     }
2768 }
2769
2770 func (self *Lexer) currentLocation() *SourceLocation {
2771     if self.location == nil {
2772         return nil
2773     }
2774     return &SourceLocation{self.location.File, self.location.Line}
2775 }
2776
2777 func (self *Lexer) currentRune() rune {
2778     if self.position >= len(self.source) {

```

```

2779         return rune(0)
2780     }
2781     current, _ := utf8.DecodeRuneInString(self.source[self.position:])
2782     return current
2783 }
2784
2785 func (self *Lexer) peekRune() rune {
2786     if self.position+1 >= len(self.source) {
2787         return rune(0)
2788     }
2789     _, currentSize := utf8.DecodeRuneInString(self.source[self.position:])
2790     peek, _ := utf8.DecodeRuneInString(self.source[self.position+currentSize:])
2791     return peek
2792 }
2793
2794 func (self *Lexer) isEof() bool {
2795     return self.position >= len(self.source)
2796 }
2797
2798 func (self *Lexer) remaining() string {
2799     return self.source[self.position:]
2800 }
2801
2802 func (self *Lexer) advanceRune() {
2803     if self.isEof() {
2804         return
2805     }
2806     if self.location != nil && self.currentRune() == '\n' {
2807         self.location.Line += 1
2808     }
2809     _, size := utf8.DecodeRuneInString(self.source[self.position:])
2810     self.position += size
2811 }
2812
2813 func (self *Lexer) expectRune(r rune) error {
2814     if self.isEof() {
2815         return ParseError{
2816             Location: self.currentLocation(),
2817             why:      fmt.Sprintf("expected %s, found end-of-file",
↳ quote(string([]rune{r}))),
2818         }
2819     }
2820     current := self.currentRune()
2821     if current != r {
2822         return ParseError{
2823             Location: self.currentLocation(),
2824             why:      fmt.Sprintf("expected %s, found %s", quote(string([]rune{r})),
↳ quote(string([]rune{current}))),
2825         }
2826     }
2827     self.advanceRune()
2828     return nil
2829 }
2830
2831 func (self *Lexer) skipWhitespace() {
2832     for !self.isEof() && unicode.IsSpace(self.currentRune()) {
2833         self.advanceRune()
2834     }
2835 }
2836
2837 func (self *Lexer) skipComment() {
2838     if self.currentRune() != '#' {
2839         return
2840     }
2841     for !self.isEof() && self.currentRune() != '\n' {
2842         self.advanceRune()
2843     }

```

```

2844     self.advanceRune()
2845 }
2846
2847 func (self *Lexer) skipWhiteSpaceAndComments() {
2848     for !self.isEof() && (unicode.IsSpace(self.currentRune()) || self.currentRune() ==
↳ '#'') {
2849         self.skipWhitespace()
2850         self.skipComment()
2851     }
2852 }
2853
2854 func (self *Lexer) lexKeywordOrIdentifier() (Token, error) {
2855     if !(unicode.IsLetter(self.currentRune()) || self.currentRune() == '_') {
2856         return Token{}, errors.New("empty keyword or identifier")
2857     }
2858     literal := ""
2859     for unicode.IsLetter(self.currentRune()) || unicode.IsNumber(self.currentRune()) ||
↳ self.currentRune() == '_' {
2860         literal += string(self.currentRune())
2861         self.advanceRune()
2862     }
2863
2864     keyword, ok := self.keywords[literal]
2865     if ok {
2866         return Token{
2867             Kind:    keyword,
2868             Literal: literal,
2869             Location: self.currentLocation(),
2870         }, nil
2871     }
2872     return Token{
2873         Kind:    TOKEN_IDENTIFIER,
2874         Literal:  literal,
2875         Location: self.currentLocation(),
2876     }, nil
2877 }
2878
2879 func (self *Lexer) lexNumber() (Token, error) {
2880     hex := self.ctx.reNumberHex.FindString(self.source[self.position:])
2881     if hex != "" {
2882         self.position += len(hex)
2883         parsed, err := strconv.ParseInt(hex, 0, 64)
2884         if err != nil {
2885             if errors.Is(err, strconv.ErrRange) {
2886                 return Token{}, ParseError{
2887                     Location: self.currentLocation(),
2888                     why:     fmt.Sprintf("hexadecimal integer %s is outside the safe
↳ integer range", hex),
2889                 }
2890             }
2891             return Token{}, err
2892         }
2893         if parsed < MIN_SAFE_INTEGER || parsed > MAX_SAFE_INTEGER {
2894             return Token{}, ParseError{
2895                 Location: self.currentLocation(),
2896                 why:     fmt.Sprintf("hexadecimal integer %s is outside the safe integer
↳ range", hex),
2897             }
2898         }
2899         return Token{
2900             Kind:    TOKEN_NUMBER,
2901             Literal:  hex,
2902             Location: self.currentLocation(),
2903             Value:   self.ctx.NewNumber(float64(parsed)),
2904         }, nil
2905     }
2906 }

```

```

2907     dec := self.ctx.reNumberDec.FindString(self.source[self.position:])
2908     if dec != "" {
2909         self.position += len(dec)
2910         parsed, err := strconv.ParseFloat(dec, 64)
2911         if err != nil {
2912             return Token{}, err
2913         }
2914         return Token{
2915             Kind:    TOKEN_NUMBER,
2916             Literal: dec,
2917             Location: self.currentLocation(),
2918             Value:    self.ctx.NewNumber(parsed),
2919         }, nil
2920     }
2921
2922     return Token{}, errors.New("invalid number")
2923 }
2924
2925 func (self *Lexer) lexEscStringPart() ([]byte, error) {
2926     if self.isEof() {
2927         return nil, ParseError{
2928             Location: self.currentLocation(),
2929             why:      "expected character, found end-of-file",
2930         }
2931     }
2932
2933     if self.currentRune() == '\n' {
2934         return nil, ParseError{
2935             Location: self.currentLocation(),
2936             why:      "expected character, found newline",
2937         }
2938     }
2939
2940     if !unicode.IsPrint(self.currentRune()) {
2941         return nil, ParseError{
2942             Location: self.currentLocation(),
2943             why:      fmt.Sprintf("expected printable character, found %#x",
↪ self.currentRune()),
2944         }
2945     }
2946
2947     if self.currentRune() == '\\' && self.peekRune() == 't' {
2948         self.advanceRune()
2949         self.advanceRune()
2950         return []byte("\t"), nil
2951     }
2952
2953     if self.currentRune() == '\\' && self.peekRune() == 'n' {
2954         self.advanceRune()
2955         self.advanceRune()
2956         return []byte("\n"), nil
2957     }
2958
2959     if self.currentRune() == '\\' && self.peekRune() == '"' {
2960         self.advanceRune()
2961         self.advanceRune()
2962         return []byte("\\"), nil
2963     }
2964
2965     if self.currentRune() == '\\' && self.peekRune() == '\\' {
2966         self.advanceRune()
2967         self.advanceRune()
2968         return []byte("\\\\"), nil
2969     }
2970
2971     if self.currentRune() == '\\' && self.peekRune() == 'x' {
2972         self.advanceRune()

```

```

2973     self.advanceRune()
2974     nybbles := []rune{self.currentRune(), self.peekRune()}
2975     self.advanceRune()
2976     self.advanceRune()
2977     mapping := map[rune]int{
2978         '0': 0x0,
2979         '1': 0x1,
2980         '2': 0x2,
2981         '3': 0x3,
2982         '4': 0x4,
2983         '5': 0x5,
2984         '6': 0x6,
2985         '7': 0x7,
2986         '8': 0x8,
2987         '9': 0x9,
2988         'A': 0xA,
2989         'B': 0xB,
2990         'C': 0xC,
2991         'D': 0xD,
2992         'E': 0xE,
2993         'F': 0xF,
2994         'a': 0xA,
2995         'b': 0xB,
2996         'c': 0xC,
2997         'd': 0xD,
2998         'e': 0xE,
2999         'f': 0xF,
3000     }
3001     nybble0, foundNybble0 := mapping[nybbles[0]]
3002     nybble1, foundNybble1 := mapping[nybbles[1]]
3003     if !(foundNybble0 && foundNybble1) {
3004         sequence := "\\x" + string(nybbles)
3005         return nil, ParseError{
3006             Location: self.currentLocation(),
3007             why:      fmt.Sprintf("expected hexadecimal escape sequence, found %s",
3008 ↪ quote(sequence)),
3009         }
3010     }
3011     return []byte{byte(nybble0<<4 | nybble1)}, nil
3012 }
3013 if self.currentRune() == '\\' {
3014     sequence := string([]rune{self.currentRune(), self.peekRune()})
3015     return nil, ParseError{
3016         Location: self.currentLocation(),
3017         why:      fmt.Sprintf("expected escape sequence, found %s",
3018 ↪ quote(escape(sequence))),
3019     }
3020 }
3021 character := self.currentRune()
3022 self.advanceRune()
3023 return []byte(string(character)), nil
3024 }
3025
3026 func (self *Lexer) lexEscString() (Token, error) {
3027     start := self.position
3028     if err := self.expectRune('"'); err != nil {
3029         return Token{}, err
3030     }
3031     bytes := []byte{}
3032     for !self.isEof() && self.currentRune() != '"' {
3033         part, err := self.lexEscStringPart()
3034         if err != nil {
3035             return Token{}, err
3036         }
3037         bytes = append(bytes, part...)

```

```

3038     }
3039     if err := self.expectRune(''); err != nil {
3040         return Token{}, err
3041     }
3042     literal := self.source[start:self.position]
3043     return Token{
3044         Kind:    TOKEN_STRING,
3045         Literal:  literal,
3046         Location: self.currentLocation(),
3047         Value:    self.ctx.NewString(string(bytes)),
3048     }, nil
3049 }
3050
3051 func (self *Lexer) lexRawStringPart() ([]byte, error) {
3052     if self.isEof() {
3053         return []byte{}, ParseError{
3054             Location: self.currentLocation(),
3055             why:      "expected character, found end-of-file",
3056         }
3057     }
3058
3059     character := self.currentRune()
3060     self.advanceRune()
3061     return []byte(string(character)), nil
3062 }
3063
3064 func (self *Lexer) lexRawString() (Token, error) {
3065     location := self.currentLocation()
3066     start := self.position
3067     bytes := []byte{}
3068     var literal string
3069     if strings.HasPrefix(self.remaining(), "```") {
3070         if err := self.expectRune(''); err != nil {
3071             return Token{}, err
3072         }
3073         if err := self.expectRune('`'); err != nil {
3074             return Token{}, err
3075         }
3076         if err := self.expectRune('`'); err != nil {
3077             return Token{}, err
3078         }
3079         for !self.isEof() && !strings.HasPrefix(self.remaining(), "```") {
3080             part, err := self.lexRawStringPart()
3081             if err != nil {
3082                 return Token{}, err
3083             }
3084             bytes = append(bytes, part...)
3085         }
3086         if err := self.expectRune('`'); err != nil {
3087             return Token{}, err
3088         }
3089         if err := self.expectRune('`'); err != nil {
3090             return Token{}, err
3091         }
3092         if err := self.expectRune('`'); err != nil {
3093             return Token{}, err
3094         }
3095         literal = self.source[start+3 : self.position-3]
3096         // Future-proof in case I want to add variable-number-of-tick raw
3097         // string literals in the future.
3098         if len(bytes) == 0 {
3099             return Token{}, ParseError{
3100                 Location: location,
3101                 why:      "invalid empty multi-tick raw string",
3102             }
3103         }
3104     } else {

```

```

3105         if err := self.expectRune('`'); err != nil {
3106             return Token{}, err
3107         }
3108         for !self.isEof() && self.currentRune() != '`' {
3109             part, err := self.lexRawStringPart()
3110             if err != nil {
3111                 return Token{}, err
3112             }
3113             bytes = append(bytes, part...)
3114         }
3115         if err := self.expectRune('`'); err != nil {
3116             return Token{}, err
3117         }
3118         literal = self.source[start:self.position]
3119     }
3120     return Token{
3121         Kind:    TOKEN_STRING,
3122         Literal: literal,
3123         Location: location,
3124         Value:    self.ctx.NewString(string(bytes)),
3125     }, nil
3126 }
3127
3128 func (self *Lexer) lexTemplate() (Token, error) {
3129     location := self.currentLocation()
3130     start := self.position
3131     if err := self.expectRune('$'); err != nil {
3132         return Token{}, err
3133     }
3134
3135     template := []AstExpression{}
3136     bytes := []byte{} // current text being parsed
3137
3138     lexTemplateElement := func(defaultFunc func() ([]byte, error)) error {
3139         if strings.HasPrefix(self.remaining(), "{{") {
3140             bytes = append(bytes, []byte("{}")...)
3141             self.position += len("{}")
3142             return nil
3143         }
3144
3145         if strings.HasPrefix(self.remaining(), "}}") {
3146             bytes = append(bytes, []byte("{}")...)
3147             self.position += len("{}")
3148             return nil
3149         }
3150
3151         if strings.HasPrefix(self.remaining(), "{") {
3152             if len(bytes) != 0 {
3153                 template = append(template, &AstExpressionString{location,
3154 ↪ self.ctx.NewString(string(bytes))})
3155             }
3156             bytes = []byte{}
3157             self.position += len("{}")
3158
3159             lexer := NewLexer(self.ctx, self.remaining(), self.currentLocation())
3160             parser, err := NewParser(&lexer)
3161             if err != nil {
3162                 return err
3163             }
3164             expression, err := parser.ParseExpression()
3165             if err != nil {
3166                 return ParseError{
3167                     Location: location,
3168                     why:     err.Error(),
3169                 }
3170             }
3171             template = append(template, expression)

```

```

3171         if parser.currentToken.Kind != TOKEN_RBRACE {
3172             return ParseError{
3173                 Location: location,
3174                 why:      fmt.Sprintf("expected `}` to close template expression,
↪ found %s", quote(parser.currentToken.Kind)),
3175             }
3176         }
3177         self.position += strings.LastIndex(self.remaining(), lexer.remaining())
3178
3179         return nil
3180     }
3181
3182     text, err := defaultFunc()
3183     if err != nil {
3184         return err
3185     }
3186     bytes = append(bytes, text...)
3187     return nil
3188 }
3189
3190 if strings.HasPrefix(self.remaining(), "`") {
3191     if err := self.expectRune('`'); err != nil {
3192         return Token{}, err
3193     }
3194     if err := self.expectRune('`'); err != nil {
3195         return Token{}, err
3196     }
3197     if err := self.expectRune('`'); err != nil {
3198         return Token{}, err
3199     }
3200 }
3201 for !self.isEof() && !strings.HasPrefix(self.remaining(), "`") {
3202     err := lexTemplateElement(func() ([]byte, error) {
3203         return self.lexRawStringPart()
3204     })
3205     if err != nil {
3206         return Token{}, err
3207     }
3208 }
3209 if len(bytes) != 0 {
3210     template = append(template, &AstExpressionString{location,
↪ self.ctx.NewString(string(bytes))})
3211 }
3212 if err := self.expectRune('`'); err != nil {
3213     return Token{}, err
3214 }
3215 if err := self.expectRune('`'); err != nil {
3216     return Token{}, err
3217 }
3218 if err := self.expectRune('`'); err != nil {
3219     return Token{}, err
3220 }
3221 } else if strings.HasPrefix(self.remaining(), "") {
3222     if err := self.expectRune('`'); err != nil {
3223         return Token{}, err
3224     }
3225     for !self.isEof() && self.currentRune() != '`' {
3226         err := lexTemplateElement(func() ([]byte, error) {
3227             return self.lexRawStringPart()
3228         })
3229         if err != nil {
3230             return Token{}, err
3231         }
3232     }
3233     if len(bytes) != 0 {
3234         template = append(template, &AstExpressionString{location,
↪ self.ctx.NewString(string(bytes))})

```

```

3235     }
3236     if err := self.expectRune('`'); err != nil {
3237         return Token{}, err
3238     }
3239 } else if self.currentRune() == '"' {
3240     if err := self.expectRune('"'); err != nil {
3241         return Token{}, err
3242     }
3243     for self.currentRune() != '"' {
3244         err := lexTemplateElement(func() ([]byte, error) {
3245             return self.lexEscStringPart()
3246         })
3247         if err != nil {
3248             return Token{}, err
3249         }
3250     }
3251     if len(bytes) != 0 {
3252         template = append(template, &AstExpressionString{location,
↪ self.ctx.NewString(string(bytes))})
3253     }
3254     if err := self.expectRune('"'); err != nil {
3255         return Token{}, err
3256     }
3257 } else {
3258     return Token{}, ParseError{
3259         Location: location,
3260         why:      fmt.Sprintf("expected template of the form `${\"...\"}`, `${...}` or
↪ `${`...`}`, found `${` followed by %s", quote(string(self.currentRune()))),
3261     }
3262 }
3263
3264 literal := self.source[start:self.position]
3265 return Token{
3266     Kind:    TOKEN_TEMPLATE,
3267     Literal: literal,
3268     Location: location,
3269     Template: template,
3270 }, nil
3271 }
3272
3273 func (self *Lexer) lexRegexp() (Token, error) {
3274     location := self.currentLocation()
3275     start := self.position
3276     if err := self.expectRune('r'); err != nil {
3277         return Token{}, err
3278     }
3279
3280     bytes := []byte{}
3281     if self.currentRune() == '"' {
3282         if err := self.expectRune('"'); err != nil {
3283             return Token{}, err
3284         }
3285         for self.currentRune() != '"' {
3286             part, err := self.lexEscStringPart()
3287             if err != nil {
3288                 return Token{}, err
3289             }
3290             bytes = append(bytes, part...)
3291         }
3292         if err := self.expectRune('"'); err != nil {
3293             return Token{}, err
3294         }
3295     } else if self.currentRune() == '`' {
3296         if err := self.expectRune('`'); err != nil {
3297             return Token{}, err
3298         }
3299         for self.currentRune() != '`' {

```

```

3300         part, err := self.lexRawStringPart()
3301         if err != nil {
3302             return Token{}, err
3303         }
3304         bytes = append(bytes, part...)
3305     }
3306     if err := self.expectRune('`'); err != nil {
3307         return Token{}, err
3308     }
3309 } else {
3310     return Token{}, ParseError{
3311         Location: location,
3312         why:      fmt.Sprintf("expected %s or %s, found %s", quote("\'"), quote("`"),
3313 ↪ quote(string(self.currentRune()))),
3314     }
3315 }
3316 literal := self.source[start:self.position]
3317 regexp, err := self.ctx.NewRegexp(string(bytes))
3318 if err != nil {
3319     return Token{}, ParseError{
3320         Location: location,
3321         why:      err.Error(),
3322     }
3323 }
3324 return Token{
3325     Kind:    TOKEN_REGEXP,
3326     Literal: literal,
3327     Location: location,
3328     Value:   regexp,
3329 }, nil
3330 }
3331 func (self *Lexer) lexRegexpGroup() (Token, error) {
3332     location := self.currentLocation()
3333     start := self.position
3334     if err := self.expectRune('$'); err != nil {
3335         return Token{}, err
3336     }
3337
3338     integer := self.ctx.reIntegerDec.FindString(self.source[self.position:])
3339     if integer != "" {
3340         self.position += len(integer)
3341         parsed, err := strconv.ParseInt(integer, 10, 0)
3342         if err != nil || parsed < MIN_SAFE_INTEGER || parsed > MAX_SAFE_INTEGER {
3343             return Token{}, ParseError{
3344                 Location: location,
3345                 why:      fmt.Sprintf("invalid regexp capture group %v", integer),
3346             }
3347         }
3348         literal := self.source[start:self.position]
3349         return Token{
3350             Kind:    TOKEN_REGEXP_GROUP,
3351             Literal: literal,
3352             Location: location,
3353             Group:   int(parsed),
3354         }, nil
3355     }
3356
3357     return Token{}, ParseError{
3358         Location: location,
3359         why:      "invalid regexp capture group",
3360     }
3361 }
3362
3363 func (self *Lexer) NextToken() (Token, error) {
3364     self.skipWhiteSpaceAndComments()
3365     if self.isEof() {

```

```

3366         return Token{
3367             Kind:    TOKEN_EOF,
3368             Literal: "",
3369             Location: self.currentLocation(),
3370         }, nil
3371     }
3372
3373     location := self.currentLocation()
3374
3375     // Literals, Identifiers, and Keywords
3376     if '0' <= self.currentRune() && self.currentRune() <= '9' {
3377         return self.lexNumber()
3378     }
3379     if self.currentRune() == '"' {
3380         return self.lexEscString()
3381     }
3382     if self.currentRune() == '`' {
3383         return self.lexRawString()
3384     }
3385     if strings.HasPrefix(self.remaining(), "$\\") || strings.HasPrefix(self.remaining(),
↵ "$`") {
3386         return self.lexTemplate()
3387     }
3388     if strings.HasPrefix(self.remaining(), "r\\") || strings.HasPrefix(self.remaining(),
↵ "r`") {
3389         return self.lexRegex()
3390     }
3391     if self.currentRune() == '$' && unicode.IsNumber(self.peekRune()) {
3392         return self.lexRegexGroup()
3393     }
3394     if unicode.IsLetter(self.currentRune()) || self.currentRune() == '_' {
3395         return self.lexKeywordOrIdentifier()
3396     }
3397
3398     // Operators and Delimiters
3399     matchRemaining := func(kind string) (Token, bool) {
3400         if strings.HasPrefix(self.remaining(), kind) {
3401             literal := self.source[self.position : self.position+len(kind)]
3402             self.position += len(kind)
3403             return Token{
3404                 Kind:    kind,
3405                 Literal: literal,
3406                 Location: location,
3407             }, true
3408         }
3409         return Token{}, false
3410     }
3411     operatorsAndDelimiters := []string{
3412         // Operators
3413         TOKEN_ADD,
3414         TOKEN_SUB,
3415         TOKEN_MUL,
3416         TOKEN_DIV,
3417         TOKEN_REM,
3418         TOKEN_EQ,
3419         TOKEN_NE,
3420         TOKEN_LE,
3421         TOKEN_GE,
3422         TOKEN_LT,
3423         TOKEN_GT,
3424         TOKEN_EQ_RE,
3425         TOKEN_NE_RE,
3426         TOKEN_MKREF,
3427         TOKEN_DEREF,
3428         TOKEN_DOT,
3429         TOKEN_SCOPE,
3430         TOKEN_ASSIGN,

```

```

3431         // Delimiters
3432         TOKEN_COMMA,
3433         TOKEN_COLON,
3434         TOKEN_SEMICOLON,
3435         TOKEN_LPAREN,
3436         TOKEN_RPAREN,
3437         TOKEN_LBRACE,
3438         TOKEN_RBRACE,
3439         TOKEN_LBRACKET,
3440         TOKEN_RBRACKET,
3441     }
3442     for _, d := range operatorsAndDelimiters {
3443         if token, match := matchRemaining(d); match {
3444             return token, nil
3445         }
3446     }
3447
3448     return Token{}, ParseError{
3449         Location: self.currentLocation(),
3450         why:      fmt.Sprintf("unknown token %s",
3451 ↪ quote(string([]rune{self.currentRune()}))),
3452     }
3453 }
3454
3455 type TraceElement struct {
3456     Location *SourceLocation // Optional
3457     FuncName string
3458 }
3459
3460 type Error struct {
3461     Location *SourceLocation // Optional
3462     Value    Value
3463     Trace    []TraceElement
3464 }
3465
3466 func (self Error) Error() string {
3467     if s, ok := self.Value.(*String); ok {
3468         return s.data
3469     }
3470     return self.Value.String()
3471 }
3472
3473 func NewError(location *SourceLocation, value Value) Error {
3474     return Error{
3475         Location: location,
3476         Value:    value,
3477         Trace:    []TraceElement{},
3478     }
3479 }
3480
3481 type regexpMatch struct {
3482     // String data tested by the last regular expression operation (=~, !~).
3483     string string
3484     // Result of the last regular expression operation (=~, !~) returned by:
3485     //
3486     //      rhsRegexp.data.FindStringSubmatchIndex(input)
3487     //
3488     // where the Nth capture group corresponds to the [bgnIndex, endIndex]
3489     // slice denoted by result[2*N:2*N+2]. A slice index pair of [-1, -1]
3490     // indicates the Nth capture group did not match. A nil slice indicates
3491     // that the last regular expression operation did not match.
3492     result []int
3493 }
3494
3495 type envRefInfo struct {
3496     location *SourceLocation // Optional
3497     name     string

```

mellifera.go

```
3497     value    Value
3498 }
3499
3500 type envElement struct {
3501     name string
3502     value Value
3503 }
3504
3505 var envPool = sync.Pool{
3506     New: func() any {
3507         return &Environment{}
3508     },
3509 }
3510
3511 const (
3512     envHashMapSmallCap = 8
3513     envHashMapSlotFree = -1
3514 )
3515
3516 type Environment struct {
3517     store []envElement
3518     index []int32      // Optional (non-nil implies large mode)
3519     outer *Environment // Optional
3520     match *regexpMatch // Optional
3521     refs  []envRefInfo
3522     nref  int
3523     held  bool
3524 }
3525
3526 func NewEnvironment(outer *Environment) *Environment {
3527     nref := 0
3528     if outer != nil {
3529         nref = outer.nref
3530     }
3531
3532     self := envPool.Get().(*Environment)
3533     *self = Environment{
3534         store: self.store,
3535         outer: outer,
3536         match: nil,
3537         refs:  self.refs,
3538         nref:  nref,
3539     }
3540     return self
3541 }
3542
3543 // SMALL MODE ONLY
3544 // Returns the index of name within the store via linear scan.
3545 // Returns -1 if name is not declared.
3546 func (self *Environment) findSmall(name string) int {
3547     for i := range self.store {
3548         if self.store[i].name == name {
3549             return i
3550         }
3551     }
3552     return -1
3553 }
3554
3555 // LARGE MODE ONLY
3556 // Returns the index of name within the store via the open-addressing index.
3557 // Returns -1 if name is not declared.
3558 func (self *Environment) findLarge(name string, hash uint64) int {
3559     mask := uint64(len(self.index) - 1)
3560     i := hash & mask
3561     for {
3562         slot := self.index[i]
3563         if slot == envHashMapSlotFree {
```

```

3564         return -1
3565     }
3566     if self.store[slot].name == name {
3567         return int(slot)
3568     }
3569     i = (i + 1) & mask
3570 }
3571 }
3572
3573 // Generate the hash map index with the provided power-of-two slot capacity.
3574 func (self *Environment) indexRebuild(capacity int) {
3575     self.index = make([]int32, capacity)
3576     for i := range self.index {
3577         self.index[i] = envHashMapSlotFree
3578     }
3579     mask := uint64(capacity - 1)
3580     for i := range self.store {
3581         slot := fnv1a(self.store[i].name) & mask
3582         for self.index[slot] != envHashMapSlotFree {
3583             slot = (slot + 1) & mask
3584         }
3585         self.index[slot] = int32(i)
3586     }
3587 }
3588
3589 // Update the hash map index with a mapping to the provided store index.
3590 func (self *Environment) indexInsert(name string, storeIndex int) {
3591     mask := uint64(len(self.index) - 1)
3592     slot := fnv1a(name) & mask
3593     for self.index[slot] != envHashMapSlotFree {
3594         slot = (slot + 1) & mask
3595     }
3596     self.index[slot] = int32(storeIndex)
3597 }
3598
3599 // Mark that this environment has been closed-over by at least one function
3600 // that now holds a live path to all values in this and outer environments.
3601 func (self *Environment) markHeld() {
3602     env := self
3603     for env != nil {
3604         if env.held {
3605             // The current environment being held implies all outer
3606             // environments are also held.
3607             break
3608         }
3609         env.held = true
3610         env = env.outer
3611     }
3612 }
3613
3614 // Return this environment to the environment pool if it has not been
3615 // closed-over by at least one function.
3616 func (self *Environment) tryReuse() {
3617     if self.held {
3618         return
3619     }
3620
3621     clear(self.store) // re-use store allocation
3622     clear(self.refs)  // re-use refs allocation
3623     *self = Environment{store: self.store[:0], refs: self.refs[:0]}
3624     envPool.Put(self)
3625 }
3626
3627 func (self *Environment) Let(name string, value Value) {
3628     self.letWithLocation(name, value, nil)
3629 }
3630

```

```

3631 func (self *Environment) letWithLocation(name string, value Value, location
↪ *SourceLocation) {
3632     if _, ok := value.(*Reference); ok {
3633         self.refs = append(self.refs, envRefInfo{location, name, value})
3634         self.nref += 1
3635     }
3636
3637     index := -1
3638     if self.index == nil {
3639         index = self.findSmall(name)
3640     } else {
3641         index = self.findLarge(name, fnv1a(name))
3642     }
3643     if index != -1 {
3644         self.store[index].value = value.Take()
3645         return
3646     }
3647     self.store = append(self.store, envElement{name, value.Take()})
3648
3649     if len(self.store) > envHashMapSmallCap {
3650         if self.index == nil {
3651             // Switch from small mode to large mode.
3652             self.indexRebuild(4 * envHashMapSmallCap)
3653         } else if len(self.store)*4 >= len(self.index)*3 {
3654             // Rebuild threshold sits at a 3/4 load factor somewhat arbitrarily.
3655             self.indexRebuild(2 * len(self.index))
3656         } else {
3657             self.indexInsert(name, len(self.store)-1)
3658         }
3659     }
3660 }
3661
3662 func (self *Environment) Set(name string, value Value) error {
3663     // References *should* only ever be introduced to a lexical scope via a
3664     // call to Environment.letWithLocation() from within the Mellifera runtime.
3665     if _, ok := value.(*Reference); ok {
3666         return errors.New("attempted assignment with a reference value")
3667     }
3668
3669     env := self
3670     for env != nil {
3671         index := -1
3672         if env.index == nil {
3673             index = env.findSmall(name)
3674         } else {
3675             index = env.findLarge(name, fnv1a(name))
3676         }
3677         if index != -1 {
3678             env.store[index].value = value.Take()
3679             return nil
3680         }
3681         env = env.outer
3682     }
3683     return fmt.Errorf("identifier %s is not defined", quote(name))
3684 }
3685
3686 func (self *Environment) Get(name string) (Value, error) {
3687     env := self
3688     for env != nil {
3689         index := -1
3690         if env.index == nil {
3691             index = env.findSmall(name)
3692         } else {
3693             index = env.findLarge(name, fnv1a(name))
3694         }
3695         if index != -1 {
3696             return env.store[index].value, nil

```

```

3697     }
3698     env = env.outer
3699 }
3700 return nil, fmt.Errorf("identifier %s is not defined", quote(name))
3701 }
3702
3703 func (self *Environment) GetOrPanic(name string) Value {
3704     value, err := self.Get(name)
3705     if err != nil {
3706         panic(err)
3707     }
3708     return value
3709 }
3710
3711 func (self *Environment) setRegexMatch(match regexMatch) {
3712     self.match = &match
3713 }
3714
3715 func (self *Environment) getRegexMatch() (regexMatch, bool) {
3716     env := self
3717     for env != nil {
3718         if env.match != nil {
3719             return *env.match, env.match.result != nil
3720         }
3721         env = env.outer
3722     }
3723     return regexMatch{}, false
3724 }
3725
3726 type ControlFlowKind int
3727
3728 const (
3729     ControlFlowNone ControlFlowKind = iota
3730     ControlFlowReturn
3731     ControlFlowBreak
3732     ControlFlowContinue
3733 )
3734
3735 type ControlFlow struct {
3736     Kind      ControlFlowKind
3737     Location  *SourceLocation // Optional
3738     Value     Value           // ControlFlowReturn only
3739 }
3740
3741 // Update the name values of named functions that are children somewhere in
3742 // this map, either direct map-level values or a descendent of another map.
3743 func updateNamedFunctions(ctx *Context, ast *AstExpressionMap, prefix string) {
3744     for _, pair := range ast.Elements {
3745         k := pair.Key
3746         v := pair.Value
3747
3748         kAstExpressionString, ok := k.(*AstExpressionString)
3749         if !ok {
3750             continue
3751         }
3752
3753         if vAstExpressionFunction, ok := v.(*AstExpressionFunction); ok {
3754             vAstExpressionFunction.Name = ctx.NewString(prefix +
3755 ↪ kAstExpressionString.Data.data)
3756             continue
3757         }
3758         if vAstExpressionMap, ok := v.(*AstExpressionMap); ok {
3759             updateNamedFunctions(ctx, vAstExpressionMap,
3760 ↪ prefix+kAstExpressionString.Data.data+TOKEN_SCOPE)
3761             continue
3762         }
3763     }
3764 }

```

```

3762 }
3763
3764 type AstNode interface {
3765     IntoValue(*Context) Value
3766 }
3767
3768 type AstExpression interface {
3769     ExpressionLocation() *SourceLocation
3770     IntoValue(*Context) Value
3771     Eval(*Context, *Environment) (Value, error)
3772 }
3773
3774 type AstStatement interface {
3775     StatementLocation() *SourceLocation
3776     IntoValue(*Context) Value
3777     Eval(*Context, *Environment) (ControlFlow, error)
3778 }
3779
3780 type AstProgram struct {
3781     Location    *SourceLocation // Optional
3782     Statements []AstStatement
3783 }
3784
3785 func (self AstProgram) IntoValue(ctx *Context) Value {
3786     statements := ctx.NewVectorOrPanic(nil)
3787     for _, statement := range self.Statements {
3788         _ = statements.Push(statement.IntoValue(ctx))
3789     }
3790     return ctx.NewMapOrPanic([]MapPair{
3791         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
3792         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
3793         {ctx.NewString("statements"), statements},
3794     })
3795 }
3796
3797 func (self *AstProgram) Eval(ctx *Context, env *Environment) (Value, error) {
3798     var result Value = ctx.Null
3799     for _, statement := range self.Statements {
3800         if statement.Expression, ok := statement.(*AstStatementExpression); ok {
3801             // If a top-level statement is an expression, directly evaluate
3802             // that expression and save the result. This allows the result of
3803             // the last top-level expression statement to be used as the
3804             // result of program execution.
3805             value, error := statement.Expression.Expression.Eval(ctx, env)
3806             if error != nil {
3807                 return nil, error
3808             }
3809             result = value
3810             continue
3811         }
3812
3813         cflow, error := statement.Eval(ctx, env)
3814         if error != nil {
3815             return nil, error
3816         }
3817         if cflow.Kind == ControlFlowReturn {
3818             return cflow.Value, nil
3819         }
3820         if cflow.Kind == ControlFlowBreak {
3821             return nil, NewError(cflow.Location, ctx.NewString("attempted to break
↪ outside of a loop"))
3822         }
3823         if cflow.Kind == ControlFlowContinue {
3824             return nil, NewError(cflow.Location, ctx.NewString("attempted to continue
↪ outside of a loop"))
3825         }
3826     }

```

```

3827     return result, nil
3828 }
3829
3830 type AstIdentifier struct {
3831     Location *SourceLocation // Optional
3832     Name     *String                // Cached
3833 }
3834
3835 func (self AstIdentifier) IntoValue(ctx *Context) Value {
3836     return ctx.NewMapOrPanic([]MapPair{
3837         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
3838         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
3839         {ctx.NewString("name"), self.Name.Copy()},
3840     })
3841 }
3842
3843 type AstExpressionIdentifier struct {
3844     Location *SourceLocation // Optional
3845     Name     *String                // Cached
3846 }
3847
3848 func (self AstExpressionIdentifier) ExpressionLocation() *SourceLocation {
3849     return self.Location
3850 }
3851
3852 func (self AstExpressionIdentifier) IntoValue(ctx *Context) Value {
3853     return ctx.NewMapOrPanic([]MapPair{
3854         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
3855         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
3856         {ctx.NewString("name"), self.Name.Copy()},
3857     })
3858 }
3859
3860 func (self *AstExpressionIdentifier) Eval(ctx *Context, env *Environment) (Value, error) {
3861     ↵ {
3862         value, err := env.Get(self.Name.data)
3863         if err != nil {
3864             return nil, NewError(self.Location, ctx.NewString(err.Error()))
3865         }
3866         return value, err
3867     }
3868
3869 type AstExpressionTemplate struct {
3870     Location *SourceLocation // Optional
3871     Template []AstExpression
3872 }
3873
3874 func (self AstExpressionTemplate) ExpressionLocation() *SourceLocation {
3875     return self.Location
3876 }
3877
3878 func (self AstExpressionTemplate) IntoValue(ctx *Context) Value {
3879     elements := []Value{}
3880     for _, element := range self.Template {
3881         elements = append(elements, element.IntoValue(ctx))
3882     }
3883     return ctx.NewMapOrPanic([]MapPair{
3884         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
3885         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
3886         {ctx.NewString("template"), ctx.NewVectorOrPanic(elements)},
3887     })
3888 }
3889
3890 func (self *AstExpressionTemplate) Eval(ctx *Context, env *Environment) (Value, error) {
3891     output := []byte{}
3892     for _, element := range self.Template {
3893         value, err := element.Eval(ctx, env)

```

```

3893         if err != nil {
3894             return nil, err
3895         }
3896
3897         if metaFunction, ok := MetaFunction(ctx, value, ctx.constStringIntoString); ok {
3898             result, err := ctx.callMetaWithSelf(element.ExpressionLocation(),
3899 ↪ metaFunction, value)
3900             if err != nil {
3901                 return nil, err
3902             }
3903             resultString, ok := result.(*String)
3904             if !ok {
3905                 return nil, NewError(
3906                     element.ExpressionLocation(),
3907                     ctx.NewStringf("metafunction %s returned %v",
3908 ↪ quote(ctx.constStringIntoString.data), result),
3909                     )
3910             }
3911             output = append(output, resultString.data...)
3912             continue
3913         }
3914
3915         if s, ok := value.(*String); ok {
3916             output = append(output, s.data...)
3917             continue
3918         }
3919
3920         output = append(output, []byte(value.String())...)
3921     }
3922     return ctx.NewString(string(output)), nil
3923 }
3924
3925 type AstExpressionNull struct {
3926     Location *SourceLocation // Optional
3927 }
3928
3929 func (self AstExpressionNull) ExpressionLocation() *SourceLocation {
3930     return self.Location
3931 }
3932
3933 func (self AstExpressionNull) IntoValue(ctx *Context) Value {
3934     return ctx.NewMapOrPanic([]MapPair{
3935         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
3936         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
3937     })
3938 }
3939
3940 func (self *AstExpressionNull) Eval(ctx *Context, env *Environment) (Value, error) {
3941     return ctx.Null.Copy(), nil
3942 }
3943
3944 type AstExpressionBoolean struct {
3945     Location *SourceLocation // Optional
3946     Data     *Boolean
3947 }
3948
3949 func (self AstExpressionBoolean) ExpressionLocation() *SourceLocation {
3950     return self.Location
3951 }
3952
3953 func (self AstExpressionBoolean) IntoValue(ctx *Context) Value {
3954     return ctx.NewMapOrPanic([]MapPair{
3955         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
3956         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
3957         {ctx.NewString("data"), self.Data.Copy()},
3958     })
3959 }

```

```

3958
3959 func (self *AstExpressionBoolean) Eval(ctx *Context, env *Environment) (Value, error) {
3960     return self.Data.Copy(), nil
3961 }
3962
3963 type AstExpressionNumber struct {
3964     Location *SourceLocation // Optional
3965     Data     *Number
3966 }
3967
3968 func (self AstExpressionNumber) ExpressionLocation() *SourceLocation {
3969     return self.Location
3970 }
3971
3972 func (self AstExpressionNumber) IntoValue(ctx *Context) Value {
3973     return ctx.NewMapOrPanic([]MapPair{
3974         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
3975         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
3976         {ctx.NewString("data"), self.Data.Copy()},
3977     })
3978 }
3979
3980 func (self *AstExpressionNumber) Eval(ctx *Context, env *Environment) (Value, error) {
3981     return self.Data.Copy(), nil
3982 }
3983
3984 type AstExpressionString struct {
3985     Location *SourceLocation // Optional
3986     Data     *String
3987 }
3988
3989 func (self AstExpressionString) ExpressionLocation() *SourceLocation {
3990     return self.Location
3991 }
3992
3993 func (self AstExpressionString) IntoValue(ctx *Context) Value {
3994     return ctx.NewMapOrPanic([]MapPair{
3995         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
3996         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
3997         {ctx.NewString("data"), self.Data.Copy()},
3998     })
3999 }
4000
4001 func (self *AstExpressionString) Eval(ctx *Context, env *Environment) (Value, error) {
4002     return self.Data.Copy(), nil
4003 }
4004
4005 type AstExpressionRegexp struct {
4006     Location *SourceLocation // Optional
4007     Data     *Regexp
4008 }
4009
4010 func (self AstExpressionRegexp) ExpressionLocation() *SourceLocation {
4011     return self.Location
4012 }
4013
4014 func (self AstExpressionRegexp) IntoValue(ctx *Context) Value {
4015     return ctx.NewMapOrPanic([]MapPair{
4016         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4017         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4018         {ctx.NewString("data"), ctx.NewString(self.Data.String())},
4019     })
4020 }
4021
4022 func (self *AstExpressionRegexp) Eval(ctx *Context, env *Environment) (Value, error) {
4023     return self.Data.Copy(), nil
4024 }

```

```

4025
4026 type AstExpressionRegexpGroup struct {
4027     Location *SourceLocation // Optional
4028     Group    int
4029 }
4030
4031 func (self AstExpressionRegexpGroup) ExpressionLocation() *SourceLocation {
4032     return self.Location
4033 }
4034
4035 func (self AstExpressionRegexpGroup) IntoValue(ctx *Context) Value {
4036     return ctx.NewMapOrPanic([]MapPair{
4037         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4038         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4039         {ctx.NewString("group"), ctx.NewNumber(float64(self.Group))},
4040     })
4041 }
4042
4043 func (self *AstExpressionRegexpGroup) Eval(ctx *Context, env *Environment) (Value, error)
↳ {
4044     match, ok := env.getRegexpMatch()
4045     if !ok {
4046         return nil, NewError(nil, ctx.NewString("regular expression did not match"))
4047     }
4048
4049     bgnIndex := (2 * self.Group)
4050     endIndex := (2 * self.Group) + 2
4051     if bgnIndex < 0 || endIndex > len(match.result) {
4052         return nil, NewError(nil, ctx.NewStringf("out-of-bounds regular expression
↳ capture group %v", self.Group))
4053     }
4054
4055     groupBgnEnd := match.result[bgnIndex:endIndex]
4056     if groupBgnEnd[0] == -1 {
4057         return ctx.NewNull(), nil
4058     }
4059
4060     return ctx.NewString(match.string[groupBgnEnd[0]:groupBgnEnd[1]]), nil
4061 }
4062
4063 type AstExpressionVector struct {
4064     Location *SourceLocation // Optional
4065     Elements []AstExpression
4066 }
4067
4068 func (self AstExpressionVector) ExpressionLocation() *SourceLocation {
4069     return self.Location
4070 }
4071
4072 func (self AstExpressionVector) IntoValue(ctx *Context) Value {
4073     elements := []Value{}
4074     for _, element := range self.Elements {
4075         elements = append(elements, element.IntoValue(ctx))
4076     }
4077     return ctx.NewMapOrPanic([]MapPair{
4078         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4079         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4080         {ctx.NewString("elements"), ctx.NewVectorOrPanic(elements)},
4081     })
4082 }
4083
4084 func (self *AstExpressionVector) Eval(ctx *Context, env *Environment) (Value, error) {
4085     elements := []Value{}
4086     for _, element := range self.Elements {
4087         value, err := element.Eval(ctx, env)
4088         if err != nil {
4089             return nil, err

```

```

4090     }
4091     elements = append(elements, moveOrCopy(value))
4092 }
4093 value, err := ctx.NewVector(elements)
4094 if err != nil {
4095     return nil, NewError(
4096         self.Location,
4097         ctx.NewString(err.Error()),
4098     )
4099 }
4100 return value, nil
4101 }
4102
4103 type AstMapPair struct {
4104     Key   AstExpression
4105     Value AstExpression
4106 }
4107
4108 type AstExpressionMap struct {
4109     Location *SourceLocation // Optional
4110     Elements []AstMapPair
4111 }
4112
4113 func (self AstExpressionMap) ExpressionLocation() *SourceLocation {
4114     return self.Location
4115 }
4116
4117 func (self AstExpressionMap) IntoValue(ctx *Context) Value {
4118     elements := []Value{}
4119     for _, element := range self.Elements {
4120         elements = append(elements,
4121             ctx.NewVectorOrPanic([]Value{element.Key.IntoValue(ctx),
4122 ↪ element.Value.IntoValue(ctx)}))
4123     }
4124     return ctx.NewMapOrPanic([]MapPair{
4125         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4126         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4127         {ctx.NewString("elements"), ctx.NewVectorOrPanic(elements)},
4128     })
4129 }
4130
4131 func (self *AstExpressionMap) Eval(ctx *Context, env *Environment) (Value, error) {
4132     pairs := []MapPair{}
4133     for _, element := range self.Elements {
4134         k, err := element.Key.Eval(ctx, env)
4135         if err != nil {
4136             return nil, err
4137         }
4138         v, err := element.Value.Eval(ctx, env)
4139         if err != nil {
4140             return nil, err
4141         }
4142         pairs = append(pairs, MapPair{moveOrCopy(k), moveOrCopy(v)})
4143     }
4144     value, err := ctx.NewMap(pairs)
4145     if err != nil {
4146         return nil, NewError(
4147             self.Location,
4148             ctx.NewString(err.Error()),
4149         )
4150     }
4151     return value, nil
4152 }
4153
4154 type AstExpressionSet struct {
4155     Location *SourceLocation // Optional
4156     Elements []AstExpression

```

```

4156 }
4157
4158 func (self AstExpressionSet) ExpressionLocation() *SourceLocation {
4159     return self.Location
4160 }
4161
4162 func (self AstExpressionSet) IntoValue(ctx *Context) Value {
4163     elements := []Value{}
4164     for _, element := range self.Elements {
4165         elements = append(elements, element.IntoValue(ctx))
4166     }
4167     return ctx.NewMapOrPanic([]MapPair{
4168         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4169         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4170         {ctx.NewString("elements"), ctx.NewVectorOrPanic(elements)},
4171     })
4172 }
4173
4174 func (self *AstExpressionSet) Eval(ctx *Context, env *Environment) (Value, error) {
4175     elements := []Value{}
4176     for _, element := range self.Elements {
4177         value, err := element.Eval(ctx, env)
4178         if err != nil {
4179             return nil, err
4180         }
4181         elements = append(elements, moveOrCopy(value))
4182     }
4183     result, err := ctx.NewSet(elements)
4184     if err != nil {
4185         return nil, NewError(
4186             self.Location,
4187             ctx.NewString(err.Error()),
4188         )
4189     }
4190     return result, nil
4191 }
4192
4193
4194 type AstExpressionFunction struct {
4195     Location      *SourceLocation // Optional
4196     Parameters    []*AstIdentifier
4197     Body          *AstBlock
4198     Name          *String // Optional
4199     SelfByReference bool
4200 }
4201
4202 func (self AstExpressionFunction) ExpressionLocation() *SourceLocation {
4203     return self.Location
4204 }
4205
4206 func (self AstExpressionFunction) IntoValue(ctx *Context) Value {
4207     parameters := ctx.NewVectorOrPanic(nil)
4208     for _, parameter := range self.Parameters {
4209         _ = parameters.Push(parameter.IntoValue(ctx))
4210     }
4211     var name Value = ctx.NewNull()
4212     if self.Name != nil {
4213         name = self.Name.Copy()
4214     }
4215     return ctx.NewMapOrPanic([]MapPair{
4216         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4217         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4218         {ctx.NewString("parameters"), parameters},
4219         {ctx.NewString("body"), self.Body.IntoValue(ctx)},
4220         {ctx.NewString("name"), name},
4221         {ctx.NewString("self_by_reference"), ctx.NewBoolean(self.SelfByReference)},
4222     })

```

```

4223 }
4224
4225 func (self *AstExpressionFunction) Eval(ctx *Context, env *Environment) (Value, error) {
4226     if env.nref > 0 {
4227         elements := []Value{}
4228         for env != nil {
4229             for _, info := range env.refs {
4230                 elements = append(elements, ctx.NewMapOrPanic([]MapPair{
4231                     {ctx.NewString("name"), ctx.NewString(info.name)},
4232                     {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx,
4233 ↪ info.location)}},
4234                     )))
4235                 env = env.outer
4236             }
4237             slices.Reverse(elements)
4238             return nil, NewError(
4239                 self.Location,
4240                 ctx.NewMapOrPanic([]MapPair{
4241                     {ctx.NewString("message"), ctx.NewString("function closes over a
4242 ↪ reference")},
4243                     {ctx.NewString("references"), ctx.NewVectorOrPanic(elements)}},
4244                     ))
4245             }
4246             env.markHeld()
4247             return ctx.newFunction(self, env), nil
4248         }
4249
4250     type AstExpressionFreeze struct {
4251         Location *SourceLocation // Optional
4252         Expression AstExpression
4253     }
4254
4255     func (self AstExpressionFreeze) ExpressionLocation() *SourceLocation {
4256         return self.Location
4257     }
4258
4259     func (self AstExpressionFreeze) IntoValue(ctx *Context) Value {
4260         return ctx.NewMapOrPanic([]MapPair{
4261             {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4262             {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4263             {ctx.NewString("expression"), self.Expression.IntoValue(ctx)},
4264         })
4265     }
4266
4267     func (self *AstExpressionFreeze) Eval(ctx *Context, env *Environment) (Value, error) {
4268         value, err := self.Expression.Eval(ctx, env)
4269         if err != nil {
4270             return nil, err
4271         }
4272
4273         return value.Freeze(), nil
4274     }
4275
4276     type AstExpressionType struct {
4277         Location *SourceLocation // Optional
4278         Name string
4279         Extends AstExpression // Optional
4280         Expression AstExpression
4281     }
4282
4283     func (self AstExpressionType) ExpressionLocation() *SourceLocation {
4284         return self.Location
4285     }
4286
4287     func (self AstExpressionType) IntoValue(ctx *Context) Value {

```

```

4288     var extends Value = ctx.NewNull()
4289     if self.Extends != nil {
4290         extends = self.Extends.IntoValue(ctx)
4291     }
4292     return ctx.NewMapOrPanic([]MapPair{
4293         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4294         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4295         {ctx.NewString("name"), ctx.NewString(self.Name)},
4296         {ctx.NewString("extends"), extends},
4297         {ctx.NewString("expression"), self.Expression.IntoValue(ctx)},
4298     })
4299 }
4300
4301 func (self *AstExpressionType) Eval(ctx *Context, env *Environment) (Value, error) {
4302     var extendsMap *Map = nil
4303     if self.Extends != nil {
4304         extends, err := self.Extends.Eval(ctx, env)
4305         if err != nil {
4306             return nil, err
4307         }
4308
4309         var ok bool
4310         extendsMap, ok = extends.(*Map)
4311         if !(ok && extendsMap.IsImmutable() && extendsMap.name != nil) {
4312             return nil, NewError(
4313                 self.Extends.ExpressionLocation(),
4314                 ctx.NewStringf("expected map value created with the %s keyword, received
↳ %v", quote(TOKEN_TYPE), extends),
4315             )
4316         }
4317     }
4318
4319     value, err := self.Expression.Eval(ctx, env)
4320     if err != nil {
4321         return nil, err
4322     }
4323
4324     valueMap, ok := value.(*Map)
4325     if !ok {
4326         return nil, NewError(
4327             self.Location,
4328             ctx.NewStringf("expected map-like value, received %s", value.TypeName()),
4329         )
4330     }
4331
4332     var result *Map = nil
4333     if extendsMap != nil {
4334         // A new map is created so that all of the key-value pairs from the
4335         // super-type, then all of the key-value pairs from the sub-type, can
4336         // be added to the resulting type in that order, preserving the order
4337         // of key-value pairs to make it seem as if the sub-type is layered on
4338         // top of the super-type with map::union.
4339         //
4340         // The super-type map is immutable, so we can directly use pairs from
4341         // the super-type map to construct the extended map.
4342         result, err = ctx.NewMap(extendsMap.Pairs())
4343         if err != nil {
4344             return nil, NewError(
4345                 self.Location,
4346                 ctx.NewString(err.Error()),
4347             )
4348         }
4349         for _, pair := range valueMap.Pairs() {
4350             err := result.Insert(pair.Key.Copy(), pair.Value.Copy())
4351             if err != nil {
4352                 return nil, NewError(
4353                     self.Location,

```

```

4354         ctx.NewString(err.Error()),
4355     )
4356 }
4357 }
4358 } else {
4359     // A new map is created and filled with copies of each key-value pair
4360     // from the evaluated map expression. Creation of a new map ensures
4361     // that assigning a name to the result of a type expression will not
4362     // overwrite the name of an already frozen map.
4363     result, err = ctx.NewMap(nil)
4364     if err != nil {
4365         return nil, NewError(
4366             self.Location,
4367             ctx.NewString(err.Error()),
4368         )
4369     }
4370     for _, pair := range valueMap.Pairs() {
4371         err := result.Insert(pair.Key.Copy(), pair.Value.Copy())
4372         if err != nil {
4373             return nil, NewError(
4374                 self.Location,
4375                 ctx.NewString(err.Error()),
4376             )
4377         }
4378     }
4379 }
4380
4381 result = result.Freeze().(*Map)
4382 result.name = &self.Name
4383 return result, nil
4384 }
4385
4386 type AstExpressionNew struct {
4387     Location    *SourceLocation // Optional
4388     Meta        AstExpression
4389     Expression  AstExpression
4390 }
4391
4392 func (self AstExpressionNew) ExpressionLocation() *SourceLocation {
4393     return self.Location
4394 }
4395
4396 func (self AstExpressionNew) IntoValue(ctx *Context) Value {
4397     return ctx.NewMapOrPanic([]MapPair{
4398         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4399         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4400         {ctx.NewString("meta"), self.Meta.IntoValue(ctx)},
4401         {ctx.NewString("expression"), self.Expression.IntoValue(ctx)},
4402     })
4403 }
4404
4405 func (self *AstExpressionNew) Eval(ctx *Context, env *Environment) (Value, error) {
4406     meta, err := self.Meta.Eval(ctx, env)
4407     if err != nil {
4408         return nil, err
4409     }
4410
4411     value, err := self.Expression.Eval(ctx, env)
4412     if err != nil {
4413         return nil, err
4414     }
4415
4416     metaMap, ok := meta.(*Map)
4417     if !ok {
4418         return nil, NewError(
4419             self.Location,
4420             ctx.NewStringf("expected map-like value, received %s", meta.TypeName()),

```

```

4421     )
4422 }
4423
4424 valueMap, ok := value.(*Map)
4425 if !ok {
4426     return nil, NewError(
4427         self.Location,
4428         ctx.NewStringf("expected map-like value, received %s", value.TypeName()),
4429     )
4430 }
4431
4432 if metaMap.name == nil {
4433     return nil, NewError(
4434         self.Location,
4435         ctx.NewStringf("expected map-like value created with the %s expression,
↪ received regular map value %v", quote(TOKEN_TYPE), meta),
4436     )
4437 }
4438
4439 // A new map is created and filled with copies of each key-value pair from
4440 // the evaluated map expression. Creation of a new map ensures that a new
4441 // expression applied to a frozen map does not silently change the existing
4442 // metamap of that frozen map.
4443 result, err := ctx.NewMap(nil)
4444 if err != nil {
4445     return nil, NewError(
4446         self.Location,
4447         ctx.NewString(err.Error()),
4448     )
4449 }
4450 for _, pair := range valueMap.Pairs() {
4451     err := result.Insert(pair.Key.Copy(), pair.Value.Copy())
4452     if err != nil {
4453         return nil, NewError(
4454             self.Location,
4455             ctx.NewString(err.Error()),
4456         )
4457     }
4458 }
4459 result.meta = metaMap.Copy().(*Map)
4460 return result, nil
4461 }
4462
4463 type AstExpressionGrouped struct {
4464     Location *SourceLocation // Optional
4465     Expression AstExpression
4466 }
4467
4468 func (self AstExpressionGrouped) ExpressionLocation() *SourceLocation {
4469     return self.Location
4470 }
4471
4472 func (self AstExpressionGrouped) IntoValue(ctx *Context) Value {
4473     return ctx.NewMapOrPanic([]MapPair{
4474         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4475         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4476         {ctx.NewString("expression"), self.Expression.IntoValue(ctx)},
4477     })
4478 }
4479
4480 func (self *AstExpressionGrouped) Eval(ctx *Context, env *Environment) (Value, error) {
4481     return self.Expression.Eval(ctx, env)
4482 }
4483
4484 type AstExpressionPositive struct {
4485     Location *SourceLocation // Optional
4486     Expression AstExpression

```

```

4487 }
4488
4489 func (self AstExpressionPositive) ExpressionLocation() *SourceLocation {
4490     return self.Location
4491 }
4492
4493 func (self AstExpressionPositive) IntoValue(ctx *Context) Value {
4494     return ctx.NewMapOrPanic([]MapPair{
4495         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4496         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4497         {ctx.NewString("expression"), self.Expression.IntoValue(ctx)},
4498     })
4499 }
4500
4501 func (self *AstExpressionPositive) Eval(ctx *Context, env *Environment) (Value, error) {
4502     value, err := self.Expression.Eval(ctx, env)
4503     if err != nil {
4504         return nil, err
4505     }
4506
4507     if number, ok := value.(*Number); ok {
4508         return ctx.NewNumber(+number.data), nil
4509     }
4510
4511     return nil, NewError(
4512         self.Location,
4513         ctx.NewStringf("attempted unary + operation with type %s",
4514             ↪ quote(value.TypeName())),
4515     )
4516 }
4517
4518 type AstExpressionNegative struct {
4519     Location *SourceLocation // Optional
4520     Expression AstExpression
4521 }
4522
4523 func (self AstExpressionNegative) ExpressionLocation() *SourceLocation {
4524     return self.Location
4525 }
4526
4527 func (self AstExpressionNegative) IntoValue(ctx *Context) Value {
4528     return ctx.NewMapOrPanic([]MapPair{
4529         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4530         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4531         {ctx.NewString("expression"), self.Expression.IntoValue(ctx)},
4532     })
4533 }
4534
4535 func (self *AstExpressionNegative) Eval(ctx *Context, env *Environment) (Value, error) {
4536     value, err := self.Expression.Eval(ctx, env)
4537     if err != nil {
4538         return nil, err
4539     }
4540
4541     if number, ok := value.(*Number); ok {
4542         return ctx.NewNumber(-number.data), nil
4543     }
4544
4545     return nil, NewError(
4546         self.Location,
4547         ctx.NewStringf("attempted unary - operation with type %s",
4548             ↪ quote(value.TypeName())),
4549     )
4550 }
4551
4552 type AstExpressionNot struct {
4553     Location *SourceLocation // Optional

```

```

4552     Expression AstExpression
4553 }
4554
4555 func (self AstExpressionNot) ExpressionLocation() *SourceLocation {
4556     return self.Location
4557 }
4558
4559 func (self AstExpressionNot) IntoValue(ctx *Context) Value {
4560     return ctx.NewMapOrPanic([]MapPair{
4561         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4562         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4563         {ctx.NewString("expression"), self.Expression.IntoValue(ctx)},
4564     })
4565 }
4566
4567 func (self *AstExpressionNot) Eval(ctx *Context, env *Environment) (Value, error) {
4568     value, err := self.Expression.Eval(ctx, env)
4569     if err != nil {
4570         return nil, err
4571     }
4572
4573     if boolean, ok := value.(*Boolean); ok {
4574         return ctx.NewBoolean(!boolean.data), nil
4575     }
4576
4577     return nil, NewError(
4578         self.Location,
4579         ctx.NewStringf("attempted unary not operation with type %s",
4580 ↪ quote(value.TypeName()))),
4581 )
4582 }
4583
4584 type AstExpressionAnd struct {
4585     Location *SourceLocation // Optional
4586     Lhs      AstExpression
4587     Rh      AstExpression
4588 }
4589
4590 func (self AstExpressionAnd) ExpressionLocation() *SourceLocation {
4591     return self.Location
4592 }
4593
4594 func (self AstExpressionAnd) IntoValue(ctx *Context) Value {
4595     return ctx.NewMapOrPanic([]MapPair{
4596         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4597         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4598         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
4599         {ctx.NewString("rhs"), self.Rhs.IntoValue(ctx)},
4600     })
4601 }
4602
4603 func (self *AstExpressionAnd) Eval(ctx *Context, env *Environment) (Value, error) {
4604     lhs, err := self.Lhs.Eval(ctx, env)
4605     if err != nil {
4606         return nil, err
4607     }
4608
4609     lhsBoolean, lhsIsBoolean := lhs.(*Boolean)
4610     if !lhsIsBoolean {
4611         return nil, NewError(
4612             self.Location,
4613             ctx.NewStringf("attempted binary and operation with left-hand-side of type
4614 ↪ %s", quote(lhs.TypeName()))),
4615     }
4616     if !lhsBoolean.data {
4617         return ctx.NewBoolean(false), nil // short circuit

```

```

4617     }
4618
4619     rhs, err := self.Rhs.Eval(ctx, env)
4620     if err != nil {
4621         return nil, err
4622     }
4623
4624     rhsBoolean, rhsIsBoolean := rhs.(*Boolean)
4625     if !rhsIsBoolean {
4626         return nil, NewError(
4627             self.Location,
4628             ctx.NewStringf("attempted binary and operation with right-hand-side of type
↳ %s", quote(rhs.TypeName()))),
4629         )
4630     }
4631     if !rhsBoolean.data {
4632         return ctx.NewBoolean(false), nil
4633     }
4634
4635     return ctx.NewBoolean(true), nil
4636 }
4637
4638 type AstExpressionOr struct {
4639     Location *SourceLocation // Optional
4640     Lhs      AstExpression
4641     Rhs      AstExpression
4642 }
4643
4644 func (self AstExpressionOr) ExpressionLocation() *SourceLocation {
4645     return self.Location
4646 }
4647
4648 func (self AstExpressionOr) IntoValue(ctx *Context) Value {
4649     return ctx.NewMapOrPanic([]MapPair{
4650         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4651         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4652         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
4653         {ctx.NewString("rhs"), self.Rhs.IntoValue(ctx)},
4654     })
4655 }
4656
4657 func (self *AstExpressionOr) Eval(ctx *Context, env *Environment) (Value, error) {
4658     lhs, err := self.Lhs.Eval(ctx, env)
4659     if err != nil {
4660         return nil, err
4661     }
4662
4663     lhsBoolean, lhsIsBoolean := lhs.(*Boolean)
4664     if !lhsIsBoolean {
4665         return nil, NewError(
4666             self.Location,
4667             ctx.NewStringf("attempted binary or operation with left-hand-side of type
↳ %s", quote(lhs.TypeName()))),
4668         )
4669     }
4670     if lhsBoolean.data {
4671         return ctx.NewBoolean(true), nil // short circuit
4672     }
4673
4674     rhs, err := self.Rhs.Eval(ctx, env)
4675     if err != nil {
4676         return nil, err
4677     }
4678
4679     rhsBoolean, rhsIsBoolean := rhs.(*Boolean)
4680     if !rhsIsBoolean {
4681         return nil, NewError(

```

```

4682         self.Location,
4683         ctx.NewStringf("attempted binary or operation with right-hand-side of type
↪ %s", quote(rhs.TypeName()))),
4684     )
4685 }
4686 if rhsBoolean.data {
4687     return ctx.NewBoolean(true), nil
4688 }
4689
4690 return ctx.NewBoolean(false), nil
4691 }
4692
4693 type AstExpressionEq struct {
4694     Location *SourceLocation // Optional
4695     Lhs      AstExpression
4696     Rh      AstExpression
4697 }
4698
4699 func (self AstExpressionEq) ExpressionLocation() *SourceLocation {
4700     return self.Location
4701 }
4702
4703 func (self AstExpressionEq) IntoValue(ctx *Context) Value {
4704     return ctx.NewMapOrPanic([]MapPair{
4705         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4706         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4707         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
4708         {ctx.NewString("rhs"), self.Rhs.IntoValue(ctx)},
4709     })
4710 }
4711
4712 func (self *AstExpressionEq) Eval(ctx *Context, env *Environment) (Value, error) {
4713     lhs, err := self.Lhs.Eval(ctx, env)
4714     if err != nil {
4715         return nil, err
4716     }
4717
4718     rhs, err := self.Rhs.Eval(ctx, env)
4719     if err != nil {
4720         return nil, err
4721     }
4722
4723     return ctx.NewBoolean(lhs.Equal(rhs)), nil
4724 }
4725
4726 type AstExpressionNe struct {
4727     Location *SourceLocation // Optional
4728     Lhs      AstExpression
4729     Rh      AstExpression
4730 }
4731
4732 func (self AstExpressionNe) ExpressionLocation() *SourceLocation {
4733     return self.Location
4734 }
4735
4736 func (self AstExpressionNe) IntoValue(ctx *Context) Value {
4737     return ctx.NewMapOrPanic([]MapPair{
4738         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4739         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4740         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
4741         {ctx.NewString("rhs"), self.Rhs.IntoValue(ctx)},
4742     })
4743 }
4744
4745 func (self *AstExpressionNe) Eval(ctx *Context, env *Environment) (Value, error) {
4746     lhs, err := self.Lhs.Eval(ctx, env)
4747     if err != nil {

```

```

4748         return nil, err
4749     }
4750
4751     rhs, err := self.Rhs.Eval(ctx, env)
4752     if err != nil {
4753         return nil, err
4754     }
4755
4756     return ctx.NewBoolean(!lhs.Equal(rhs)), nil
4757 }
4758
4759 type AstExpressionLe struct {
4760     Location *SourceLocation // Optional
4761     Lhs      AstExpression
4762     Rhs      AstExpression
4763 }
4764
4765 func (self AstExpressionLe) ExpressionLocation() *SourceLocation {
4766     return self.Location
4767 }
4768
4769 func (self AstExpressionLe) IntoValue(ctx *Context) Value {
4770     return ctx.NewMapOrPanic([]MapPair{
4771         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4772         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4773         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
4774         {ctx.NewString("rhs"), self.Rhs.IntoValue(ctx)},
4775     })
4776 }
4777
4778 func (self *AstExpressionLe) Eval(ctx *Context, env *Environment) (Value, error) {
4779     lhs, err := self.Lhs.Eval(ctx, env)
4780     if err != nil {
4781         return nil, err
4782     }
4783
4784     rhs, err := self.Rhs.Eval(ctx, env)
4785     if err != nil {
4786         return nil, err
4787     }
4788
4789     lhsNumber, lhsIsNumber := lhs.(*Number)
4790     if lhsIsNumber {
4791         rhsNumber, rhsIsNumber := rhs.(*Number)
4792         if rhsIsNumber {
4793             return ctx.NewBoolean(lhsNumber.data <= rhsNumber.data), nil
4794         }
4795     }
4796
4797     lhsString, lhsIsString := lhs.(*String)
4798     if lhsIsString {
4799         rhsString, rhsIsString := rhs.(*String)
4800         if rhsIsString {
4801             return ctx.NewBoolean(lhsString.data <= rhsString.data), nil
4802         }
4803     }
4804
4805     return nil, NewError(
4806         self.Location,
4807         ctx.NewStringf("attempted <= operation with types %s and %s",
4808             ↵ quote(lhs.TypeName()), quote(rhs.TypeName()))),
4809     )
4810 }
4811
4812 type AstExpressionGe struct {
4813     Location *SourceLocation // Optional
4814     Lhs      AstExpression

```

```

4814     Rhs      AstExpression
4815 }
4816
4817 func (self AstExpressionGe) ExpressionLocation() *SourceLocation {
4818     return self.Location
4819 }
4820
4821 func (self AstExpressionGe) IntoValue(ctx *Context) Value {
4822     return ctx.NewMapOrPanic([]MapPair{
4823         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4824         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4825         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
4826         {ctx.NewString("rhs"), self.Rhs.IntoValue(ctx)},
4827     })
4828 }
4829
4830 func (self *AstExpressionGe) Eval(ctx *Context, env *Environment) (Value, error) {
4831     lhs, err := self.Lhs.Eval(ctx, env)
4832     if err != nil {
4833         return nil, err
4834     }
4835
4836     rhs, err := self.Rhs.Eval(ctx, env)
4837     if err != nil {
4838         return nil, err
4839     }
4840
4841     lhsNumber, lhsIsNumber := lhs.(*Number)
4842     if lhsIsNumber {
4843         rhsNumber, rhsIsNumber := rhs.(*Number)
4844         if rhsIsNumber {
4845             return ctx.NewBoolean(lhsNumber.data >= rhsNumber.data), nil
4846         }
4847     }
4848
4849     lhsString, lhsIsString := lhs.(*String)
4850     if lhsIsString {
4851         rhsString, rhsIsString := rhs.(*String)
4852         if rhsIsString {
4853             return ctx.NewBoolean(lhsString.data >= rhsString.data), nil
4854         }
4855     }
4856
4857     return nil, NewError(
4858         self.Location,
4859         ctx.NewStringf("attempted >= operation with types %s and %s",
4860             ↪ quote(lhs.TypeName()), quote(rhs.TypeName())),
4861     )
4862 }
4863
4864 type AstExpressionLt struct {
4865     Location *SourceLocation // Optional
4866     Lhs      AstExpression
4867     Rhs      AstExpression
4868 }
4869
4870 func (self AstExpressionLt) ExpressionLocation() *SourceLocation {
4871     return self.Location
4872 }
4873
4874 func (self AstExpressionLt) IntoValue(ctx *Context) Value {
4875     return ctx.NewMapOrPanic([]MapPair{
4876         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4877         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4878         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
4879         {ctx.NewString("rhs"), self.Rhs.IntoValue(ctx)},
4880     })
4881 }

```

```

4880 }
4881
4882 func (self *AstExpressionLt) Eval(ctx *Context, env *Environment) (Value, error) {
4883     lhs, err := self.Lhs.Eval(ctx, env)
4884     if err != nil {
4885         return nil, err
4886     }
4887
4888     rhs, err := self.Rhs.Eval(ctx, env)
4889     if err != nil {
4890         return nil, err
4891     }
4892
4893     lhsNumber, lhsIsNumber := lhs.(*Number)
4894     if lhsIsNumber {
4895         rhsNumber, rhsIsNumber := rhs.(*Number)
4896         if rhsIsNumber {
4897             return ctx.NewBoolean(lhsNumber.data < rhsNumber.data), nil
4898         }
4899     }
4900
4901     lhsString, lhsIsString := lhs.(*String)
4902     if lhsIsString {
4903         rhsString, rhsIsString := rhs.(*String)
4904         if rhsIsString {
4905             return ctx.NewBoolean(lhsString.data < rhsString.data), nil
4906         }
4907     }
4908
4909     return nil, NewError(
4910         self.Location,
4911         ctx.NewStringf("attempted < operation with types %s and %s",
↪      quote(lhs.TypeName()), quote(rhs.TypeName()))
4912     )
4913 }
4914
4915 type AstExpressionGt struct {
4916     Location *SourceLocation // Optional
4917     Lhs      AstExpression
4918     Rhs      AstExpression
4919 }
4920
4921 func (self AstExpressionGt) ExpressionLocation() *SourceLocation {
4922     return self.Location
4923 }
4924
4925 func (self AstExpressionGt) IntoValue(ctx *Context) Value {
4926     return ctx.NewMapOrPanic([]MapPair{
4927         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4928         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4929         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
4930         {ctx.NewString("rhs"), self.Rhs.IntoValue(ctx)},
4931     })
4932 }
4933
4934 func (self *AstExpressionGt) Eval(ctx *Context, env *Environment) (Value, error) {
4935     lhs, err := self.Lhs.Eval(ctx, env)
4936     if err != nil {
4937         return nil, err
4938     }
4939
4940     rhs, err := self.Rhs.Eval(ctx, env)
4941     if err != nil {
4942         return nil, err
4943     }
4944
4945     lhsNumber, lhsIsNumber := lhs.(*Number)

```

```

4946     if lhsIsNumber {
4947         rhsNumber, rhsIsNumber := rhs.(*Number)
4948         if rhsIsNumber {
4949             return ctx.NewBoolean(lhsNumber.data > rhsNumber.data), nil
4950         }
4951     }
4952
4953     lhsString, lhsIsString := lhs.(*String)
4954     if lhsIsString {
4955         rhsString, rhsIsString := rhs.(*String)
4956         if rhsIsString {
4957             return ctx.NewBoolean(lhsString.data > rhsString.data), nil
4958         }
4959     }
4960
4961     return nil, NewError(
4962         self.Location,
4963         ctx.NewStringf("attempted > operation with types %s and %s",
↪ quote(lhs.TypeName()), quote(rhs.TypeName()))),
4964     )
4965 }
4966
4967 type AstExpressionEqRe struct {
4968     Location *SourceLocation // Optional
4969     Lhs      AstExpression
4970     Rhs      AstExpression
4971 }
4972
4973 func (self AstExpressionEqRe) ExpressionLocation() *SourceLocation {
4974     return self.Location
4975 }
4976
4977 func (self AstExpressionEqRe) IntoValue(ctx *Context) Value {
4978     return ctx.NewMapOrPanic([]MapPair{
4979         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
4980         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
4981         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
4982         {ctx.NewString("rhs"), self.Rhs.IntoValue(ctx)},
4983     })
4984 }
4985
4986 func (self *AstExpressionEqRe) Eval(ctx *Context, env *Environment) (Value, error) {
4987     lhs, err := self.Lhs.Eval(ctx, env)
4988     if err != nil {
4989         return nil, err
4990     }
4991
4992     rhs, err := self.Rhs.Eval(ctx, env)
4993     if err != nil {
4994         return nil, err
4995     }
4996
4997     lhsString, lhsIsString := lhs.(*String)
4998     if lhsIsString {
4999         rhsRegexp, rhsIsRegexp := rhs.(*Regexp)
5000         if rhsIsRegexp {
5001             result := rhsRegexp.data.FindStringSubmatchIndex(lhsString.data)
5002             env.setRegexpMatch(regexpMatch{lhsString.data, result})
5003             return ctx.NewBoolean(result != nil), nil
5004         }
5005     }
5006
5007     return nil, NewError(
5008         self.Location,
5009         ctx.NewStringf("attempted =~ operation with types %s and %s",
↪ quote(lhs.TypeName()), quote(rhs.TypeName()))),
5010     )

```

```

5011 }
5012
5013 type AstExpressionNeRe struct {
5014     Location *SourceLocation // Optional
5015     Lhs      AstExpression
5016     Rh      AstExpression
5017 }
5018
5019 func (self AstExpressionNeRe) ExpressionLocation() *SourceLocation {
5020     return self.Location
5021 }
5022
5023 func (self AstExpressionNeRe) IntoValue(ctx *Context) Value {
5024     return ctx.NewMapOrPanic([]MapPair{
5025         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
5026         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5027         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
5028         {ctx.NewString("rhs"), self.Rhs.IntoValue(ctx)},
5029     })
5030 }
5031
5032 func (self *AstExpressionNeRe) Eval(ctx *Context, env *Environment) (Value, error) {
5033     lhs, err := self.Lhs.Eval(ctx, env)
5034     if err != nil {
5035         return nil, err
5036     }
5037
5038     rhs, err := self.Rhs.Eval(ctx, env)
5039     if err != nil {
5040         return nil, err
5041     }
5042
5043     lhsString, lhsIsString := lhs.(*String)
5044     if lhsIsString {
5045         rhsRegexp, rhsIsRegexp := rhs.(*Regexp)
5046         if rhsIsRegexp {
5047             result := rhsRegexp.data.FindStringSubmatchIndex(lhsString.data)
5048             env.setRegexpMatch(regexpMatch{lhsString.data, result})
5049             return ctx.NewBoolean(result == nil), nil
5050         }
5051     }
5052
5053     return nil, NewError(
5054         self.Location,
5055         ctx.NewStringf("attempted !~ operation with types %s and %s",
5056             ↪ quote(lhs.TypeName()), quote(rhs.TypeName()))
5057     )
5058 }
5059
5060 type AstExpressionAdd struct {
5061     Location *SourceLocation // Optional
5062     Lhs      AstExpression
5063     Rh      AstExpression
5064 }
5065
5066 func (self AstExpressionAdd) ExpressionLocation() *SourceLocation {
5067     return self.Location
5068 }
5069
5070 func (self AstExpressionAdd) IntoValue(ctx *Context) Value {
5071     return ctx.NewMapOrPanic([]MapPair{
5072         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
5073         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5074         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
5075         {ctx.NewString("rhs"), self.Rhs.IntoValue(ctx)},
5076     })
5077 }

```

```

5077
5078 func (self *AstExpressionAdd) Eval(ctx *Context, env *Environment) (Value, error) {
5079     lhs, err := self.Lhs.Eval(ctx, env)
5080     if err != nil {
5081         return nil, err
5082     }
5083
5084     rhs, err := self.Rhs.Eval(ctx, env)
5085     if err != nil {
5086         return nil, err
5087     }
5088
5089     lhsNumber, lhsIsNumber := lhs.(*Number)
5090     if lhsIsNumber {
5091         rhsNumber, rhsIsNumber := rhs.(*Number)
5092         if rhsIsNumber {
5093             return ctx.NewNumber(lhsNumber.data + rhsNumber.data), nil
5094         }
5095     }
5096
5097     lhsString, lhsIsString := lhs.(*String)
5098     if lhsIsString {
5099         rhsString, rhsIsString := rhs.(*String)
5100         if rhsIsString {
5101             return ctx.NewString(lhsString.data + rhsString.data), nil
5102         }
5103     }
5104
5105     lhsVector, lhsIsVector := lhs.(*Vector)
5106     if lhsIsVector {
5107         rhsVector, rhsIsVector := rhs.(*Vector)
5108         if rhsIsVector {
5109             elements := []Value{}
5110             for _, element := range lhsVector.Elements() {
5111                 elements = append(elements, element.Copy())
5112             }
5113             for _, element := range rhsVector.Elements() {
5114                 elements = append(elements, element.Copy())
5115             }
5116             return ctx.NewVectorOrPanic(elements), nil
5117         }
5118     }
5119
5120     return nil, NewError(
5121         self.Location,
5122         ctx.NewStringf("attempted + operation with types %s and %s",
5123             ↪ quote(lhs.TypeName()), quote(rhs.TypeName())),
5124     )
5125 }
5126
5127 type AstExpressionSub struct {
5128     Location *SourceLocation // Optional
5129     Lhs      AstExpression
5130     Rhs      AstExpression
5131 }
5132
5133 func (self AstExpressionSub) ExpressionLocation() *SourceLocation {
5134     return self.Location
5135 }
5136
5137 func (self AstExpressionSub) IntoValue(ctx *Context) Value {
5138     return ctx.NewMapOrPanic([]MapPair{
5139         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
5140         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5141         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
5142         {ctx.NewString("rhs"), self.Rhs.IntoValue(ctx)},
5143     })

```

```

5143 }
5144
5145 func (self *AstExpressionSub) Eval(ctx *Context, env *Environment) (Value, error) {
5146     lhs, err := self.Lhs.Eval(ctx, env)
5147     if err != nil {
5148         return nil, err
5149     }
5150
5151     rhs, err := self.Rhs.Eval(ctx, env)
5152     if err != nil {
5153         return nil, err
5154     }
5155
5156     lhsNumber, lhsIsNumber := lhs.(*Number)
5157     if lhsIsNumber {
5158         rhsNumber, rhsIsNumber := rhs.(*Number)
5159         if rhsIsNumber {
5160             return ctx.NewNumber(lhsNumber.data - rhsNumber.data), nil
5161         }
5162     }
5163
5164     return nil, NewError(
5165         self.Location,
5166         ctx.NewStringf("attempted - operation with types %s and %s",
5167             ↪ quote(lhs.TypeName()), quote(rhs.TypeName())),
5168     )
5169 }
5170
5171 type AstExpressionMul struct {
5172     Location *SourceLocation // Optional
5173     Lhs      AstExpression
5174     Rhs      AstExpression
5175 }
5176
5177 func (self AstExpressionMul) ExpressionLocation() *SourceLocation {
5178     return self.Location
5179 }
5180
5181 func (self AstExpressionMul) IntoValue(ctx *Context) Value {
5182     return ctx.NewMapOrPanic([]MapPair{
5183         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
5184         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5185         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
5186         {ctx.NewString("rhs"), self.Rhs.IntoValue(ctx)},
5187     })
5188 }
5189
5190 func (self *AstExpressionMul) Eval(ctx *Context, env *Environment) (Value, error) {
5191     lhs, err := self.Lhs.Eval(ctx, env)
5192     if err != nil {
5193         return nil, err
5194     }
5195
5196     rhs, err := self.Rhs.Eval(ctx, env)
5197     if err != nil {
5198         return nil, err
5199     }
5200
5201     lhsNumber, lhsIsNumber := lhs.(*Number)
5202     if lhsIsNumber {
5203         rhsNumber, rhsIsNumber := rhs.(*Number)
5204         if rhsIsNumber {
5205             return ctx.NewNumber(lhsNumber.data * rhsNumber.data), nil
5206         }
5207     }
5208
5209     return nil, NewError(

```

```

5209         self.Location,
5210         ctx.NewStringf("attempted * operation with types %s and %s",
↪ quote(lhs.TypeName()), quote(rhs.TypeName()))),
5211     )
5212 }
5213
5214 type AstExpressionDiv struct {
5215     Location *SourceLocation // Optional
5216     Lhs      AstExpression
5217     Rh      AstExpression
5218 }
5219
5220 func (self AstExpressionDiv) ExpressionLocation() *SourceLocation {
5221     return self.Location
5222 }
5223
5224 func (self AstExpressionDiv) IntoValue(ctx *Context) Value {
5225     return ctx.NewMapOrPanic([]MapPair{
5226         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
5227         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5228         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
5229         {ctx.NewString("rhs"), self.Rhs.IntoValue(ctx)},
5230     })
5231 }
5232
5233 func (self *AstExpressionDiv) Eval(ctx *Context, env *Environment) (Value, error) {
5234     lhs, err := self.Lhs.Eval(ctx, env)
5235     if err != nil {
5236         return nil, err
5237     }
5238
5239     rhs, err := self.Rhs.Eval(ctx, env)
5240     if err != nil {
5241         return nil, err
5242     }
5243
5244     lhsNumber, lhsIsNumber := lhs.(*Number)
5245     if lhsIsNumber {
5246         rhsNumber, rhsIsNumber := rhs.(*Number)
5247         if rhsIsNumber {
5248             return ctx.NewNumber(lhsNumber.data / rhsNumber.data), nil
5249         }
5250     }
5251
5252     return nil, NewError(
5253         self.Location,
5254         ctx.NewStringf("attempted / operation with types %s and %s",
↪ quote(lhs.TypeName()), quote(rhs.TypeName()))),
5255     )
5256 }
5257
5258 type AstExpressionRem struct {
5259     Location *SourceLocation // Optional
5260     Lhs      AstExpression
5261     Rh      AstExpression
5262 }
5263
5264 func (self AstExpressionRem) ExpressionLocation() *SourceLocation {
5265     return self.Location
5266 }
5267
5268 func (self AstExpressionRem) IntoValue(ctx *Context) Value {
5269     return ctx.NewMapOrPanic([]MapPair{
5270         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
5271         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5272         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
5273         {ctx.NewString("rhs"), self.Rhs.IntoValue(ctx)},

```

```

5274     })
5275 }
5276
5277 func (self *AstExpressionRem) Eval(ctx *Context, env *Environment) (Value, error) {
5278     lhs, err := self.Lhs.Eval(ctx, env)
5279     if err != nil {
5280         return nil, err
5281     }
5282
5283     rhs, err := self.Rhs.Eval(ctx, env)
5284     if err != nil {
5285         return nil, err
5286     }
5287
5288     lhsNumber, lhsIsNumber := lhs.(*Number)
5289     if lhsIsNumber {
5290         rhsNumber, rhsIsNumber := rhs.(*Number)
5291         if rhsIsNumber {
5292             // The remainder will have the same sign as the dividend.
5293             // This behavior is identical to C's remainder operator.
5294             // +7 % +3 => +1
5295             // +7 % -3 => +1
5296             // -7 % +3 => -1
5297             // -7 % -3 => -1
5298             return ctx.NewNumber(math.Mod(lhsNumber.data, rhsNumber.data)), nil
5299         }
5300     }
5301
5302     return nil, NewError(
5303         self.Location,
5304         ctx.NewStringf("attempted %% operation with types %s and %s",
5305             ↪ quote(lhs.TypeName()), quote(rhs.TypeName()))),
5306     )
5307 }
5308
5309 type AstExpressionAccessIndex struct {
5310     Location *SourceLocation // Optional
5311     Store    AstExpression
5312     Field    AstExpression
5313 }
5314
5315 func (self AstExpressionAccessIndex) ExpressionLocation() *SourceLocation {
5316     return self.Location
5317 }
5318
5319 func (self AstExpressionAccessIndex) IntoValue(ctx *Context) Value {
5320     return ctx.NewMapOrPanic([]MapPair{
5321         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
5322         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5323         {ctx.NewString("store"), self.Store.IntoValue(ctx)},
5324         {ctx.NewString("field"), self.Field.IntoValue(ctx)},
5325     })
5326 }
5327
5328 func (self *AstExpressionAccessIndex) Eval(ctx *Context, env *Environment) (Value, error)
5329 ↪ {
5330     store, err := self.Store.Eval(ctx, env)
5331     if err != nil {
5332         return nil, err
5333     }
5334
5335     field, err := self.Field.Eval(ctx, env)
5336     if err != nil {
5337         return nil, err
5338     }
5339
5340     accessVector := func(storeVector *Vector) (Value, error) {

```

```

5339         index, err := ValueAsIndex(field)
5340         if err != nil {
5341             return nil, NewError(
5342                 self.Location,
5343                 ctx.NewStringf("invalid vector access with index %v (%s)", field,
↪ err.Error()),
5344             )
5345         }
5346
5347         if index >= storeVector.Count() {
5348             return nil, NewError(
5349                 self.Location,
5350                 ctx.NewStringf("invalid vector access with index %v (vector has a count
↪ of %v)", field, storeVector.Count()),
5351             )
5352         }
5353
5354         return storeVector.Get(index), nil
5355     }
5356
5357     accessMap := func(storeMap *Map) (Value, error) {
5358         lookup, ok := storeMap.Lookup(field)
5359         if !ok {
5360             return nil, NewError(
5361                 self.Location,
5362                 ctx.NewStringf("invalid map access with field %v", field),
5363             )
5364         }
5365
5366         return lookup, nil
5367     }
5368
5369     if storeVector, ok := store.(*Vector); ok {
5370         return accessVector(storeVector)
5371     }
5372     if storeMap, ok := store.(*Map); ok {
5373         return accessMap(storeMap)
5374     }
5375     if storeReference, ok := store.(*Reference); ok {
5376         storeDeref := storeReference.data
5377         if storeDerefVector, ok := storeDeref.(*Vector); ok {
5378             return accessVector(storeDerefVector)
5379         }
5380         if storeDerefMap, ok := storeDeref.(*Map); ok {
5381             return accessMap(storeDerefMap)
5382         }
5383         return nil, NewError(
5384             self.Location,
5385             ctx.NewStringf("invalid %s to %s access with field %v", store.TypeName(),
↪ storeDeref.TypeName(), field),
5386         )
5387     }
5388
5389     return nil, NewError(
5390         self.Location,
5391         ctx.NewStringf("attempted to access field of type %s with type %s",
↪ quote(store.TypeName()), quote(field.TypeName())),
5392     )
5393 }
5394
5395 type AstExpressionAccessScope struct {
5396     Location *SourceLocation // Optional
5397     Store    AstExpression
5398     Field    *AstIdentifier
5399 }
5400
5401 func (self AstExpressionAccessScope) ExpressionLocation() *SourceLocation {

```

```

5402     return self.Location
5403 }
5404
5405 func (self AstExpressionAccessScope) IntoValue(ctx *Context) Value {
5406     return ctx.NewMapOrPanic([]MapPair{
5407         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
5408         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5409         {ctx.NewString("store"), self.Store.IntoValue(ctx)},
5410         {ctx.NewString("field"), self.Field.IntoValue(ctx)},
5411     })
5412 }
5413
5414 func (self *AstExpressionAccessScope) Eval(ctx *Context, env *Environment) (Value, error)
5415 ↪ {
5416     store, err := self.Store.Eval(ctx, env)
5417     if err != nil {
5418         return nil, err
5419     }
5420
5421     field := self.Field.Name
5422
5423     accessMap := func(storeMap *Map) (Value, error) {
5424         lookup, ok := storeMap.Lookup(field)
5425         if !ok {
5426             return nil, NewError(
5427                 self.Location,
5428                 ctx.NewStringf("invalid map access with field %v", field),
5429             )
5430         }
5431         return lookup, nil
5432     }
5433
5434     if storeMap, ok := store.(*Map); ok {
5435         return accessMap(storeMap)
5436     }
5437     if storeReference, ok := store.(*Reference); ok {
5438         storeDeref := storeReference.data
5439         if storeDerefMap, ok := storeDeref.(*Map); ok {
5440             return accessMap(storeDerefMap)
5441         }
5442         return nil, NewError(
5443             self.Location,
5444             ctx.NewStringf("invalid %s to %s access with field %v", store.TypeName(),
5445 ↪ storeDeref.TypeName(), field),
5446         )
5447     }
5448     return nil, NewError(
5449         self.Location,
5450         ctx.NewStringf("attempted to access field of type %s", quote(store.TypeName())),
5451     )
5452 }
5453
5454 type AstExpressionAccessDot struct {
5455     Location *SourceLocation // Optional
5456     Store    AstExpression
5457     Field    *AstIdentifier
5458 }
5459
5460 func (self AstExpressionAccessDot) ExpressionLocation() *SourceLocation {
5461     return self.Location
5462 }
5463
5464 func (self AstExpressionAccessDot) IntoValue(ctx *Context) Value {
5465     return ctx.NewMapOrPanic([]MapPair{
5466         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},

```

```

5467         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5468         {ctx.NewString("store"), self.Store.IntoValue(ctx)},
5469         {ctx.NewString("field"), self.Field.IntoValue(ctx)},
5470     })
5471 }
5472
5473 func (self *AstExpressionAccessDot) Eval(ctx *Context, env *Environment) (Value, error) {
5474     store, err := self.Store.Eval(ctx, env)
5475     if err != nil {
5476         return nil, err
5477     }
5478
5479     field := self.Field.Name
5480
5481     // When directly reading property via dot access, prioritize the fields
5482     // of value itself *before* looking at the fields of the value's
5483     // metamap. This is done so that operations such as:
5484     //
5485     //     somemap.foo
5486     //
5487     // will find the field "foo" rather than a metafunction `foo` in the
5488     // map, which is almost certainly the desired behavior for nominal
5489     // property lookup.
5490     if m, ok := store.(*Map); ok {
5491         lookup, ok := m.Lookup(field)
5492         if ok {
5493             return lookup, nil
5494         }
5495     }
5496
5497     if store.Meta(ctx) != nil {
5498         lookup, ok := store.Meta(ctx).Lookup(field)
5499         if ok {
5500             return lookup, nil
5501         }
5502     }
5503
5504     // Special case where a reference value is implicitly dereferenced when
5505     // accessing the target field.
5506     if reference, ok := store.(*Reference); ok {
5507         storeDeref := reference.data
5508
5509         // Prioritize fields of the value itself *before* looking at the
5510         // fields of the value's metamap.
5511         if m, ok := storeDeref.(*Map); ok {
5512             lookup, ok := m.Lookup(field)
5513             if ok {
5514                 return lookup, nil
5515             }
5516         }
5517
5518         if storeDeref.Meta(ctx) != nil {
5519             lookup, ok := storeDeref.Meta(ctx).Lookup(field)
5520             if ok {
5521                 return lookup, nil
5522             }
5523         }
5524
5525         return nil, NewError(
5526             self.Location,
5527             ctx.NewStringf("invalid %s to %s access with field %v", store.TypeName(),
5528 ↪ storeDeref.TypeName(), field),
5529         )
5530     }
5531
5532     return nil, NewError(
5533         self.Location,

```

```

5533         ctx.NewStringf("invalid %s access with field %v", store.TypeName(), field),
5534     )
5535 }
5536
5537 type AstExpressionMkref struct {
5538     Location *SourceLocation // Optional
5539     Lhs      AstExpression
5540 }
5541
5542 func (self AstExpressionMkref) ExpressionLocation() *SourceLocation {
5543     return self.Location
5544 }
5545
5546 func (self AstExpressionMkref) IntoValue(ctx *Context) Value {
5547     return ctx.NewMapOrPanic([]MapPair{
5548         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
5549         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5550         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
5551     })
5552 }
5553
5554 // Returns the constructed reference value as well as the full list of values
5555 // marked as referenced (access chain plus the returned referenced value). The
5556 // caller may choose to unmark the list of values at the end of the current
5557 // lexical scope.
5558 func (self *AstExpressionMkref) eval(ctx *Context, env *Environment) (*Reference, []Value,
5559     error) {
5560     chain := []Value{}
5561     value, err := evalLvalue(self.Lhs, ctx, env, &chain)
5562     if err != nil {
5563         return nil, nil, err
5564     }
5565     marked := []Value{}
5566     for _, v := range chain {
5567         markReferenced(v)
5568         marked = append(marked, v)
5569     }
5570     marked = append(marked, value)
5571     return ctx.newReference(value), marked, nil
5572 }
5573
5574 func (self *AstExpressionMkref) Eval(ctx *Context, env *Environment) (Value, error) {
5575     // References may only be constructed at specific points in a program:
5576     //
5577     // + From an implicit reference to self created when calling a function
5578     //   declared with function.&(self, args...) { ... } => foo.fn(bar)
5579     // + With the reference operator .& applied to function argument at the
5580     //   call site of an arbitrary function invocation => fn(foo.&, bar, baz.&)
5581     // + A key reference or value reference declared as an iteration variable
5582     //   in a for-loop => for k.& in x { ... } and for k, v.& in x { ... }
5583     //
5584     // Each of these places has dedicated syntax that is explicitly parsed as a
5585     // special case, and evaluated by the interpreter using the internal
5586     // AstExpressionMkref.eval() function.
5587     return nil, NewError(
5588         self.Location,
5589         ctx.NewString("invalid reference operator use"),
5590     )
5591 }
5592
5593 type AstExpressionDeref struct {
5594     Location *SourceLocation // Optional
5595     Lhs      AstExpression
5596 }
5597
5598 func (self AstExpressionDeref) ExpressionLocation() *SourceLocation {
5599     return self.Location
5600 }

```

```

5599 }
5600
5601 func (self AstExpressionDeref) IntoValue(ctx *Context) Value {
5602     return ctx.NewMapOrPanic([]MapPair{
5603         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
5604         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5605         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
5606     })
5607 }
5608
5609 func (self *AstExpressionDeref) Eval(ctx *Context, env *Environment) (Value, error) {
5610     value, err := self.Lhs.Eval(ctx, env)
5611     if err != nil {
5612         return nil, err
5613     }
5614
5615     reference, ok := value.(*Reference)
5616     if !ok {
5617         return nil, NewError(
5618             self.Location,
5619             ctx.NewStringf("attempted dereference of non-reference type %s",
5620 ↪ quote(value.TypeName()))),
5621     }
5622
5623     return reference.data, nil
5624 }
5625
5626 type AstExpressionFunctionCall struct {
5627     Location *SourceLocation // Optional
5628     Function AstExpression
5629     Arguments []AstExpression
5630 }
5631
5632 func (self AstExpressionFunctionCall) ExpressionLocation() *SourceLocation {
5633     return self.Location
5634 }
5635
5636 func (self AstExpressionFunctionCall) IntoValue(ctx *Context) Value {
5637     arguments := ctx.NewVectorOrPanic(nil)
5638     for _, argument := range self.Arguments {
5639         _ = arguments.Push(argument.IntoValue(ctx))
5640     }
5641     return ctx.NewMapOrPanic([]MapPair{
5642         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
5643         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5644         {ctx.NewString("function"), self.Function.IntoValue(ctx)},
5645         {ctx.NewString("arguments"), arguments},
5646     })
5647 }
5648
5649 func (self *AstExpressionFunctionCall) Eval(ctx *Context, env *Environment) (Value,
5650 ↪ error) {
5651     var function Value = nil
5652     var callableSelfArgument Value = nil
5653     var err error = nil
5654     if accessDot, ok := self.Function.(*AstExpressionAccessDot); ok {
5655         // Special case when dot access is used for a function call. An
5656         // implicit `self` argument is passed to the function, either by value
5657         // or by reference, depending on whether the function was declared
5658         // with:
5659         //
5660         // function(self) { ... } # pass by value
5661         // function.&(self) { ... } # pass by reference
5662         chain := []Value{}
5663         store, err := evalLvalue(accessDot.Store, ctx, env, &chain)
5664         if err != nil {

```

```

5664         return nil, err
5665     }
5666     storeIsSelfReference := false
5667
5668     // When making a function call using dot access syntax, perform
5669     // function lookup using the value's metemap *before* looking at the
5670     // fields of the value itself. This is done so that expressions such
5671     // as:
5672     //
5673     //     somemap.foo()
5674     //
5675     // will find the metafunction `foo` rather than some key "foo" in
5676     // the map, which is almost certainly *not* the desired behavior.
5677     if meta := store.Meta(ctx); meta != nil {
5678         // Value meta lookup.
5679         function, _ = meta.Lookup(accessDot.Field.Name)
5680     }
5681     if function == nil {
5682         // Map field lookup.
5683         if storeMap, ok := store.(*Map); ok {
5684             function, _ = storeMap.Lookup(accessDot.Field.Name)
5685         }
5686     }
5687     if function == nil {
5688         // Implicit value dereference meta lookup.
5689         if storeReference, ok := store.(*Reference); ok {
5690             if meta := storeReference.data.Meta(ctx); meta != nil {
5691                 storeIsSelfReference = true
5692                 function, _ = meta.Lookup(accessDot.Field.Name)
5693             }
5694         }
5695     }
5696     if function == nil {
5697         return nil, NewError(
5698             self.Location,
5699             ctx.NewStringf("invalid method access with name %v",
5700 ↪ accessDot.Field.Name),
5701         )
5702     }
5703
5704     // Construct the implicit `self` argument, which is either going to be
5705     // the lhs store by copy (value -> pass by value), the lhs store by new
5706     // reference (value.& -> pass by reference), the dereferenced lhs store
5707     // by copy (value.* -> pass by value), or a passthrough of the existing
5708     // reference (pass by reference).
5709     if callableSelfIsPassedByReference(function) {
5710         for _, value := range chain {
5711             markReferenced(value)
5712         }
5713         if storeIsSelfReference {
5714             // Reference passthrough (auto-deref)
5715             callableSelfArgument = store
5716         } else {
5717             // value.& -> pass by reference
5718             callableSelfArgument = ctx.newReference(store)
5719         }
5720     }
5721
5722     // The implicit `self` reference is guaranteed not to escape the
5723     // lexical environment of the callee, and can be safely unmarked
5724     // after control returns from the invoked function.
5725     defer func() {
5726         for _, value := range chain {
5727             unmarkReferenced(value)
5728         }
5729         if !storeIsSelfReference {
5730             unmarkReferenced(store)
5731         }
5732     }()

```

```

5730         }()
5731     } else {
5732         if storeIsSelfReference {
5733             // value.* -> pass by value (auto-deref)
5734             callableSelfArgument = store.(*Reference).data.Copy()
5735         } else {
5736             // value -> pass by value
5737             callableSelfArgument = store.Copy()
5738         }
5739     }
5740 } else {
5741     function, err = self.Function.Eval(ctx, env)
5742     if err != nil {
5743         return nil, err
5744     }
5745 }
5746
5747 capacity := len(self.Arguments)
5748 if callableSelfArgument != nil {
5749     capacity += 1
5750 }
5751 arguments := make([]Value, 0, capacity)
5752 if callableSelfArgument != nil {
5753     arguments = append(arguments, callableSelfArgument)
5754 }
5755
5756 // Arguments passed by reference with the .& operator are guaranteed not to
5757 // escape the lexical environment of the callee, and can be safely unmarked
5758 // after control returns from the invoked function.
5759 var referenced []Value
5760 defer func() {
5761     for _, value := range referenced {
5762         unmarkReferenced(value)
5763     }
5764 }()
5765
5766 for _, argument := range self.Arguments {
5767     if argumentMkref, ok := argument.(*AstExpressionMkref); ok {
5768         reference, marked, err := argumentMkref.eval(ctx, env)
5769         if err != nil {
5770             return nil, err
5771         }
5772         referenced = append(referenced, marked...)
5773         arguments = append(arguments, reference)
5774         continue
5775     }
5776
5777     result, err := argument.Eval(ctx, env)
5778     if err != nil {
5779         return nil, err
5780     }
5781     arguments = append(arguments, moveOrCopy(result))
5782 }
5783
5784 return call(ctx, self.Location, function, arguments)
5785 }
5786
5787 type AstBlock struct {
5788     Location *SourceLocation // Optional
5789     Statements []AstStatement
5790 }
5791
5792 func (self AstBlock) IntoValue(ctx *Context) Value {
5793     statements := ctx.NewVectorOrPanic(nil)
5794     for _, statement := range self.Statements {
5795         _ = statements.Push(statement.IntoValue(ctx))
5796     }

```

```

5797     return ctx.NewMapOrPanic([]MapPair{
5798         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
5799         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5800         {ctx.NewString("statements"), statements},
5801     })
5802 }
5803
5804 func (self *AstBlock) Eval(ctx *Context, env *Environment) (ControlFlow, error) {
5805     for _, statement := range self.Statements {
5806         cflow, error := statement.Eval(ctx, env)
5807         if error != nil {
5808             return ControlFlow{}, error
5809         }
5810         if cflow.Kind != ControlFlowNone {
5811             return cflow, nil
5812         }
5813     }
5814     return ControlFlow{}, nil
5815 }
5816
5817 type AstConditional struct {
5818     Location *SourceLocation // Optional
5819     Condition AstExpression
5820     Body      *AstBlock
5821 }
5822
5823 func (self AstConditional) IntoValue(ctx *Context) Value {
5824     return ctx.NewMapOrPanic([]MapPair{
5825         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
5826         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5827         {ctx.NewString("condition"), self.Condition.IntoValue(ctx)},
5828         {ctx.NewString("body"), self.Body.IntoValue(ctx)},
5829     })
5830 }
5831
5832 func (self AstConditional) exec(ctx *Context, env *Environment) (ControlFlow, bool,
5833     ↪ error) {
5834     value, error := self.Condition.Eval(ctx, env)
5835     if error != nil {
5836         return ControlFlow{}, false, error
5837     }
5838     valueBoolean, ok := value.(*Boolean)
5839     if !ok {
5840         return ControlFlow{}, false, NewError(
5841             self.Location,
5842             ctx.NewStringf("conditional with non-boolean type %s",
5843                 ↪ quote(value.TypeName()))
5844         )
5845     }
5846     if valueBoolean.data {
5847         env := NewEnvironment(env)
5848         result, err := self.Body.Eval(ctx, env)
5849         env.tryReuse()
5850         return result, true, err
5851     }
5852     return ControlFlow{}, false, nil
5853 }
5854
5855 func (self *AstConditional) Eval(ctx *Context, env *Environment) (ControlFlow, error) {
5856     result, _, err := self.exec(ctx, env)
5857     return result, err
5858 }
5859
5860 type AstStatementLet struct {
5861

```

```

5862     Location    *SourceLocation // Optional
5863     Identifier  *AstIdentifier
5864     Expression  AstExpression
5865 }
5866
5867 func (self AstStatementLet) StatementLocation() *SourceLocation {
5868     return self.Location
5869 }
5870
5871 func (self AstStatementLet) IntoValue(ctx *Context) Value {
5872     return ctx.NewMapOrPanic([]MapPair{
5873         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
5874         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5875         {ctx.NewString("identifier"), self.Identifier.IntoValue(ctx)},
5876         {ctx.NewString("expression"), self.Expression.IntoValue(ctx)},
5877     })
5878 }
5879
5880 func (self *AstStatementLet) Eval(ctx *Context, env *Environment) (ControlFlow, error) {
5881     rhs, error := self.Expression.Eval(ctx, env)
5882     if error != nil {
5883         return ControlFlow{}, error
5884     }
5885     if rhsReference, ok := rhs.(*Reference); ok {
5886         return ControlFlow{}, NewError(
5887             self.Location,
5888             ctx.NewStringf("attempted assignment statement with type reference to %v",
5889 ↪ rhsReference.data.TypeName()),
5890         )
5891     }
5892     env.letWithLocation(self.Identifier.Name.data, moveOrCopy(rhs), self.Location)
5893     return ControlFlow{}, nil
5894 }
5895
5896 type AstStatementIfElifElse struct {
5897     Location    *SourceLocation // Optional
5898     Conditionals []*AstConditional
5899     ElseBlock   *AstBlock // Optional
5900 }
5901
5902 func (self AstStatementIfElifElse) StatementLocation() *SourceLocation {
5903     return self.Location
5904 }
5905
5906 func (self AstStatementIfElifElse) IntoValue(ctx *Context) Value {
5907     conditionals := ctx.NewVectorOrPanic(nil)
5908     for _, conditional := range self.Conditionals {
5909         _ = conditionals.Push(conditional.IntoValue(ctx))
5910     }
5911     var elseBlock Value = ctx.NewNull()
5912     if self.ElseBlock != nil {
5913         elseBlock = self.ElseBlock.IntoValue(ctx)
5914     }
5915     return ctx.NewMapOrPanic([]MapPair{
5916         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
5917         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5918         {ctx.NewString("conditionals"), conditionals},
5919         {ctx.NewString("else_block"), elseBlock},
5920     })
5921 }
5922
5923 func (self *AstStatementIfElifElse) Eval(ctx *Context, env *Environment) (ControlFlow,
5924 ↪ error) {
5925     for _, conditional := range self.Conditionals {
5926         result, executed, err := conditional.exec(ctx, env)

```

```

5927         if err != nil {
5928             return ControlFlow{}, err
5929         }
5930         if executed {
5931             return result, nil
5932         }
5933     }
5934
5935     if self.ElseBlock != nil {
5936         elseEnv := NewEnvironment(env)
5937         result, err := self.ElseBlock.Eval(ctx, elseEnv)
5938         elseEnv.tryReuse()
5939         return result, err
5940     }
5941
5942     return ControlFlow{}, nil
5943 }
5944
5945 type AstStatementFor struct {
5946     Location      *SourceLocation // Optional
5947     IdentifierK   *AstIdentifier
5948     IdentifierV   *AstIdentifier // Optional
5949     KIsReference  bool
5950     VIsReference  bool
5951     Collection    AstExpression
5952     Block        *AstBlock
5953 }
5954
5955 func (self AstStatementFor) StatementLocation() *SourceLocation {
5956     return self.Location
5957 }
5958
5959 func (self AstStatementFor) IntoValue(ctx *Context) Value {
5960     var identifierV Value = ctx.NewNull()
5961     if self.IdentifierV != nil {
5962         identifierV = self.IdentifierV.IntoValue(ctx)
5963     }
5964
5965     var vIsReference Value = ctx.NewNull()
5966     if self.IdentifierV != nil {
5967         vIsReference = ctx.NewBoolean(self.VIsReference)
5968     }
5969
5970     return ctx.NewMapOrPanic([]MapPair{
5971         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
5972         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
5973         {ctx.NewString("identifier_k"), self.IdentifierK.IntoValue(ctx)},
5974         {ctx.NewString("identifier_v"), identifierV},
5975         {ctx.NewString("k_is_reference"), ctx.NewBoolean(self.KIsReference)},
5976         {ctx.NewString("v_is_reference"), vIsReference},
5977         {ctx.NewString("collection"), self.Collection.IntoValue(ctx)},
5978         {ctx.NewString("block"), self.Block.IntoValue(ctx)},
5979     })
5980 }
5981
5982 func (self *AstStatementFor) Eval(ctx *Context, env *Environment) (ControlFlow, error) {
5983     collection, error := self.Collection.Eval(ctx, env)
5984     if error != nil {
5985         return ControlFlow{}, error
5986     }
5987     if self.KIsReference || self.VIsReference {
5988         // Reference iteration aliases elements from the original collection.
5989         // The collection is specifically *not* copied here, since updates
5990         // through referenced values should be observed in that original
5991         // collection.
5992         collection.CopyOnWrite()
5993         markReferenced(collection)

```

```

5994     defer unmarkReferenced(collection)
5995 } else {
5996     // Value iteration iterates over copies of each element, so we iterate
5997     // over a shallow copy of the collection to allow modification of that
5998     // original collection during iteration.
5999     collection = collection.Copy()
6000 }
6001
6002 if metaFunction, ok := MetaFunction(ctx, collection, ctx.constStringNext); ok {
6003     if self.IdentifierV != nil {
6004         return ControlFlow{}, NewError(
6005             self.Location,
6006             ctx.NewStringf("attempted key-value iteration over iterator %s",
↪ quote(collection.TypeName()))),
6007         )
6008     }
6009     if self.KIsReference {
6010         return ControlFlow{}, NewError(
6011             self.Location,
6012             ctx.NewStringf("cannot use a key-reference over iterator %s",
↪ quote(collection.TypeName()))),
6013         )
6014     }
6015
6016     if !callableSelfIsPassedByReference(metaFunction) {
6017         return ControlFlow{}, NewError(
6018             self.Location,
6019             ctx.NewString("iterator next must receive self by reference (declared
↪ with function.&(self))"),
6020         )
6021     }
6022     reference := ctx.newReference(collection)
6023     defer unmarkReferenced(collection)
6024     for {
6025         iterated, err := call(ctx, self.Location, metaFunction, []Value{reference})
6026         if err != nil {
6027             if error, ok := err.(Error); ok {
6028                 if _, ok := error.Value.(*Null); ok {
6029                     break // end-of-iteration
6030                 }
6031             }
6032             return ControlFlow{}, err
6033         }
6034         loopEnv := NewEnvironment(env)
6035         loopEnv.letWithLocation(self.IdentifierK.Name.data, moveOrCopy(iterated),
↪ self.IdentifierK.Location)
6036         result, err := self.Block.Eval(ctx, loopEnv)
6037         loopEnv.tryReuse()
6038         if err != nil {
6039             return ControlFlow{}, err
6040         }
6041         if result.Kind == ControlFlowReturn {
6042             return result, nil
6043         }
6044         if result.Kind == ControlFlowBreak {
6045             return ControlFlow{}, nil
6046         }
6047         if result.Kind == ControlFlowContinue {
6048             continue
6049         }
6050     }
6051 } else if collectionNumber, ok := collection.(*Number); ok {
6052     if self.IdentifierV != nil {
6053         return ControlFlow{}, NewError(
6054             self.Location,
6055             ctx.NewStringf("attempted key-value iteration over type %s",
↪ quote(collection.TypeName()))),

```

```

6056         )
6057     }
6058     if self.KIsReference {
6059         return ControlFlow{}, NewError(
6060             self.Location,
6061             ctx.NewStringf("cannot use a key-reference over type %s",
↪ quote(collection.TypeName()))),
6062         )
6063     }
6064     collectionInteger, err := ValueAsIndex(collectionNumber)
6065     if err != nil {
6066         return ControlFlow{}, NewError(
6067             self.Location,
6068             ctx.NewString(err.Error()),
6069         )
6070     }
6071     for i := range collectionInteger {
6072         loopEnv := NewEnvironment(env)
6073         loopEnv.letWithLocation(self.IdentifierK.Name.data,
↪ ctx.NewNumber(float64(i)), self.IdentifierK.Location)
6074         result, err := self.Block.Eval(ctx, loopEnv)
6075         loopEnv.tryReuse()
6076         if err != nil {
6077             return ControlFlow{}, err
6078         }
6079         if result.Kind == ControlFlowReturn {
6080             return result, nil
6081         }
6082         if result.Kind == ControlFlowBreak {
6083             return ControlFlow{}, nil
6084         }
6085         if result.Kind == ControlFlowContinue {
6086             continue
6087         }
6088     }
6089     } else if collectionVector, ok := collection.(*Vector); ok {
6090         if self.IdentifierV != nil {
6091             return ControlFlow{}, NewError(
6092                 self.Location,
6093                 ctx.NewStringf("attempted key-value iteration over type %s",
↪ quote(collection.TypeName()))),
6094             )
6095         }
6096         // Save the list of elements so that modification to the collection
6097         // during iteration does not change the sequence of iteration.
6098         elements := collectionVector.Elements()
6099         snapshot := make([]Value, len(elements))
6100         copy(snapshot, elements)
6101         for _, x := range snapshot {
6102             var k Value = nil
6103             if self.KIsReference {
6104                 k = ctx.newReference(x)
6105             } else {
6106                 k = x.Copy()
6107             }
6108             loopEnv := NewEnvironment(env)
6109             loopEnv.letWithLocation(self.IdentifierK.Name.data, k,
↪ self.IdentifierK.Location)
6110             result, err := self.Block.Eval(ctx, loopEnv)
6111             if self.KIsReference {
6112                 unmarkReferenced(x)
6113             }
6114             loopEnv.tryReuse()
6115             if err != nil {
6116                 return ControlFlow{}, err
6117             }
6118             if result.Kind == ControlFlowReturn {

```

```

6119         return result, nil
6120     }
6121     if result.Kind == ControlFlowBreak {
6122         return ControlFlow{}, nil
6123     }
6124     if result.Kind == ControlFlowContinue {
6125         continue
6126     }
6127 }
6128 } else if collectionMap, ok := collection.(*Map); ok {
6129     if self.KIsReference {
6130         return ControlFlow{}, NewError(
6131             self.Location,
6132             ctx.NewStringf("cannot use a key-reference over type %s",
↪ quote(collection.TypeName()))),
6133         )
6134     }
6135     // Save the list of key-value pairs so that modification to the
6136     // collection during iteration does not change the sequence of
6137     // iteration.
6138     pairs := collectionMap.Pairs()
6139     for _, pair := range pairs {
6140         loopEnv := NewEnvironment(env)
6141         loopEnv.setWithLocation(self.IdentifierK.Name.data, pair.Key.Copy(),
↪ self.IdentifierK.Location)
6142         if self.IdentifierV != nil {
6143             var v Value = nil
6144             if self.VIsReference {
6145                 v = ctx.newReference(pair.Value)
6146             } else {
6147                 v = pair.Value.Copy()
6148             }
6149             loopEnv.setWithLocation(self.IdentifierV.Name.data, v,
↪ self.IdentifierV.Location)
6150         }
6151         result, err := self.Block.Eval(ctx, loopEnv)
6152         if self.VIsReference {
6153             unmarkReferenced(pair.Value)
6154         }
6155         loopEnv.tryReuse()
6156         if err != nil {
6157             return ControlFlow{}, err
6158         }
6159         if result.Kind == ControlFlowReturn {
6160             return result, nil
6161         }
6162         if result.Kind == ControlFlowBreak {
6163             return ControlFlow{}, nil
6164         }
6165         if result.Kind == ControlFlowContinue {
6166             continue
6167         }
6168     }
6169 } else if collectionSet, ok := collection.(*Set); ok {
6170     if self.IdentifierV != nil {
6171         return ControlFlow{}, NewError(
6172             self.Location,
6173             ctx.NewStringf("attempted key-value iteration over type %s",
↪ quote(collection.TypeName()))),
6174         )
6175     }
6176     if self.KIsReference {
6177         return ControlFlow{}, NewError(
6178             self.Location,
6179             ctx.NewStringf("cannot use a key-reference over type %s",
↪ quote(collection.TypeName()))),
6180     }

```

```

6181     }
6182     for _, element := range collectionSet.Elements() {
6183         loopEnv := NewEnvironment(env)
6184         loopEnv.WithLocation(self.IdentifierK.Name.data, element.Copy(),
↪ self.IdentifierK.Location)
6185         result, err := self.Block.Eval(ctx, loopEnv)
6186         loopEnv.tryReuse()
6187         if err != nil {
6188             return ControlFlow{}, err
6189         }
6190         if result.Kind == ControlFlowReturn {
6191             return result, nil
6192         }
6193         if result.Kind == ControlFlowBreak {
6194             return ControlFlow{}, nil
6195         }
6196         if result.Kind == ControlFlowContinue {
6197             continue
6198         }
6199     }
6200 } else {
6201     return ControlFlow{}, NewError(
6202         self.Location,
6203         ctx.NewStringf("attempted iteration over type %s",
↪ quote(collection.TypeName())),
6204     )
6205 }
6206
6207 return ControlFlow{}, nil
6208 }
6209
6210 type AstStatementWhile struct {
6211     Location *SourceLocation // Optional
6212     Expression AstExpression
6213     Block *AstBlock
6214 }
6215
6216 func (self AstStatementWhile) StatementLocation() *SourceLocation {
6217     return self.Location
6218 }
6219
6220 func (self AstStatementWhile) IntoValue(ctx *Context) Value {
6221     return ctx.NewMapOrPanic([]MapPair{
6222         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
6223         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
6224         {ctx.NewString("expression"), self.Expression.IntoValue(ctx)},
6225         {ctx.NewString("block"), self.Block.IntoValue(ctx)},
6226     })
6227 }
6228
6229 func (self *AstStatementWhile) Eval(ctx *Context, env *Environment) (ControlFlow, error)
↪ {
6230     for {
6231         expression, error := self.Expression.Eval(ctx, env)
6232         if error != nil {
6233             return ControlFlow{}, error
6234         }
6235
6236         expressionBoolean, ok := expression.(*Boolean)
6237         if !ok {
6238             return ControlFlow{}, NewError(
6239                 self.Location,
6240                 ctx.NewStringf("conditional with non-boolean type %s",
↪ quote(expression.TypeName())),
6241             )
6242         }
6243

```

```

6244         if !expressionBoolean.data {
6245             break
6246         }
6247
6248         loopEnv := NewEnvironment(env)
6249         result, err := self.Block.Eval(ctx, loopEnv)
6250         loopEnv.tryReuse()
6251         if err != nil {
6252             return ControlFlow{}, err
6253         }
6254         if result.Kind == ControlFlowReturn {
6255             return result, nil
6256         }
6257         if result.Kind == ControlFlowBreak {
6258             return ControlFlow{}, nil
6259         }
6260         if result.Kind == ControlFlowContinue {
6261             continue
6262         }
6263     }
6264
6265     return ControlFlow{}, nil
6266 }
6267
6268 type AstStatementBreak struct {
6269     Location *SourceLocation // Optional
6270 }
6271
6272 func (self AstStatementBreak) StatementLocation() *SourceLocation {
6273     return self.Location
6274 }
6275
6276 func (self AstStatementBreak) IntoValue(ctx *Context) Value {
6277     return ctx.NewMapOrPanic([]MapPair{
6278         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
6279         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
6280     })
6281 }
6282
6283 func (self *AstStatementBreak) Eval(ctx *Context, env *Environment) (ControlFlow, error) {
6284     ↪ {
6285         return ControlFlow{ControlFlowBreak, self.Location, nil}, nil
6286     }
6287
6288 type AstStatementContinue struct {
6289     Location *SourceLocation // Optional
6290 }
6291
6292 func (self AstStatementContinue) StatementLocation() *SourceLocation {
6293     return self.Location
6294 }
6295
6296 func (self AstStatementContinue) IntoValue(ctx *Context) Value {
6297     return ctx.NewMapOrPanic([]MapPair{
6298         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
6299         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
6300     })
6301 }
6302
6303 func (self *AstStatementContinue) Eval(ctx *Context, env *Environment) (ControlFlow,
6304     ↪ error) {
6305     return ControlFlow{ControlFlowContinue, self.Location, nil}, nil
6306 }
6307
6308 type AstStatementTry struct {
6309     Location      *SourceLocation // Optional
6310     TryBlock      *AstBlock

```

```

6309     CatchIdentifier *AstIdentifier // Optional
6310     CatchBlock      *AstBlock
6311 }
6312
6313 func (self AstStatementTry) StatementLocation() *SourceLocation {
6314     return self.Location
6315 }
6316
6317 func (self AstStatementTry) IntoValue(ctx *Context) Value {
6318     var catchIdentifier Value = ctx.NewNull()
6319     if self.CatchIdentifier != nil {
6320         catchIdentifier = self.CatchIdentifier.IntoValue(ctx)
6321     }
6322
6323     return ctx.NewMapOrPanic([]MapPair{
6324         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
6325         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
6326         {ctx.NewString("try_block"), self.TryBlock.IntoValue(ctx)},
6327         {ctx.NewString("catch_identifier"), catchIdentifier},
6328         {ctx.NewString("catch_block"), self.CatchBlock.IntoValue(ctx)},
6329     })
6330 }
6331
6332 func (self *AstStatementTry) Eval(ctx *Context, env *Environment) (ControlFlow, error) {
6333     tryEnv := NewEnvironment(env)
6334     result, err := self.TryBlock.Eval(ctx, tryEnv)
6335     tryEnv.tryReuse()
6336     if err != nil {
6337         error, ok := err.(Error)
6338         if !ok {
6339             return ControlFlow{}, err
6340         }
6341         catchEnv := NewEnvironment(env)
6342         if self.CatchIdentifier != nil {
6343             catchEnv.letWithLocation(self.CatchIdentifier.Name.data, error.Value.Copy(),
6344 ↪ self.CatchIdentifier.Location)
6345         }
6346         result, err := self.CatchBlock.Eval(ctx, catchEnv)
6347         catchEnv.tryReuse()
6348         return result, err
6349     }
6350     if result.Kind == ControlFlowReturn {
6351         return result, nil
6352     }
6353     if result.Kind == ControlFlowBreak {
6354         return result, nil
6355     }
6356     if result.Kind == ControlFlowContinue {
6357         return result, nil
6358     }
6359     return ControlFlow{}, nil
6360 }
6361
6362 type AstStatementError struct {
6363     Location *SourceLocation // Optional
6364     Expression AstExpression
6365 }
6366
6367 func (self AstStatementError) StatementLocation() *SourceLocation {
6368     return self.Location
6369 }
6370
6371 func (self AstStatementError) IntoValue(ctx *Context) Value {
6372     return ctx.NewMapOrPanic([]MapPair{
6373         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
6374         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
6375         {ctx.NewString("expression"), self.Expression.IntoValue(ctx)},
6376     })

```

```

6375     })
6376 }
6377
6378 func (self *AstStatementError) Eval(ctx *Context, env *Environment) (ControlFlow, error)
↳ {
6379     value, err := self.Expression.Eval(ctx, env)
6380     if err != nil {
6381         return ControlFlow{}, err
6382     }
6383     if valueReference, ok := value.(*Reference); ok {
6384         return ControlFlow{}, NewError(
6385             self.Location,
6386             ctx.NewStringf("attempted error statement with type reference to %v",
↳ valueReference.data.TypeName()),
6387         )
6388     }
6389
6390     return ControlFlow{}, NewError(self.Location, moveOrCopy(value))
6391 }
6392
6393 type AstStatementReturn struct {
6394     Location *SourceLocation // Optional
6395     Expression AstExpression // Optional
6396 }
6397
6398 func (self AstStatementReturn) StatementLocation() *SourceLocation {
6399     return self.Location
6400 }
6401
6402 func (self AstStatementReturn) IntoValue(ctx *Context) Value {
6403     var expression Value = ctx.NewNull()
6404     if self.Expression != nil {
6405         expression = self.Expression.IntoValue(ctx)
6406     }
6407
6408     return ctx.NewMapOrPanic([]MapPair{
6409         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
6410         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
6411         {ctx.NewString("expression"), expression},
6412     })
6413 }
6414
6415 func (self *AstStatementReturn) Eval(ctx *Context, env *Environment) (ControlFlow, error)
↳ {
6416     if self.Expression == nil {
6417         return ControlFlow{ControlFlowReturn, self.Location, ctx.NewNull()}, nil
6418     }
6419
6420     value, err := self.Expression.Eval(ctx, env)
6421     if err != nil {
6422         return ControlFlow{}, err
6423     }
6424     if valueReference, ok := value.(*Reference); ok {
6425         return ControlFlow{}, NewError(
6426             self.Location,
6427             ctx.NewStringf("attempted return statement with type reference to %v",
↳ valueReference.data.TypeName()),
6428         )
6429     }
6430
6431     return ControlFlow{ControlFlowReturn, self.Location, moveOrCopy(value)}, nil
6432 }
6433
6434 type AstStatementAssignment struct {
6435     Location *SourceLocation // Optional
6436     Lhs      AstExpression
6437     Rh      AstExpression

```

```

6438 }
6439
6440 func (self AstStatementAssignment) StatementLocation() *SourceLocation {
6441     return self.Location
6442 }
6443
6444 func (self AstStatementAssignment) IntoValue(ctx *Context) Value {
6445     return ctx.NewMapOrPanic([]MapPair{
6446         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
6447         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
6448         {ctx.NewString("lhs"), self.Lhs.IntoValue(ctx)},
6449         {ctx.NewString("rhs"), self.Rhs.IntoValue(ctx)},
6450     })
6451 }
6452
6453 // The evalLvalue() function is similar to the Eval() method associated with
6454 // every AstExpression, but is intended to be used specifically for expressions
6455 // on the left hand side of an assignment statement. Unlike Eval(), this
6456 // function will recursively perform a copy-on-write operation when an index,
6457 // scope, or dot access expression is encountered *before* performing that
6458 // access operation, so that modification of nested values will not mutate
6459 // shared objects pointed to by semantically separate values.
6460 //
6461 // If `chain` is not nil, then each value in the post-copy-on-write access
6462 // chain to the evaluated value will be recorded by appending to `*chain`.
6463 func evalLvalue(expr AstExpression, ctx *Context, env *Environment, chain *[]Value)
    ↪ (Value, error) {
6464     if expr.AccessIndex, ok := expr.(*AstExpressionAccessIndex); ok {
6465         innerStore, err := evalLvalue(expr.AccessIndex.Store, ctx, env, chain)
6466         if err != nil {
6467             return nil, err
6468         }
6469         innerStore.CopyOnWrite()
6470         if chain != nil {
6471             *chain = append(*chain, innerStore)
6472         }
6473
6474         field, err := expr.AccessIndex.Field.Eval(ctx, env)
6475         if err != nil {
6476             return nil, err
6477         }
6478
6479         accessVector := func(storeVector *Vector) (Value, error) {
6480             index, err := ValueAsIndex(field)
6481             if err != nil {
6482                 return nil, NewError(
6483                     expr.AccessIndex.Location,
6484                     ctx.NewStringf("invalid vector access with index %v (%s)", field,
    ↪ err.Error()),
6485                 )
6486             }
6487
6488             if index >= storeVector.Count() {
6489                 return nil, NewError(
6490                     expr.AccessIndex.Location,
6491                     ctx.NewStringf("invalid vector access with index %v (vector has a
    ↪ count of %v)", field, storeVector.Count()),
6492                 )
6493             }
6494
6495             return storeVector.Get(index), nil
6496         }
6497
6498         accessMap := func(storeMap *Map) (Value, error) {
6499             lookup, ok := storeMap.Lookup(field)
6500             if !ok {
6501                 return nil, NewError(

```

```

6502             exprAccessIndex.Location,
6503             ctx.NewStringf("invalid map access with field %v", field),
6504         )
6505     }
6506
6507     return lookup, nil
6508 }
6509
6510     if storeVector, ok := innerStore.(*Vector); ok {
6511         return accessVector(storeVector)
6512     }
6513     if storeMap, ok := innerStore.(*Map); ok {
6514         return accessMap(storeMap)
6515     }
6516     if storeReference, ok := innerStore.(*Reference); ok {
6517         storeDeref := storeReference.data
6518         storeDeref.CopyOnWrite()
6519         if chain != nil {
6520             *chain = append(*chain, storeDeref)
6521         }
6522
6523         if storeDerefVector, ok := storeDeref.(*Vector); ok {
6524             return accessVector(storeDerefVector)
6525         }
6526         if storeDerefMap, ok := storeDeref.(*Map); ok {
6527             return accessMap(storeDerefMap)
6528         }
6529         return nil, NewError(
6530             exprAccessIndex.Location,
6531             ctx.NewStringf("invalid %s to %s access with field %v",
6532 ↪ innerStore.TypeName(), storeDeref.TypeName(), field),
6533         )
6534     }
6535     return nil, NewError(
6536         exprAccessIndex.Location,
6537         ctx.NewStringf("attempted to access field of type %s with type %s",
6538 ↪ quote(innerStore.TypeName()), quote(field.TypeName())),
6539     )
6540 }
6541
6542     if exprAccessScope, ok := expr.(*AstExpressionAccessScope); ok {
6543         innerStore, err := evalLvalue(exprAccessScope.Store, ctx, env, chain)
6544         if err != nil {
6545             return nil, err
6546         }
6547         innerStore.CopyOnWrite()
6548         if chain != nil {
6549             *chain = append(*chain, innerStore)
6550         }
6551
6552         field := exprAccessScope.Field.Name
6553
6554         accessMap := func(storeMap *Map) (Value, error) {
6555             lookup, ok := storeMap.Lookup(field)
6556             if !ok {
6557                 return nil, NewError(
6558                     exprAccessScope.Location,
6559                     ctx.NewStringf("invalid map access with field %v", field),
6560                 )
6561             }
6562             return lookup, nil
6563         }
6564
6565         if storeMap, ok := innerStore.(*Map); ok {
6566             return accessMap(storeMap)

```

```

6567     }
6568     if storeReference, ok := innerStore.(*Reference); ok {
6569         storeDeref := storeReference.data
6570         storeDeref.CopyOnWrite()
6571         if chain != nil {
6572             *chain = append(*chain, storeDeref)
6573         }
6574
6575         if storeDerefMap, ok := storeDeref.(*Map); ok {
6576             return accessMap(storeDerefMap)
6577         }
6578         return nil, NewError(
6579             exprAccessScope.Location,
6580             ctx.NewStringf("invalid %s to %s access with field %v",
↪ innerStore.TypeName(), storeDeref.TypeName(), field),
6581         )
6582     }
6583
6584     return nil, NewError(
6585         exprAccessScope.Location,
6586         ctx.NewStringf("attempted to access field of type %s",
↪ quote(innerStore.TypeName()))
6587     )
6588 }
6589
6590 if exprAccessDot, ok := expr.(*AstExpressionAccessDot); ok {
6591     innerStore, err := evalLvalue(exprAccessDot.Store, ctx, env, chain)
6592     if err != nil {
6593         return nil, err
6594     }
6595     innerStore.CopyOnWrite()
6596     if chain != nil {
6597         *chain = append(*chain, innerStore)
6598     }
6599
6600     field := exprAccessDot.Field.Name
6601
6602     accessMap := func(storeMap *Map) (Value, error) {
6603         lookup, ok := storeMap.Lookup(field)
6604         if !ok {
6605             return nil, NewError(
6606                 exprAccessDot.Location,
6607                 ctx.NewStringf("invalid map access with field %v", field),
6608             )
6609         }
6610
6611         return lookup, nil
6612     }
6613
6614     if storeMap, ok := innerStore.(*Map); ok {
6615         return accessMap(storeMap)
6616     }
6617     if storeReference, ok := innerStore.(*Reference); ok {
6618         storeDeref := storeReference.data
6619         storeDeref.CopyOnWrite()
6620         if chain != nil {
6621             *chain = append(*chain, storeDeref)
6622         }
6623
6624         if storeDerefMap, ok := storeDeref.(*Map); ok {
6625             return accessMap(storeDerefMap)
6626         }
6627         return nil, NewError(
6628             exprAccessDot.Location,
6629             ctx.NewStringf("invalid %s to %s access with field %v",
↪ innerStore.TypeName(), storeDeref.TypeName(), field),
6630         )

```

```

6631     }
6632
6633     return nil, NewError(
6634         exprAccessDot.Location,
6635         ctx.NewStringf("invalid %s access with field %v", innerStore.TypeName(),
6636 ↪     field),
6637     )
6638 }
6639
6640 return expr.Eval(ctx, env)
6641 }
6642
6643 func (self *AstStatementAssignment) Eval(ctx *Context, env *Environment) (ControlFlow,
6644 ↪ error) {
6645     // Special case for identifier assignment, where we can directly update
6646     // the environment using `Environment.Set`.
6647     if lhsIdentifier, ok := self.Lhs.(*AstExpressionIdentifier); ok {
6648         rhs, err := self.Rhs.Eval(ctx, env)
6649         if err != nil {
6650             return ControlFlow{}, err
6651         }
6652         if rhsReference, ok := rhs.(*Reference); ok {
6653             return ControlFlow{}, NewError(
6654                 self.Location,
6655                 ctx.NewStringf("attempted assignment statement with type reference to
6656 ↪ %v", rhsReference.data.TypeName()),
6657             )
6658         }
6659
6660         err = env.Set(lhsIdentifier.Name.data, moveOrCopy(rhs))
6661         if err != nil {
6662             return ControlFlow{}, NewError(self.Location, ctx.NewString(err.Error()))
6663         }
6664
6665         return ControlFlow{}, nil
6666     }
6667
6668     var store Value
6669     var field Value
6670     var err error
6671
6672     if lhsAccessIndex, ok := self.Lhs.(*AstExpressionAccessIndex); ok {
6673         store, err = evalLValue(lhsAccessIndex.Store, ctx, env, nil)
6674         if err != nil {
6675             return ControlFlow{}, err
6676         }
6677         field, err = lhsAccessIndex.Field.Eval(ctx, env)
6678         if err != nil {
6679             return ControlFlow{}, err
6680         }
6681     } else if lhsAccessDot, ok := self.Lhs.(*AstExpressionAccessDot); ok {
6682         store, err = evalLValue(lhsAccessDot.Store, ctx, env, nil)
6683         if err != nil {
6684             return ControlFlow{}, err
6685         }
6686         field = lhsAccessDot.Field.Name
6687     } else if lhsAccessScope, ok := self.Lhs.(*AstExpressionAccessScope); ok {
6688         store, err = evalLValue(lhsAccessScope.Store, ctx, env, nil)
6689         if err != nil {
6690             return ControlFlow{}, err
6691         }
6692         field = lhsAccessScope.Field.Name
6693     } else {
6694         return ControlFlow{}, NewError(
6695             self.Location,
6696             ctx.NewString("attempted assignment to non-lvalue"),
6697         )
6698     }

```

```

6695     }
6696
6697     rhs, err := self.Rhs.Eval(ctx, env)
6698     if err != nil {
6699         return ControlFlow{}, err
6700     }
6701
6702     assignVector := func(storeVector *Vector) error {
6703         index, err := ValueAsIndex(field)
6704         if err != nil {
6705             return NewError(
6706                 self.Location,
6707                 ctx.NewStringf("invalid vector assignment with index %v (%s)", field,
6708 ↪ err.Error()),
6709             )
6710         }
6711         if index >= storeVector.Count() {
6712             return NewError(
6713                 self.Location,
6714                 ctx.NewStringf("invalid vector assignment with index %v (vector has a
6715 ↪ count of %v)", field, storeVector.Count()),
6716             )
6717         }
6718         err = storeVector.Set(index, moveOrCopy(rhs))
6719         if err != nil {
6720             return NewError(
6721                 self.Location,
6722                 ctx.NewStringf("invalid vector assignment with index %v (%s)", field,
6723 ↪ err.Error()),
6724             )
6725         }
6726         return nil
6727     }
6728
6729     assignMap := func(storeMap *Map) error {
6730         err := storeMap.Insert(moveOrCopy(field), moveOrCopy(rhs))
6731         if err != nil {
6732             return NewError(
6733                 self.Location,
6734                 ctx.NewStringf("invalid map assignment with key %v (%s)", field,
6735 ↪ err.Error()),
6736             )
6737         }
6738         return nil
6739     }
6740
6741     if storeVector, ok := store.(*Vector); ok {
6742         if _, lhsIsAccessIndex := self.Lhs.(*AstExpressionAccessIndex); lhsIsAccessIndex
6743 ↪ {
6744             return ControlFlow{}, assignVector(storeVector)
6745         }
6746     }
6747     if storeMap, ok := store.(*Map); ok {
6748         return ControlFlow{}, assignMap(storeMap)
6749     }
6750     if storeReference, ok := store.(*Reference); ok {
6751         storeDeref := storeReference.data
6752         if storeDerefVector, ok := storeDeref.(*Vector); ok {
6753             if _, lhsIsAccessIndex := self.Lhs.(*AstExpressionAccessIndex);
6754 ↪ lhsIsAccessIndex {
6755                 return ControlFlow{}, assignVector(storeDerefVector)
6756             }
6757         }
6758     }

```

```

6756         if storeDerefMap, ok := storeDeref.(*Map); ok {
6757             return ControlFlow{}, assignMap(storeDerefMap)
6758         }
6759         return ControlFlow{}, NewError(
6760             self.Location,
6761             ctx.NewStringf("invalid %s to %s access with field %v", store.TypeName(),
↪ storeDeref.TypeName(), field),
6762         )
6763     }
6764
6765     return ControlFlow{}, NewError(
6766         self.Location,
6767         ctx.NewStringf("attempted access into type %s with type %s",
↪ quote(store.TypeName()), quote(field.TypeName()))),
6768     )
6769 }
6770
6771 type AstStatementExpression struct {
6772     Location *SourceLocation // Optional
6773     Expression AstExpression
6774 }
6775
6776 func (self AstStatementExpression) StatementLocation() *SourceLocation {
6777     return self.Location
6778 }
6779
6780 func (self AstStatementExpression) IntoValue(ctx *Context) Value {
6781     return ctx.NewMapOrPanic([]MapPair{
6782         {ctx.NewString("kind"), ctx.NewString(reflect.TypeOf(self).Name())},
6783         {ctx.NewString("location"), optionalSourceLocationIntoValue(ctx, self.Location)},
6784         {ctx.NewString("expression"), self.Expression.IntoValue(ctx)},
6785     })
6786 }
6787
6788 func (self *AstStatementExpression) Eval(ctx *Context, env *Environment) (ControlFlow,
↪ error) {
6789     _, error := self.Expression.Eval(ctx, env)
6790     if error != nil {
6791         return ControlFlow{}, error
6792     }
6793     return ControlFlow{}, nil
6794 }
6795
6796 // Precedence Levels
6797 const (
6798     PRECEDENCE_LOWEST = iota
6799     PRECEDENCE_OR     = iota // or
6800     PRECEDENCE_AND     = iota // and
6801     PRECEDENCE_COMPARE = iota // == != <= >= < > =~ !~
6802     PRECEDENCE_ADD_SUB = iota // + -
6803     PRECEDENCE_MUL_DIV = iota // * / %
6804     PRECEDENCE_PREFIX  = iota // +x -x
6805     PRECEDENCE_POSTFIX  = iota // foo(bar, 123) foo[42] .& .*
6806 )
6807
6808 type Parser struct {
6809     lexer *Lexer
6810     currentToken Token
6811
6812     precedences map[string]int
6813     parseNudFunctions map[string]func(*Parser) (AstExpression, error)
6814     parseLedFunctions map[string]func(*Parser, AstExpression) (AstExpression, error)
6815 }
6816
6817 func NewParser(lexer *Lexer) (Parser, error) {
6818     self := Parser{
6819         lexer: lexer,

```

```

6820     currentToken: Token{"invalid program", "", lexer.location, nil, nil, 0},
6821
6822     precedences: map[string]int{
6823         TOKEN_OR:      PRECEDENCE_OR,
6824         TOKEN_AND:     PRECEDENCE_AND,
6825         TOKEN_EQ:      PRECEDENCE_COMPARE,
6826         TOKEN_NE:      PRECEDENCE_COMPARE,
6827         TOKEN_LE:      PRECEDENCE_COMPARE,
6828         TOKEN_GE:      PRECEDENCE_COMPARE,
6829         TOKEN_LT:      PRECEDENCE_COMPARE,
6830         TOKEN_GT:      PRECEDENCE_COMPARE,
6831         TOKEN_EQ_RE:   PRECEDENCE_COMPARE,
6832         TOKEN_NE_RE:   PRECEDENCE_COMPARE,
6833         TOKEN_ADD:     PRECEDENCE_ADD_SUB,
6834         TOKEN_SUB:     PRECEDENCE_ADD_SUB,
6835         TOKEN_MUL:     PRECEDENCE_MUL_DIV,
6836         TOKEN_DIV:     PRECEDENCE_MUL_DIV,
6837         TOKEN_REM:     PRECEDENCE_MUL_DIV,
6838         TOKEN_LPAREN:  PRECEDENCE_POSTFIX,
6839         TOKEN_LBRACKET: PRECEDENCE_POSTFIX,
6840         TOKEN_SCOPE:   PRECEDENCE_POSTFIX,
6841         TOKEN_DOT:     PRECEDENCE_POSTFIX,
6842         TOKEN_MKREF:   PRECEDENCE_POSTFIX,
6843         TOKEN_DEREF:   PRECEDENCE_POSTFIX,
6844     },
6845     parseNudFunctions: map[string]func(*Parser) (AstExpression, error){
6846         TOKEN_IDENTIFIER: (*Parser).ParseExpressionIdentifier,
6847         TOKEN_TEMPLATE:   (*Parser).ParseExpressionTemplate,
6848         TOKEN_NULL:       (*Parser).ParseExpressionNull,
6849         TOKEN_TRUE:       (*Parser).ParseExpressionBoolean,
6850         TOKEN_FALSE:      (*Parser).ParseExpressionBoolean,
6851         TOKEN_NUMBER:     (*Parser).ParseExpressionNumber,
6852         TOKEN_STRING:     (*Parser).ParseExpressionString,
6853         TOKEN_REGEXP:      (*Parser).ParseExpressionRegexp,
6854         TOKEN_REGEXP_GROUP: (*Parser).ParseExpressionRegexpGroup,
6855         TOKEN_LBRACKET:    (*Parser).ParseExpressionVector,
6856         TOKEN_MAP:         (*Parser).ParseExpressionMapOrSet,
6857         TOKEN_SET:         (*Parser).ParseExpressionMapOrSet,
6858         TOKEN_LBRACE:      (*Parser).ParseExpressionMapOrSet,
6859         TOKEN_FUNCTION:   (*Parser).ParseExpressionFunction,
6860         TOKEN_FREEZE:     (*Parser).ParseExpressionFreeze,
6861         TOKEN_TYPE:       (*Parser).ParseExpressionType,
6862         TOKEN_NEW:        (*Parser).ParseExpressionNew,
6863         TOKEN_LPAREN:     (*Parser).ParseExpressionGrouped,
6864         TOKEN_ADD:        (*Parser).ParseExpressionPositive,
6865         TOKEN_SUB:        (*Parser).ParseExpressionNegative,
6866         TOKEN_NOT:        (*Parser).ParseExpressionNot,
6867     },
6868     parseLedFunctions: map[string]func(*Parser, AstExpression) (AstExpression,
6869 ↪ error){
6869         TOKEN_AND:      (*Parser).ParseExpressionAnd,
6870         TOKEN_OR:        (*Parser).ParseExpressionOr,
6871         TOKEN_EQ:        (*Parser).ParseExpressionEq,
6872         TOKEN_NE:        (*Parser).ParseExpressionNe,
6873         TOKEN_LE:        (*Parser).ParseExpressionLe,
6874         TOKEN_GE:        (*Parser).ParseExpressionGe,
6875         TOKEN_LT:        (*Parser).ParseExpressionLt,
6876         TOKEN_GT:        (*Parser).ParseExpressionGt,
6877         TOKEN_EQ_RE:     (*Parser).ParseExpressionEqRe,
6878         TOKEN_NE_RE:     (*Parser).ParseExpressionNeRe,
6879         TOKEN_ADD:       (*Parser).ParseExpressionAdd,
6880         TOKEN_SUB:       (*Parser).ParseExpressionSub,
6881         TOKEN_MUL:       (*Parser).ParseExpressionMul,
6882         TOKEN_DIV:       (*Parser).ParseExpressionDiv,
6883         TOKEN_REM:       (*Parser).ParseExpressionRem,
6884         TOKEN_LPAREN:    (*Parser).ParseExpressionFunctionCall,
6885         TOKEN_LBRACKET:  (*Parser).ParseExpressionAccessIndex,

```

```

6886         TOKEN_SCOPE:    (*Parser).ParseExpressionAccessScope,
6887         TOKEN_DOT:       (*Parser).ParseExpressionAccessDot,
6888         TOKEN_DEREF:     (*Parser).ParseExpressionDeref,
6889     },
6890 }
6891
6892 _, err := self.advanceToken()
6893 if err != nil {
6894     return Parser{}, err
6895 }
6896
6897 return self, err
6898 }
6899
6900 func (self *Parser) context() *Context {
6901     return self.lexer.ctx
6902 }
6903
6904 func (self *Parser) advanceToken() (Token, error) {
6905     current := self.currentToken
6906     token, err := self.lexer.NextToken()
6907     if err != nil {
6908         return token, err
6909     }
6910     self.currentToken = token
6911     return current, nil
6912 }
6913
6914 func (self *Parser) checkCurrent(kind string) bool {
6915     return self.currentToken.Kind == kind
6916 }
6917
6918 func (self *Parser) expectCurrent(kind string) (Token, error) {
6919     current := self.currentToken
6920     if current.Kind != kind {
6921         return Token{}, ParseError{
6922             current.Location,
6923             fmt.Sprintf("expected %s, found %s", quote(kind), quote(current.String()))),
6924         }
6925     }
6926     if _, err := self.advanceToken(); err != nil {
6927         return Token{}, err
6928     }
6929     return current, nil
6930 }
6931
6932 func (self *Parser) identifier(name string) *String {
6933     cached, ok := self.lexer.ctx.identifierCache[name]
6934     if ok {
6935         return cached
6936     }
6937     self.lexer.ctx.identifierCache[name] = self.lexer.ctx.NewString(name)
6938     return self.lexer.ctx.identifierCache[name]
6939 }
6940
6941 func (self *Parser) ParseProgram() (*AstProgram, error) {
6942     location := self.currentToken.Location
6943     statements := []AstStatement{}
6944     for !self.checkCurrent(TOKEN_EOF) {
6945         statement, err := self.ParseStatement()
6946         if err != nil {
6947             return &AstProgram{}, err
6948         }
6949         statements = append(statements, statement)
6950     }
6951     return &AstProgram{location, statements}, nil
6952 }

```

```

6953
6954 func (self *Parser) ParseIdentifier() (*AstIdentifier, error) {
6955     token, err := self.expectCurrent(TOKEN_IDENTIFIER)
6956     if err != nil {
6957         return &AstIdentifier{}, err
6958     }
6959     return &AstIdentifier{token.Location, self.identifier(token.Literal)}, nil
6960 }
6961
6962 func (self *Parser) ParseExpression() (AstExpression, error) {
6963     return self.parseExpression(PRECEDENCE_LOWEST)
6964 }
6965
6966 func (self *Parser) parseExpression(precedence int) (AstExpression, error) {
6967     parseNud, ok := self.parseNudFunctions[self.currentToken.Kind]
6968     if !ok {
6969         return nil, ParseError{
6970             self.currentToken.Location,
6971             fmt.Sprintf("expected expression, found %v", self.currentToken),
6972         }
6973     }
6974     expression, err := parseNud(self)
6975     if err != nil {
6976         return nil, err
6977     }
6978
6979     getPrecedence := func(kind string) int {
6980         precedence, ok := self.precedences[kind]
6981         if !ok {
6982             return PRECEDENCE_LOWEST
6983         }
6984         return precedence
6985     }
6986
6987     for precedence < getPrecedence(self.currentToken.Kind) {
6988         parseLed, ok := self.parseLedFunctions[self.currentToken.Kind]
6989         if !ok {
6990             return expression, nil
6991         }
6992         expression, err = parseLed(self, expression)
6993         if err != nil {
6994             return nil, err
6995         }
6996     }
6997
6998     return expression, nil
6999 }
7000
7001 func (self *Parser) ParseExpressionIdentifier() (AstExpression, error) {
7002     token, err := self.expectCurrent(TOKEN_IDENTIFIER)
7003     if err != nil {
7004         return nil, err
7005     }
7006     return &AstExpressionIdentifier{token.Location, self.identifier(token.Literal)}, nil
7007 }
7008
7009 func (self *Parser) ParseExpressionTemplate() (AstExpression, error) {
7010     token, err := self.expectCurrent(TOKEN_TEMPLATE)
7011     if err != nil {
7012         return nil, err
7013     }
7014     return &AstExpressionTemplate{token.Location, token.Template}, nil
7015 }
7016
7017 func (self *Parser) ParseExpressionNull() (AstExpression, error) {
7018     token, err := self.expectCurrent(TOKEN_NULL)
7019     if err != nil {

```

```

7020         return nil, err
7021     }
7022     return &AstExpressionNull{token.Location}, nil
7023 }
7024
7025 func (self *Parser) ParseExpressionBoolean() (AstExpression, error) {
7026     if self.checkCurrent(TOKEN_TRUE) {
7027         token, err := self.expectCurrent(TOKEN_TRUE)
7028         if err != nil {
7029             return nil, err
7030         }
7031         return &AstExpressionBoolean{token.Location, self.context().NewBoolean(true)},
7032     }
7033     if self.checkCurrent(TOKEN_FALSE) {
7034         token, err := self.expectCurrent(TOKEN_FALSE)
7035         if err != nil {
7036             return nil, err
7037         }
7038         return &AstExpressionBoolean{token.Location, self.context().NewBoolean(false)},
7039     }
7040     return nil, ParseError{
7041         self.currentToken.Location,
7042         fmt.Sprintf("expected boolean, found %v", self.currentToken),
7043     }
7044 }
7045
7046 func (self *Parser) ParseExpressionNumber() (AstExpression, error) {
7047     token, err := self.expectCurrent(TOKEN_NUMBER)
7048     if err != nil {
7049         return nil, ParseError{
7050             self.currentToken.Location,
7051             fmt.Sprintf("expected number, found %v", self.currentToken),
7052         }
7053     }
7054     value, ok := token.Value.(*Number)
7055     if !ok {
7056         return nil, errors.New("missing number token value")
7057     }
7058     return &AstExpressionNumber{token.Location, value}, nil
7059 }
7060
7061 func (self *Parser) ParseExpressionString() (AstExpression, error) {
7062     token, err := self.expectCurrent(TOKEN_STRING)
7063     if err != nil {
7064         return nil, ParseError{
7065             self.currentToken.Location,
7066             fmt.Sprintf("expected string, found %v", self.currentToken),
7067         }
7068     }
7069     value, ok := token.Value.(*String)
7070     if !ok {
7071         return nil, errors.New("missing string token value")
7072     }
7073     return &AstExpressionString{token.Location, value}, nil
7074 }
7075
7076 func (self *Parser) ParseExpressionRegex() (AstExpression, error) {
7077     token, err := self.expectCurrent(TOKEN_REGEX)
7078     if err != nil {
7079         return nil, ParseError{
7080             self.currentToken.Location,
7081             fmt.Sprintf("expected regexp, found %v", self.currentToken),
7082         }
7083     }
7084     value, ok := token.Value.(*Regex)

```

```

7085     if !ok {
7086         return nil, errors.New("missing regexp token value")
7087     }
7088     return &AstExpressionRegexp{token.Location, value}, nil
7089 }
7090
7091 func (self *Parser) ParseExpressionRegexpGroup() (AstExpression, error) {
7092     token, err := self.expectCurrent(TOKEN_REGEXP_GROUP)
7093     if err != nil {
7094         return nil, ParseError{
7095             self.currentToken.Location,
7096             fmt.Sprintf("expected regexp group, found %v", self.currentToken),
7097         }
7098     }
7099     return &AstExpressionRegexpGroup{token.Location, token.Group}, nil
7100 }
7101
7102 func (self *Parser) ParseExpressionVector() (AstExpression, error) {
7103     token, err := self.expectCurrent(TOKEN_LBRACKET)
7104     if err != nil {
7105         return nil, err
7106     }
7107     location := token.Location
7108
7109     elements := []AstExpression{}
7110     for !self.checkCurrent(TOKEN_RBRACKET) {
7111         if len(elements) != 0 {
7112             if _, err := self.expectCurrent(TOKEN_COMMA); err != nil {
7113                 return nil, err
7114             }
7115         }
7116         if self.checkCurrent(TOKEN_RBRACKET) {
7117             break
7118         }
7119
7120         expression, err := self.ParseExpression()
7121         if err != nil {
7122             return nil, err
7123         }
7124         elements = append(elements, expression)
7125     }
7126
7127     if _, err := self.expectCurrent(TOKEN_RBRACKET); err != nil {
7128         return nil, err
7129     }
7130
7131     return &AstExpressionVector{location, elements}, nil
7132 }
7133
7134 func (self *Parser) ParseExpressionMapOrSet() (AstExpression, error) {
7135     var mapOrSet string = "UNKNOWN"
7136     var location *SourceLocation
7137     if self.checkCurrent(TOKEN_MAP) {
7138         mapOrSet = TOKEN_MAP
7139         location = self.currentToken.Location
7140         _, err := self.advanceToken()
7141         if err != nil {
7142             return nil, err
7143         }
7144         _, err = self.expectCurrent(TOKEN_LBRACE)
7145         if err != nil {
7146             return nil, err
7147         }
7148     } else if self.checkCurrent(TOKEN_SET) {
7149         mapOrSet = TOKEN_SET
7150         location = self.currentToken.Location
7151         _, err := self.advanceToken()

```

```

7152         if err != nil {
7153             return nil, err
7154         }
7155         _, err = self.expectCurrent(TOKEN_LBRACE)
7156         if err != nil {
7157             return nil, err
7158         }
7159     } else {
7160         token, err := self.expectCurrent(TOKEN_LBRACE)
7161         if err != nil {
7162             return nil, err
7163         }
7164         location = token.Location
7165     }
7166
7167     var mapElements []AstMapPair
7168     var setElements []AstExpression
7169     for !self.checkCurrent(TOKEN_RBRACE) {
7170         if len(mapElements) != 0 || len(setElements) != 0 {
7171             if _, err := self.expectCurrent(TOKEN_COMMA); err != nil {
7172                 return nil, err
7173             }
7174         }
7175         if self.checkCurrent(TOKEN_RBRACE) {
7176             break
7177         }
7178
7179         var expression AstExpression = nil
7180         var identifier *AstExpressionIdentifier = nil
7181         if self.checkCurrent(TOKEN_DOT) {
7182             if mapOrSet == "UNKNOWN" {
7183                 mapOrSet = TOKEN_MAP
7184             } else if mapOrSet == TOKEN_SET {
7185                 return nil, ParseError{
7186                     Location: location,
7187                     why:      fmt.Sprintf("expected expression, found %s",
↪ self.currentToken.Kind),
7188                 }
7189             }
7190             _, err := self.expectCurrent(TOKEN_DOT)
7191             if err != nil {
7192                 return nil, err
7193             }
7194             expression, err = self.ParseExpressionIdentifier()
7195             if err != nil {
7196                 return nil, err
7197             }
7198             identifier = expression.(*AstExpressionIdentifier)
7199             expression = &AstExpressionString{identifier.Location, identifier.Name}
7200         } else {
7201             element, err := self.ParseExpression()
7202             if err != nil {
7203                 return nil, err
7204             }
7205             expression = element
7206         }
7207
7208         if mapOrSet == "UNKNOWN" {
7209             if self.checkCurrent(TOKEN_COLON) || self.checkCurrent(TOKEN_ASSIGN) {
7210                 mapOrSet = TOKEN_MAP
7211             } else {
7212                 mapOrSet = TOKEN_SET
7213             }
7214         }
7215
7216         if mapOrSet == TOKEN_MAP {
7217             if self.checkCurrent(TOKEN_COLON) {

```

```

7218         if identifier != nil {
7219             return nil, ParseError{
7220                 Location: location,
7221                 why:      fmt.Sprintf(`expected .%s = VALUE, found .%s: VALUE`,
↪ identifier.Name.data, identifier.Name.data),
7222             }
7223         }
7224         _, err := self.expectCurrent(TOKEN_COLON)
7225         if err != nil {
7226             return nil, err
7227         }
7228     } else if self.checkCurrent(TOKEN_ASSIGN) {
7229         if identifier == nil {
7230             return nil, ParseError{
7231                 Location: location,
7232                 why:      `expected "KEY": VALUE, found "KEY" = VALUE`,
7233             }
7234         }
7235         _, err := self.expectCurrent(TOKEN_ASSIGN)
7236         if err != nil {
7237             return nil, err
7238         }
7239     } else {
7240         return nil, ParseError{
7241             Location: location,
7242             why:      fmt.Sprintf("expected %s or %s, found %s", TOKEN_COLON,
↪ TOKEN_ASSIGN, self.currentToken.Kind),
7243         }
7244     }
7245
7246     key := expression
7247     value, err := self.ParseExpression()
7248     if err != nil {
7249         return nil, err
7250     }
7251     mapElements = append(mapElements, AstMapPair{key, value})
7252 } else if mapOrSet == TOKEN_SET {
7253     setElements = append(setElements, expression)
7254 }
7255 }
7256
7257 if _, err := self.expectCurrent(TOKEN_RBRACE); err != nil {
7258     return nil, err
7259 }
7260
7261 if mapOrSet == TOKEN_MAP {
7262     result := &AstExpressionMap{location, mapElements}
7263     updateNamedFunctions(self.lexer.ctx, result, "")
7264     return result, nil
7265 }
7266 if mapOrSet == TOKEN_SET {
7267     return &AstExpressionSet{location, setElements}, nil
7268 }
7269 return nil, ParseError{
7270     Location: location,
7271     why:      "ambiguous empty map or set",
7272 }
7273 }
7274
7275 func (self *Parser) ParseExpressionFunction() (AstExpression, error) {
7276     token, err := self.expectCurrent(TOKEN_FUNCTION)
7277     if err != nil {
7278         return nil, err
7279     }
7280     location := token.Location
7281
7282     selfByReference := false

```

```

7283     if self.checkCurrent(TOKEN_MKREF) {
7284         _, err = self.advanceToken()
7285         if err != nil {
7286             return nil, err
7287         }
7288         selfByReference = true
7289     }
7290
7291     _, err = self.expectCurrent(TOKEN_LPAREN)
7292     if err != nil {
7293         return nil, err
7294     }
7295
7296     parameters := []*AstIdentifier{}
7297     for !self.checkCurrent(TOKEN_RPAREN) {
7298         if len(parameters) != 0 {
7299             if _, err := self.expectCurrent(TOKEN_COMMA); err != nil {
7300                 return nil, err
7301             }
7302         }
7303
7304         identifier, err := self.ParseIdentifier()
7305         if err != nil {
7306             return nil, err
7307         }
7308         parameters = append(parameters, identifier)
7309     }
7310
7311     if _, err := self.expectCurrent(TOKEN_RPAREN); err != nil {
7312         return nil, err
7313     }
7314
7315     body, err := self.ParseBlock()
7316     if err != nil {
7317         return nil, err
7318     }
7319
7320     for i := range parameters {
7321         for j := i + 1; j < len(parameters); j++ {
7322             if parameters[i].Name == parameters[j].Name {
7323                 return nil, ParseError{
7324                     Location: location,
7325                     why:      fmt.Sprintf("duplicate function parameter %s",
7326 ↪ quote(parameters[i].Name.data)),
7327                 }
7328             }
7329         }
7330     }
7331     if selfByReference && len(parameters) == 0 {
7332         return nil, ParseError{
7333             Location: location,
7334             why:      "function takes self by reference, but has no parameters",
7335         }
7336     }
7337     return &AstExpressionFunction{location, parameters, body, nil, selfByReference}, nil
7338 }
7339
7340 func (self *Parser) ParseExpressionFreeze() (AstExpression, error) {
7341     token, err := self.expectCurrent(TOKEN_FREEZE)
7342     if err != nil {
7343         return nil, err
7344     }
7345     location := token.Location
7346
7347     expression, err := self.parseExpression(PRECEDENCE_PREFIX)
7348     if err != nil {

```

```

7349         return nil, err
7350     }
7351
7352     return &AstExpressionFreeze{location, expression}, nil
7353 }
7354
7355 func (self *Parser) ParseExpressionType() (AstExpression, error) {
7356     token, err := self.expectCurrent(TOKEN_TYPE)
7357     if err != nil {
7358         return nil, err
7359     }
7360     location := token.Location
7361
7362     var extends AstExpression = nil
7363     if self.checkCurrent(TOKEN_EXTENDS) {
7364         _, err := self.advanceToken()
7365         if err != nil {
7366             return nil, err
7367         }
7368         extends, err = self.ParseExpression()
7369         if err != nil {
7370             return nil, err
7371         }
7372     }
7373
7374     expression, err := self.ParseExpression()
7375     if err != nil {
7376         return nil, err
7377     }
7378
7379     name := "anonymous type"
7380     if location != nil {
7381         name = fmt.Sprintf("type@[%s, line %v]", location.File, location.Line)
7382     }
7383
7384     return &AstExpressionType{location, name, extends, expression}, nil
7385 }
7386
7387 func (self *Parser) ParseExpressionNew() (AstExpression, error) {
7388     token, err := self.expectCurrent(TOKEN_NEW)
7389     if err != nil {
7390         return nil, err
7391     }
7392     location := token.Location
7393
7394     meta, err := self.ParseExpression()
7395     if err != nil {
7396         return nil, err
7397     }
7398
7399     expression, err := self.ParseExpression()
7400     if err != nil {
7401         return nil, err
7402     }
7403
7404     return &AstExpressionNew{location, meta, expression}, nil
7405 }
7406
7407 func (self *Parser) ParseExpressionGrouped() (AstExpression, error) {
7408     token, err := self.expectCurrent(TOKEN_LPAREN)
7409     if err != nil {
7410         return nil, err
7411     }
7412     location := token.Location
7413
7414     expression, err := self.ParseExpression()
7415     if err != nil {

```

```

7416         return nil, err
7417     }
7418
7419     _, err = self.expectCurrent(TOKEN_RPAREN)
7420     if err != nil {
7421         return nil, err
7422     }
7423
7424     return &AstExpressionGrouped{location, expression}, nil
7425 }
7426
7427 func (self *Parser) ParseExpressionPositive() (AstExpression, error) {
7428     token, err := self.expectCurrent(TOKEN_ADD)
7429     if err != nil {
7430         return nil, err
7431     }
7432     location := token.Location
7433
7434     expression, err := self.parseExpression(PRECEDENCE_PREFIX)
7435     if err != nil {
7436         return nil, err
7437     }
7438
7439     return &AstExpressionPositive{location, expression}, nil
7440 }
7441
7442 func (self *Parser) ParseExpressionNegative() (AstExpression, error) {
7443     token, err := self.expectCurrent(TOKEN_SUB)
7444     if err != nil {
7445         return nil, err
7446     }
7447     location := token.Location
7448
7449     expression, err := self.parseExpression(PRECEDENCE_PREFIX)
7450     if err != nil {
7451         return nil, err
7452     }
7453
7454     return &AstExpressionNegative{location, expression}, nil
7455 }
7456
7457 func (self *Parser) ParseExpressionNot() (AstExpression, error) {
7458     token, err := self.expectCurrent(TOKEN_NOT)
7459     if err != nil {
7460         return nil, err
7461     }
7462     location := token.Location
7463
7464     expression, err := self.parseExpression(PRECEDENCE_PREFIX)
7465     if err != nil {
7466         return nil, err
7467     }
7468
7469     return &AstExpressionNot{location, expression}, nil
7470 }
7471
7472 func (self *Parser) ParseExpressionAnd(lhs AstExpression) (AstExpression, error) {
7473     token, err := self.expectCurrent(TOKEN_AND)
7474     if err != nil {
7475         return nil, err
7476     }
7477     location := token.Location
7478
7479     rhs, err := self.parseExpression(self.precedences[TOKEN_AND])
7480     if err != nil {
7481         return nil, err
7482     }

```

```

7483
7484     return &AstExpressionAnd{location, lhs, rhs}, nil
7485 }
7486
7487 func (self *Parser) ParseExpressionOr(lhs AstExpression) (AstExpression, error) {
7488     token, err := self.expectCurrent(TOKEN_OR)
7489     if err != nil {
7490         return nil, err
7491     }
7492     location := token.Location
7493
7494     rhs, err := self.parseExpression(self.precedences[TOKEN_OR])
7495     if err != nil {
7496         return nil, err
7497     }
7498
7499     return &AstExpressionOr{location, lhs, rhs}, nil
7500 }
7501
7502 func (self *Parser) ParseExpressionEq(lhs AstExpression) (AstExpression, error) {
7503     token, err := self.expectCurrent(TOKEN_EQ)
7504     if err != nil {
7505         return nil, err
7506     }
7507     location := token.Location
7508
7509     rhs, err := self.parseExpression(self.precedences[TOKEN_EQ])
7510     if err != nil {
7511         return nil, err
7512     }
7513
7514     return &AstExpressionEq{location, lhs, rhs}, nil
7515 }
7516
7517 func (self *Parser) ParseExpressionNe(lhs AstExpression) (AstExpression, error) {
7518     token, err := self.expectCurrent(TOKEN_NE)
7519     if err != nil {
7520         return nil, err
7521     }
7522     location := token.Location
7523
7524     rhs, err := self.parseExpression(self.precedences[TOKEN_NE])
7525     if err != nil {
7526         return nil, err
7527     }
7528
7529     return &AstExpressionNe{location, lhs, rhs}, nil
7530 }
7531
7532 func (self *Parser) ParseExpressionLe(lhs AstExpression) (AstExpression, error) {
7533     token, err := self.expectCurrent(TOKEN_LE)
7534     if err != nil {
7535         return nil, err
7536     }
7537     location := token.Location
7538
7539     rhs, err := self.parseExpression(self.precedences[TOKEN_LE])
7540     if err != nil {
7541         return nil, err
7542     }
7543
7544     return &AstExpressionLe{location, lhs, rhs}, nil
7545 }
7546
7547 func (self *Parser) ParseExpressionGe(lhs AstExpression) (AstExpression, error) {
7548     token, err := self.expectCurrent(TOKEN_GE)
7549     if err != nil {

```

```

7550         return nil, err
7551     }
7552     location := token.Location
7553
7554     rhs, err := self.parseExpression(self.precedences[TOKEN_GE])
7555     if err != nil {
7556         return nil, err
7557     }
7558
7559     return &AstExpressionGe{location, lhs, rhs}, nil
7560 }
7561
7562 func (self *Parser) ParseExpressionLt(lhs AstExpression) (AstExpression, error) {
7563     token, err := self.expectCurrent(TOKEN_LT)
7564     if err != nil {
7565         return nil, err
7566     }
7567     location := token.Location
7568
7569     rhs, err := self.parseExpression(self.precedences[TOKEN_LT])
7570     if err != nil {
7571         return nil, err
7572     }
7573
7574     return &AstExpressionLt{location, lhs, rhs}, nil
7575 }
7576
7577 func (self *Parser) ParseExpressionGt(lhs AstExpression) (AstExpression, error) {
7578     token, err := self.expectCurrent(TOKEN_GT)
7579     if err != nil {
7580         return nil, err
7581     }
7582     location := token.Location
7583
7584     rhs, err := self.parseExpression(self.precedences[TOKEN_GT])
7585     if err != nil {
7586         return nil, err
7587     }
7588
7589     return &AstExpressionGt{location, lhs, rhs}, nil
7590 }
7591
7592 func (self *Parser) ParseExpressionEqRe(lhs AstExpression) (AstExpression, error) {
7593     token, err := self.expectCurrent(TOKEN_EQ_RE)
7594     if err != nil {
7595         return nil, err
7596     }
7597     location := token.Location
7598
7599     rhs, err := self.parseExpression(self.precedences[TOKEN_EQ_RE])
7600     if err != nil {
7601         return nil, err
7602     }
7603
7604     return &AstExpressionEqRe{location, lhs, rhs}, nil
7605 }
7606
7607 func (self *Parser) ParseExpressionNeRe(lhs AstExpression) (AstExpression, error) {
7608     token, err := self.expectCurrent(TOKEN_NE_RE)
7609     if err != nil {
7610         return nil, err
7611     }
7612     location := token.Location
7613
7614     rhs, err := self.parseExpression(self.precedences[TOKEN_NE_RE])
7615     if err != nil {
7616         return nil, err

```

```

7617     }
7618
7619     return &AstExpressionNeRe{location, lhs, rhs}, nil
7620 }
7621
7622 func (self *Parser) ParseExpressionAdd(lhs AstExpression) (AstExpression, error) {
7623     token, err := self.expectCurrent(TOKEN_ADD)
7624     if err != nil {
7625         return nil, err
7626     }
7627     location := token.Location
7628
7629     rhs, err := self.parseExpression(self.precedences[TOKEN_ADD])
7630     if err != nil {
7631         return nil, err
7632     }
7633
7634     return &AstExpressionAdd{location, lhs, rhs}, nil
7635 }
7636
7637 func (self *Parser) ParseExpressionSub(lhs AstExpression) (AstExpression, error) {
7638     token, err := self.expectCurrent(TOKEN_SUB)
7639     if err != nil {
7640         return nil, err
7641     }
7642     location := token.Location
7643
7644     rhs, err := self.parseExpression(self.precedences[TOKEN_SUB])
7645     if err != nil {
7646         return nil, err
7647     }
7648
7649     return &AstExpressionSub{location, lhs, rhs}, nil
7650 }
7651
7652 func (self *Parser) ParseExpressionMul(lhs AstExpression) (AstExpression, error) {
7653     token, err := self.expectCurrent(TOKEN_MUL)
7654     if err != nil {
7655         return nil, err
7656     }
7657     location := token.Location
7658
7659     rhs, err := self.parseExpression(self.precedences[TOKEN_MUL])
7660     if err != nil {
7661         return nil, err
7662     }
7663
7664     return &AstExpressionMul{location, lhs, rhs}, nil
7665 }
7666
7667 func (self *Parser) ParseExpressionDiv(lhs AstExpression) (AstExpression, error) {
7668     token, err := self.expectCurrent(TOKEN_DIV)
7669     if err != nil {
7670         return nil, err
7671     }
7672     location := token.Location
7673
7674     rhs, err := self.parseExpression(self.precedences[TOKEN_DIV])
7675     if err != nil {
7676         return nil, err
7677     }
7678
7679     return &AstExpressionDiv{location, lhs, rhs}, nil
7680 }
7681
7682 func (self *Parser) ParseExpressionRem(lhs AstExpression) (AstExpression, error) {
7683     token, err := self.expectCurrent(TOKEN_REM)

```

```

7684     if err != nil {
7685         return nil, err
7686     }
7687     location := token.Location
7688
7689     rhs, err := self.parseExpression(self.precedences[TOKEN_REM])
7690     if err != nil {
7691         return nil, err
7692     }
7693
7694     return &AstExpressionRem{location, lhs, rhs}, nil
7695 }
7696
7697 func (self *Parser) ParseExpressionFunctionCall(lhs AstExpression) (AstExpression, error)
7698 ↪ {
7699     token, err := self.expectCurrent(TOKEN_LPAREN)
7700     if err != nil {
7701         return nil, err
7702     }
7703     location := token.Location
7704
7705     arguments := []AstExpression{}
7706     for !self.checkCurrent(TOKEN_RPAREN) {
7707         if len(arguments) != 0 {
7708             if _, err := self.expectCurrent(TOKEN_COMMA); err != nil {
7709                 return nil, err
7710             }
7711             if self.checkCurrent(TOKEN_RPAREN) {
7712                 break
7713             }
7714
7715             expression, err := self.ParseExpression()
7716             if err != nil {
7717                 return nil, err
7718             }
7719             if self.checkCurrent(TOKEN_MKREF) {
7720                 token, err = self.advanceToken()
7721                 if err != nil {
7722                     return nil, err
7723                 }
7724                 expression = &AstExpressionMkref{token.Location, expression}
7725             }
7726             arguments = append(arguments, expression)
7727         }
7728
7729         if _, err := self.expectCurrent(TOKEN_RPAREN); err != nil {
7730             return nil, err
7731         }
7732
7733         return &AstExpressionFunctionCall{location, lhs, arguments}, nil
7734     }
7735
7736 func (self *Parser) ParseExpressionAccessIndex(lhs AstExpression) (AstExpression, error)
7737 ↪ {
7738     token, err := self.expectCurrent(TOKEN_LBRACKET)
7739     if err != nil {
7740         return nil, err
7741     }
7742     field, err := self.ParseExpression()
7743     if err != nil {
7744         return nil, err
7745     }
7746     _, err = self.expectCurrent(TOKEN_RBRACKET)
7747     if err != nil {
7748         return nil, err
7749     }

```

```

7749     return &AstExpressionAccessIndex{token.Location, lhs, field}, nil
7750 }
7751
7752 func (self *Parser) ParseExpressionAccessScope(lhs AstExpression) (AstExpression, error)
7753 ↪ {
7754     token, err := self.expectCurrent(TOKEN_SCOPE)
7755     if err != nil {
7756         return nil, err
7757     }
7758     field, err := self.ParseIdentifier()
7759     if err != nil {
7760         return nil, err
7761     }
7762     return &AstExpressionAccessScope{token.Location, lhs, field}, nil
7763 }
7764
7765 func (self *Parser) ParseExpressionAccessDot(lhs AstExpression) (AstExpression, error) {
7766     token, err := self.expectCurrent(TOKEN_DOT)
7767     if err != nil {
7768         return nil, err
7769     }
7770     field, err := self.ParseIdentifier()
7771     if err != nil {
7772         return nil, err
7773     }
7774     return &AstExpressionAccessDot{token.Location, lhs, field}, nil
7775 }
7776
7777 func (self *Parser) ParseExpressionDeref(lhs AstExpression) (AstExpression, error) {
7778     token, err := self.expectCurrent(TOKEN_DEREF)
7779     if err != nil {
7780         return nil, err
7781     }
7782     return &AstExpressionDeref{token.Location, lhs}, nil
7783 }
7784
7785 func (self *Parser) ParseBlock() (*AstBlock, error) {
7786     token, err := self.expectCurrent(TOKEN_LBRACE)
7787     if err != nil {
7788         return &AstBlock{}, err
7789     }
7790
7791     statements := []AstStatement{}
7792     for !self.checkCurrent(TOKEN_RBRACE) {
7793         statement, err := self.ParseStatement()
7794         if err != nil {
7795             return &AstBlock{}, err
7796         }
7797         statements = append(statements, statement)
7798     }
7799
7800     _, err = self.expectCurrent(TOKEN_RBRACE)
7801     if err != nil {
7802         return &AstBlock{}, err
7803     }
7804
7805     return &AstBlock{token.Location, statements}, nil
7806 }
7807
7808 func (self *Parser) ParseStatement() (AstStatement, error) {
7809     if self.checkCurrent(TOKEN_LET) {
7810         return self.ParseStatementLet()
7811     }
7812
7813     if self.checkCurrent(TOKEN_IF) {
7814         return self.ParseStatementIfElse()
7815     }

```

```

7815
7816     if self.checkCurrent(TOKEN_FOR) {
7817         return self.ParseStatementFor()
7818     }
7819
7820     if self.checkCurrent(TOKEN_WHILE) {
7821         return self.ParseStatementWhile()
7822     }
7823
7824     if self.checkCurrent(TOKEN_BREAK) {
7825         return self.ParseStatementBreak()
7826     }
7827
7828     if self.checkCurrent(TOKEN_CONTINUE) {
7829         return self.ParseStatementContinue()
7830     }
7831
7832     if self.checkCurrent(TOKEN_TRY) {
7833         return self.ParseStatementTry()
7834     }
7835
7836     if self.checkCurrent(TOKEN_ERROR) {
7837         return self.ParseStatementError()
7838     }
7839
7840     if self.checkCurrent(TOKEN_RETURN) {
7841         return self.ParseStatementReturn()
7842     }
7843
7844     return self.ParseStatementExpressionOrAssignment()
7845 }
7846
7847 func (self *Parser) ParseStatementLet() (AstStatement, error) {
7848     token, err := self.expectCurrent(TOKEN_LET)
7849     if err != nil {
7850         return nil, err
7851     }
7852     location := token.Location
7853
7854     identifier, err := self.ParseIdentifier()
7855     if err != nil {
7856         return nil, err
7857     }
7858
7859     _, err = self.expectCurrent(TOKEN_ASSIGN)
7860     if err != nil {
7861         return nil, err
7862     }
7863
7864     expression, err := self.ParseExpression()
7865     if err != nil {
7866         return nil, err
7867     }
7868
7869     _, err = self.expectCurrent(TOKEN_SEMICOLON)
7870     if err != nil {
7871         return nil, err
7872     }
7873
7874     if astExpressionFunction, ok := expression.(*AstExpressionFunction); ok {
7875         astExpressionFunction.Name = identifier.Name
7876     } else if astExpressionType, ok := expression.(*AstExpressionType); ok {
7877         astExpressionType.Name = identifier.Name.data
7878         if astExpressionTypeExpressionMap, ok :=
7879 ↪ astExpressionType.Expression.(*AstExpressionMap); ok {
7879             updateNamedFunctions(self.lexer.ctx, astExpressionTypeExpressionMap,
7880 ↪ identifier.Name.data+TOKEN_SCOPE)

```

```

7880     }
7881     } else if astExpressionMap, ok := expression.(*AstExpressionMap); ok {
7882         updateNamedFunctions(self.lexer.ctx, astExpressionMap,
7883             ↵ identifier.Name.data+TOKEN_SCOPE)
7884     }
7885     return &AstStatementLet{location, identifier, expression}, nil
7886 }
7887
7888 func (self *Parser) ParseStatementIfElifElse() (AstStatement, error) {
7889     if !self.checkCurrent(TOKEN_IF) {
7890         _, err := self.expectCurrent(TOKEN_IF)
7891         if err != nil {
7892             return nil, err
7893         }
7894     }
7895     location := self.currentToken.Location
7896
7897     parseConditional := func() (*AstConditional, error) {
7898         token, err := self.advanceToken()
7899         if err != nil {
7900             return &AstConditional{}, err
7901         }
7902         location := token.Location
7903
7904         condition, err := self.ParseExpression()
7905         if err != nil {
7906             return &AstConditional{}, err
7907         }
7908
7909         body, err := self.ParseBlock()
7910         if err != nil {
7911             return &AstConditional{}, err
7912         }
7913
7914         return &AstConditional{location, condition, body}, nil
7915     }
7916
7917     conditionals := []*AstConditional{}
7918     for {
7919         if len(conditionals) == 0 && self.checkCurrent(TOKEN_IF) {
7920             conditional, err := parseConditional()
7921             if err != nil {
7922                 return nil, err
7923             }
7924             conditionals = append(conditionals, conditional)
7925             continue
7926         }
7927
7928         if len(conditionals) != 0 && self.checkCurrent(TOKEN_ELIF) {
7929             conditional, err := parseConditional()
7930             if err != nil {
7931                 return nil, err
7932             }
7933             conditionals = append(conditionals, conditional)
7934             continue
7935         }
7936
7937         break
7938     }
7939
7940     var elseBlock *AstBlock = nil
7941     if self.checkCurrent(TOKEN_ELSE) {
7942         _, err := self.expectCurrent(TOKEN_ELSE)
7943         if err != nil {
7944             return nil, err
7945         }

```

```

7946
7947     block, err := self.ParseBlock()
7948     if err != nil {
7949         return nil, err
7950     }
7951     elseBlock = block
7952 }
7953
7954 return &AstStatementIfElifElse{location, conditionals, elseBlock}, nil
7955 }
7956
7957 func (self *Parser) ParseStatementFor() (AstStatement, error) {
7958     token, err := self.expectCurrent(TOKEN_FOR)
7959     if err != nil {
7960         return nil, err
7961     }
7962     location := token.Location
7963
7964     identifierK, err := self.ParseIdentifier()
7965     if err != nil {
7966         return nil, err
7967     }
7968     kIsReference := false
7969     vIsReference := false
7970
7971     if self.checkCurrent(TOKEN_MKREF) {
7972         _, err := self.expectCurrent(TOKEN_MKREF)
7973         if err != nil {
7974             return nil, err
7975         }
7976         kIsReference = true
7977     }
7978
7979     var identifierV *AstIdentifier = nil
7980     if self.checkCurrent(TOKEN_COMMA) {
7981         _, err := self.expectCurrent(TOKEN_COMMA)
7982         if err != nil {
7983             return nil, err
7984         }
7985
7986         identifier, err := self.ParseIdentifier()
7987         if err != nil {
7988             return nil, err
7989         }
7990         identifierV = identifier
7991
7992         if self.checkCurrent(TOKEN_MKREF) {
7993             _, err := self.expectCurrent(TOKEN_MKREF)
7994             if err != nil {
7995                 return nil, err
7996             }
7997             vIsReference = true
7998         }
7999     }
8000
8001     _, err = self.expectCurrent(TOKEN_IN)
8002     if err != nil {
8003         return nil, err
8004     }
8005
8006     collection, err := self.ParseExpression()
8007     if err != nil {
8008         return nil, err
8009     }
8010
8011     block, err := self.ParseBlock()
8012     if err != nil {

```

```

8013         return nil, err
8014     }
8015
8016     if identifierV != nil && identifierK.Name.data == identifierV.Name.data {
8017         return nil, ParseError{
8018             Location: location,
8019             why:      fmt.Sprintf("duplicate iterator name %s",
↵ quote(identifierK.Name.data)),
8020         }
8021     }
8022
8023     return &AstStatementFor{location, identifierK, identifierV, kIsReference,
↵ vIsReference, collection, block}, nil
8024 }
8025
8026 func (self *Parser) ParseStatementWhile() (AstStatement, error) {
8027     token, err := self.expectCurrent(TOKEN_WHILE)
8028     if err != nil {
8029         return nil, err
8030     }
8031     location := token.Location
8032
8033     expression, err := self.ParseExpression()
8034     if err != nil {
8035         return nil, err
8036     }
8037
8038     block, err := self.ParseBlock()
8039     if err != nil {
8040         return nil, err
8041     }
8042
8043     return &AstStatementWhile{location, expression, block}, nil
8044 }
8045
8046 func (self *Parser) ParseStatementBreak() (AstStatement, error) {
8047     token, err := self.expectCurrent(TOKEN_BREAK)
8048     if err != nil {
8049         return nil, err
8050     }
8051     location := token.Location
8052
8053     _, err = self.expectCurrent(TOKEN_SEMICOLON)
8054     if err != nil {
8055         return nil, err
8056     }
8057
8058     return &AstStatementBreak{location}, nil
8059 }
8060
8061 func (self *Parser) ParseStatementContinue() (AstStatement, error) {
8062     token, err := self.expectCurrent(TOKEN_CONTINUE)
8063     if err != nil {
8064         return nil, err
8065     }
8066     location := token.Location
8067
8068     _, err = self.expectCurrent(TOKEN_SEMICOLON)
8069     if err != nil {
8070         return nil, err
8071     }
8072
8073     return &AstStatementContinue{location}, nil
8074 }
8075
8076 func (self *Parser) ParseStatementTry() (AstStatement, error) {
8077     token, err := self.expectCurrent(TOKEN_TRY)

```

```

8078     if err != nil {
8079         return nil, err
8080     }
8081     location := token.Location
8082
8083     tryBlock, err := self.ParseBlock()
8084     if err != nil {
8085         return nil, err
8086     }
8087
8088     if _, err := self.expectCurrent(TOKEN_CATCH); err != nil {
8089         return nil, err
8090     }
8091
8092     var catchIdentifier *AstIdentifier = nil
8093     if self.checkCurrent(TOKEN_IDENTIFIER) {
8094         identifier, err := self.ParseIdentifier()
8095         if err != nil {
8096             return nil, err
8097         }
8098         catchIdentifier = identifier
8099     }
8100
8101     catchBlock, err := self.ParseBlock()
8102     if err != nil {
8103         return nil, err
8104     }
8105
8106     return &AstStatementTry{location, tryBlock, catchIdentifier, catchBlock}, nil
8107 }
8108
8109 func (self *Parser) ParseStatementError() (AstStatement, error) {
8110     token, err := self.expectCurrent(TOKEN_ERROR)
8111     if err != nil {
8112         return nil, err
8113     }
8114     location := token.Location
8115
8116     expression, err := self.ParseExpression()
8117     if err != nil {
8118         return nil, err
8119     }
8120
8121     _, err = self.expectCurrent(TOKEN_SEMICOLON)
8122     if err != nil {
8123         return nil, err
8124     }
8125
8126     return &AstStatementError{location, expression}, nil
8127 }
8128
8129 func (self *Parser) ParseStatementReturn() (AstStatement, error) {
8130     token, err := self.expectCurrent(TOKEN_RETURN)
8131     if err != nil {
8132         return nil, err
8133     }
8134     location := token.Location
8135
8136     var expression AstExpression = nil
8137     if !self.checkCurrent(TOKEN_SEMICOLON) {
8138         expr, err := self.ParseExpression()
8139         if err != nil {
8140             return nil, err
8141         }
8142         expression = expr
8143     }
8144

```

```

8145     _, err = self.expectCurrent(TOKEN_SEMICOLON)
8146     if err != nil {
8147         return nil, err
8148     }
8149
8150     return &AstStatementReturn{location, expression}, nil
8151 }
8152
8153 func (self *Parser) ParseStatementExpressionOrAssignment() (AstStatement, error) {
8154     expression, err := self.ParseExpression()
8155     if err != nil {
8156         return nil, err
8157     }
8158
8159     if !self.checkCurrent(TOKEN_ASSIGN) {
8160         _, err = self.expectCurrent(TOKEN_SEMICOLON)
8161         if err != nil {
8162             return nil, err
8163         }
8164
8165         return &AstStatementExpression{expression.ExpressionLocation(), expression}, nil
8166     }
8167
8168     token, err := self.expectCurrent(TOKEN_ASSIGN)
8169     if err != nil {
8170         return nil, err
8171     }
8172     location := token.Location
8173
8174     rhs, err := self.ParseExpression()
8175     if err != nil {
8176         return nil, err
8177     }
8178
8179     _, err = self.expectCurrent(TOKEN_SEMICOLON)
8180     if err != nil {
8181         return nil, err
8182     }
8183
8184     return &AstStatementAssignment{location, expression, rhs}, nil
8185 }
8186
8187 // Invoke a callable function, either a *Function or *Builtin value.
8188 func Call(ctx *Context, location *SourceLocation, callable Value, arguments []Value)
8189     (Value, error) {
8190     copied := make([]Value, len(arguments))
8191     for i, argument := range arguments {
8192         copied[i] = argument.Copy()
8193     }
8194     return call(ctx, location, callable, copied)
8195 }
8196
8197 // Invoke a callable function, either a *Function or *Builtin value, with the
8198 // requirement that every argument passed in the arguments slice MUST be a
8199 // freshly copied or moved value. All arguments will automatically have their
8200 // uses unmarked after control returns from the function invocation.
8201 func call(ctx *Context, location *SourceLocation, callable Value, arguments []Value)
8202     (Value, error) {
8203     ctx.callDepth += 1
8204     defer func() { ctx.callDepth -= 1 }()
8205     // A relatively low function call depth of 512 is used (1) to make error
8206     // traces resulting from recursive stack overflow more digestible, and (2)
8207     // to gently encourage the use of iteration over recursion since Mellifera
8208     // does *not* perform any form of tail call optimization.
8209     MAX_CALL_DEPTH := 512
8210     if ctx.callDepth > MAX_CALL_DEPTH {
8211         return nil, NewError(

```

```

8210         location,
8211         ctx.NewStringf("exceeded maximum function call depth of %v", MAX_CALL_DEPTH),
8212     )
8213 }
8214
8215 if len(arguments) > 0 && callableSelfIsPassedByReference(callable) {
8216     if _, ok := arguments[0].(*Reference); !ok {
8217         return nil, NewError(
8218             location,
8219             ctx.NewStringf("invalid function self argument (expected reference,
↪ received %v)", arguments[0].Type()),
8220         )
8221     }
8222 }
8223
8224 if function, ok := callable.(*Function); ok {
8225     if len(arguments) != len(function.ast.Parameters) {
8226         return nil, NewError(
8227             location,
8228             ctx.NewStringf("invalid function argument count (expected %v, received
↪ %v)", len(function.ast.Parameters), len(arguments)),
8229         )
8230     }
8231
8232     env := NewEnvironment(function.env)
8233     for i, parameter := range function.ast.Parameters {
8234         env.letWithLocation(parameter.Name.data, arguments[i], parameter.Location)
8235     }
8236
8237     result, err := function.ast.Body.Eval(ctx, env)
8238     envHeld := env.held
8239     env.tryReuse()
8240     if err != nil {
8241         if e, ok := err.(Error); ok {
8242             e.Trace = append(e.Trace, TraceElement{location, function.String()})
8243             return nil, e
8244         }
8245         return nil, err
8246     }
8247
8248     // Arguments can be safely considered dead as long as the callee
8249     // environment was not closed-over by a newly-created function.
8250     if !envHeld {
8251         for _, argument := range arguments {
8252             unmarkUsed(argument)
8253         }
8254     }
8255
8256     if result.Kind == ControlFlowReturn {
8257         return result.Value, nil
8258     }
8259     if result.Kind == ControlFlowBreak {
8260         return nil, NewError(
8261             result.Location,
8262             ctx.NewString("attempted to break outside of a loop"),
8263         )
8264     }
8265     if result.Kind == ControlFlowContinue {
8266         return nil, NewError(
8267             result.Location,
8268             ctx.NewString("attempted to continue outside of a loop"),
8269         )
8270     }
8271
8272     return ctx.NewNull(), nil
8273 }
8274

```

```

8275     if builtin, ok := callable.(*Builtin); ok {
8276         if err := typeCheckArguments(builtin.types, arguments); err != nil {
8277             e := NewError(
8278                 nil,
8279                 ctx.NewString(err.Error()),
8280             )
8281             e.Trace = append(e.Trace, TraceElement{location, builtin.String()})
8282             return nil, e
8283         }
8284         result, err := builtin.impl(ctx, arguments)
8285         if err != nil {
8286             if e, ok := err.(Error); ok {
8287                 e.Trace = append(e.Trace, TraceElement{location, builtin.String()})
8288                 return nil, e
8289             }
8290             return nil, err
8291         }
8292         // Built-in functions do not produce values that close-over their
8293         // environment, so it is always safe to unmark built-in arguments.
8294         for _, argument := range arguments {
8295             unmarkUsed(argument)
8296         }
8297         return result, nil
8298     }
8299
8300     return nil, NewError(
8301         location,
8302         ctx.NewStringf("attempted to call non-function type %s with value %s",
8303             ↪ quote(callable.TypeName()), callable.String()),
8304     )
8305 }
8306
8307 // Invoke a callable function, either a *Function or *Builtin value, and strip
8308 // internal trace elements on error. Used to invoke built-in functions created
8309 // with Context.NewValueFromSource() and Context.NewValueFromSourceOrPanic() in
8310 // a way that hides the internal source of those functions.
8311 func CallBuiltinFromSource(ctx *Context, function Value, arguments []Value) (Value,
8312     ↪ error) {
8313     result, err := Call(ctx, nil, function, arguments)
8314     if err != nil {
8315         if e, ok := err.(Error); ok {
8316             e.Trace = nil
8317             return nil, e
8318         }
8319     }
8320     return result, err
8321 }
8322
8323 const (
8324     ANY          = "any"
8325     NULL         = "null"
8326     BOOLEAN     = "boolean"
8327     NUMBER      = "number"
8328     STRING      = "string"
8329     REGEXP      = "regexp"
8330     VECTOR      = "vector"
8331     MAP         = "map"
8332     SET         = "set"
8333     REFERENCE   = "reference"
8334     CALLABLE    = "callable"
8335     EXTERNAL    = "external"
8336 )
8337
8338 type Type struct {
8339     Kind string
8340     Base *Type
8341 }

```

```

8340
8341 func (self Type) String() string {
8342     if self.Kind == REFERENCE && self.Base != nil {
8343         return fmt.Sprintf("reference to %s", self.Base.String())
8344     }
8345     return self.Kind
8346 }
8347
8348 func TVal(kind string) Type {
8349     return Type{kind, nil}
8350 }
8351
8352 func TRef(base Type) Type {
8353     return Type{REFERENCE, &base}
8354 }
8355
8356 func typeCheckArgument(index int, expected Type, received Value) error {
8357     switch expected.Kind {
8358     case ANY:
8359         return nil
8360     case NULL:
8361         if _, ok := received.(*Null); ok {
8362             return nil
8363         }
8364     case BOOLEAN:
8365         if _, ok := received.(*Boolean); ok {
8366             return nil
8367         }
8368     case NUMBER:
8369         if _, ok := received.(*Number); ok {
8370             return nil
8371         }
8372     case STRING:
8373         if _, ok := received.(*String); ok {
8374             return nil
8375         }
8376     case REGEXP:
8377         if _, ok := received.(*Regexp); ok {
8378             return nil
8379         }
8380     case VECTOR:
8381         if _, ok := received.(*Vector); ok {
8382             return nil
8383         }
8384     case MAP:
8385         if _, ok := received.(*Map); ok {
8386             return nil
8387         }
8388     case SET:
8389         if _, ok := received.(*Set); ok {
8390             return nil
8391         }
8392     case REFERENCE:
8393         reference, ok := received.(*Reference)
8394         if !ok {
8395             break
8396         }
8397         if expected.Base == nil {
8398             return nil // No required base type.
8399         }
8400         if err := typeCheckArgument(index, *expected.Base, reference.data); err != nil {
8401             return fmt.Errorf("expected reference to %s value for argument %v, received
↪ reference to %s", expected.Base, index+1, reference.data.TypeName())
8402         }
8403         return nil
8404     case CALLABLE:
8405         if _, ok := received.(*Function); ok {

```

```

8406         return nil
8407     }
8408     if _, ok := received.(*Builtin); ok {
8409         return nil
8410     }
8411     case EXTERNAL:
8412         if _, ok := received.(*External); ok {
8413             return nil
8414         }
8415     }
8416     return fmt.Errorf("expected %s value for argument %v, received %s", expected,
↪ index+1, received.TypeName())
8417 }
8418
8419 func typeCheckArguments(types []Type, arguments []Value) error {
8420     if len(types) != len(arguments) {
8421         return fmt.Errorf("invalid function argument count (expected %v, received %v)",
↪ len(types), len(arguments))
8422     }
8423
8424     for i := range types {
8425         if err := typeCheckArgument(i, types[i], arguments[i]); err != nil {
8426             return err
8427         }
8428     }
8429
8430     return nil
8431 }
8432
8433 const _ITERATOR_SOURCE = `
8434 let iterator = type {
8435     .eoi = function() {
8436         error null; # end-of-iteration
8437     },
8438     .next = function.&(self) {
8439         error "unimplemented iterator::next";
8440     },
8441     .count = function(self) {
8442         let count = 0;
8443         for _ in self {
8444             count = count + 1;
8445         }
8446         return count;
8447     },
8448     .contains = function(self, value) {
8449         for x in self {
8450             if x == value {
8451                 return true;
8452             }
8453         }
8454         return false;
8455     },
8456     .any = function(self, func) {
8457         for x in self {
8458             let result = func(x);
8459             if not ty::is_boolean(result) {
8460                 error $"expected function {func} to return a boolean (received
↪ {typename(result)} {repr(result)})";
8461             }
8462             if result {
8463                 return true;
8464             }
8465         }
8466         return false;
8467     },
8468     .all = function(self, func) {
8469         for x in self {

```

```

8470         let result = func(x);
8471         if not ty::is_boolean(result) {
8472             error $"expected function {func} to return a boolean (received
↪ {typename(result)} {repr(result)})";
8473         }
8474         if not result {
8475             return false;
8476         }
8477     }
8478     return true;
8479 },
8480 .map = function(self, func) {
8481     let map_iterator = type extends iterator {
8482         .next = function.&(self) {
8483             return func(self.base.next());
8484         },
8485     };
8486     return new map_iterator {
8487         .base = self,
8488     };
8489 },
8490 .filter = function(self, func) {
8491     let filter_iterator = type extends iterator {
8492         .next = function.&(self) {
8493             while true {
8494                 let current = self.base.next();
8495                 let result = func(current);
8496                 if not ty::is_boolean(result) {
8497                     error $"expected function {func} to return a boolean (received
↪ {typename(result)} {repr(result)})";
8498                 }
8499                 if result {
8500                     return current;
8501                 }
8502             }
8503         },
8504     };
8505     return new filter_iterator {
8506         .base = self,
8507     };
8508 },
8509 .reduce = function(self, func) {
8510     let accumulator = null;
8511     let has_element = false;
8512     for x in self {
8513         if not has_element {
8514             accumulator = x;
8515             has_element = true;
8516             continue;
8517         }
8518         accumulator = func(accumulator, x);
8519     }
8520     if not has_element {
8521         error "attempted iterator::reduce on an exhausted iterator";
8522     }
8523     return accumulator;
8524 },
8525 .into_vector = function(self) {
8526     return vector::init(self);
8527 },
8528 .into_map = function(self) {
8529     return map::init(self);
8530 },
8531 .into_set = function(self) {
8532     return set::init(self);
8533 },
8534 };

```

```

8535 return iterator;
8536
8537
8538 func BuiltinBooleanInit(ctx *Context) Value {
8539     return ctx.NewBuiltin("boolean::init", []Type{TVal(ANY)}, func(ctx *Context,
8540 ↪ arguments []Value) (Value, error) {
8541         if x, ok := arguments[0].(*Boolean); ok {
8542             return ctx.NewBoolean(x.data), nil
8543         }
8544         if x, ok := arguments[0].(*Number); ok {
8545             return ctx.NewBoolean(!(math.IsNaN(x.data) || x.data == 0.0)), nil
8546         }
8547         if x, ok := arguments[0].(*String); ok && x.data == "true" {
8548             return ctx.NewBoolean(true), nil
8549         }
8550         if x, ok := arguments[0].(*String); ok && x.data == "false" {
8551             return ctx.NewBoolean(false), nil
8552         }
8553         return nil, NewError(nil, ctx.NewStringf("cannot convert value %v to boolean",
8554 ↪ arguments[0]))
8555     })
8556 }
8557
8558 func BuiltinNumberInit(ctx *Context) Value {
8559     return ctx.NewBuiltin("number::init", []Type{TVal(ANY)}, func(ctx *Context, arguments
8560 ↪ []Value) (Value, error) {
8561         if x, ok := arguments[0].(*Number); ok {
8562             return ctx.NewNumber(x.data), nil
8563         }
8564         if x, ok := arguments[0].(*Boolean); ok && x.data {
8565             return ctx.NewNumber(1.0), nil
8566         }
8567         if x, ok := arguments[0].(*Boolean); ok && !x.data {
8568             return ctx.NewNumber(0.0), nil
8569         }
8570         if x, ok := arguments[0].(*String); ok {
8571             data := x.data
8572             sign := +1 // implicitly positive
8573             if strings.HasPrefix(x.data, "+") {
8574                 sign = +1 // explicitly positive
8575                 data = data[1:]
8576             } else if strings.HasPrefix(x.data, "-") {
8577                 sign = -1 // explicitly negative
8578                 data = data[1:]
8579             }
8580
8581             if data == "Inf" {
8582                 return ctx.NewNumber(math.Inf(sign)), nil
8583             }
8584             if data == "NaN" {
8585                 return ctx.NewNumber(math.NaN()), nil
8586             }
8587
8588             if ctx.reNumberHexFullmatch.MatchString(data) {
8589                 parsed, err := strconv.ParseInt(data, 0, 64)
8590                 if err != nil {
8591                     if errors.Is(err, strconv.ErrRange) {
8592                         return nil, NewError(nil, ctx.NewStringf("hexadecimal integer %s
8593 ↪ is outside the safe integer range", data))
8594                     }
8595                     return nil, NewError(nil, ctx.NewString(err.Error()))
8596                 }
8597                 if parsed < MIN_SAFE_INTEGER || parsed > MAX_SAFE_INTEGER {
8598                     return nil, NewError(nil, ctx.NewStringf("hexadecimal integer %s is
8599 ↪ outside the safe integer range", data))
8600                 }
8601                 return ctx.NewNumber(float64(sign) * float64(parsed)), nil

```

```

8597     }
8598     if ctx.reNumberDecFullmatch.MatchString(data) {
8599         parsed, err := strconv.ParseFloat(data, 64)
8600         if err != nil {
8601             return nil, NewError(nil, ctx.NewString(err.Error()))
8602         }
8603         return ctx.NewNumber(float64(sign) * parsed), nil
8604     }
8605 }
8606 return nil, NewError(nil, ctx.NewStringf("cannot convert value %v to number",
↪ arguments[0]))
8607 })
8608 }
8609
8610 func BuiltinNumberIsNan(ctx *Context) Value {
8611     return ctx.NewBuiltin("number::is_nan", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
8612         self := arguments[0].(*Number)
8613
8614         return ctx.NewBoolean(math.IsNaN(self.data)), nil
8615     })
8616 }
8617
8618 func BuiltinNumberIsInf(ctx *Context) Value {
8619     return ctx.NewBuiltin("number::is_inf", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
8620         self := arguments[0].(*Number)
8621
8622         return ctx.NewBoolean(math.IsInf(self.data, 0)), nil
8623     })
8624 }
8625
8626 func BuiltinNumberIsFinite(ctx *Context) Value {
8627     return ctx.NewBuiltin("number::is_finite", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
8628         self := arguments[0].(*Number)
8629
8630         return ctx.NewBoolean(!math.IsInf(self.data, 0) && !math.IsNaN(self.data)), nil
8631     })
8632 }
8633
8634 func BuiltinNumberIsInteger(ctx *Context) Value {
8635     return ctx.NewBuiltin("number::is_integer", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
8636         self := arguments[0].(*Number)
8637
8638         if math.IsInf(self.data, 0) || math.IsNaN(self.data) {
8639             return ctx.NewBoolean(false), nil
8640         }
8641
8642         return ctx.NewBoolean(math.Trunc(self.data) == self.data), nil
8643     })
8644 }
8645
8646 func BuiltinNumberFormat(ctx *Context) Value {
8647     // $1 sign (+): Always show sign.
8648     // $2 altf (#): Alternate form => 0b for binary, 0x for hex with base x, 0X
8649     //               for hex with base X.
8650     // $3 zero (0): Pad with zeros up to the provided width.
8651     // $4 wstr (width): Minimum total width of the output string.
8652     // $5 prec (precision): Number of displayed digits after the decimal point.
8653     // $6 base (fdbxX): Digit base => b for binary, x for hex (lower), X for
8654     //                   hex (upper), d for decimal integer, f for decimal
8655     //                   float.
8656     formatString := `^(\+)?(#{0})?(\d*)(\.\d+)?([fdbxX])$`
8657     formatRegexp := regexp.MustCompile(formatString)
8658     // Expected format used for error messages, matching the general shape of

```

```

8659 // the actual format string, but without the captures and anchors visible.
8660 expected := `\\+?#?0?\\d*(\\.\\d+)?[fdbxX]`
8661
8662 return ctx.NewBuiltin("number::format", []Type{TVal(NUMBER), TVal(STRING)}, func(ctx
↪ *Context, arguments []Value) (Value, error) {
8663     self := arguments[0].(*Number)
8664     format := arguments[1].(*String)
8665
8666     matches := formatRegexp.FindStringSubmatch(format.data)
8667     if matches == nil {
8668         return nil, NewError(nil, ctx.NewStringf("expected format string matching %s,
↪ received %v", expected, format))
8669     }
8670
8671     sign := matches[1]
8672     altf := matches[2]
8673     zero := matches[3]
8674     wstr := matches[4]
8675     prec := matches[5]
8676     base := matches[6]
8677
8678     switch base {
8679     case "b", "x", "X":
8680         if prec != "" {
8681             var ifmt string
8682             if base == "b" {
8683                 ifmt = "binary"
8684             } else {
8685                 ifmt = "hexadecimal"
8686             }
8687             return nil, NewError(nil, ctx.NewStringf("precision specifier not
↪ supported for %s integer format, received %v", ifmt, format))
8688         }
8689         integer, err := ValueAsSafeInteger(self)
8690         if err != nil {
8691             return nil, NewError(nil, ctx.NewString(err.Error()))
8692         }
8693         if altf != "" && zero != "" && wstr != "" {
8694             // For some reason, Go's printf-family of functions exclude the
8695             // 0b/0x/0X prefix from the width when zero-padding. Reduce the
8696             // width by 2 so that the actual minimum output width matches
8697             // the requested minimum output width.
8698             n, err := strconv.Atoi(wstr)
8699             if err != nil {
8700                 return nil, NewError(nil, ctx.NewString(err.Error()))
8701             }
8702             n -= 2
8703             if n < 0 {
8704                 n = 0
8705             }
8706             wstr = strconv.Itoa(n)
8707         }
8708         fmtStr := "%" + sign + altf + zero + wstr + base
8709         return ctx.NewString(fmt.Sprintf(fmtStr, integer)), nil
8710     case "d":
8711         if altf != "" {
8712             return nil, NewError(nil, ctx.NewStringf("# flag not supported for
↪ decimal integer format, received %v", format))
8713         }
8714         if prec != "" {
8715             return nil, NewError(nil, ctx.NewStringf("precision specifier not
↪ supported for decimal integer format, received %v", format))
8716         }
8717         integer, err := ValueAsSafeInteger(self)
8718         if err != nil {
8719             return nil, NewError(nil, ctx.NewString(err.Error()))
8720         }

```

```

8721         fmtStr := "%" + sign + zero + wstr + "d"
8722         return ctx.NewString(fmt.Sprintf(fmtStr, integer)), nil
8723     case "f":
8724         if altf != "" {
8725             return nil, NewError(nil, ctx.NewStringf("# flag not supported for
↪ decimal float format, received %v", format))
8726         }
8727         var s string
8728         if prec != "" && !math.IsNaN(self.data) && !math.IsInf(self.data, 0) {
8729             ndigits, err := strconv.Atoi(prec[1:])
8730             if err != nil {
8731                 return nil, NewError(nil, ctx.NewString(err.Error()))
8732             }
8733             s = strconv.FormatFloat(self.data, 'f', ndigits, 64)
8734         } else {
8735             s = self.String()
8736         }
8737         if sign == "+" && !math.IsNaN(self.data) && !strings.HasPrefix(s, "-") {
8738             s = "+" + s
8739         }
8740         if wstr != "" {
8741             width, err := strconv.Atoi(wstr)
8742             if err != nil {
8743                 return nil, NewError(nil, ctx.NewString(err.Error()))
8744             }
8745             if len(s) < width {
8746                 if zero == "0" && !math.IsNaN(self.data) && !math.IsInf(self.data, 0)
↪ {
8747                     if s[0] == '+' || s[0] == '-' {
8748                         s = string(s[0]) + strings.Repeat("0", width-len(s)) + s[1:]
8749                     } else {
8750                         s = strings.Repeat("0", width-len(s)) + s
8751                     }
8752                 } else {
8753                     s = strings.Repeat(" ", width-len(s)) + s
8754                 }
8755             }
8756         }
8757         return ctx.NewString(s), nil
8758     default:
8759         panic("unreachable")
8760 }
8761 })
8762 }
8763
8764 func BuiltinNumberFixed(ctx *Context) Value {
8765     return ctx.NewBuiltin("number::fixed", []Type{TVal(NUMBER), TVal(NUMBER)}, func(ctx
↪ *Context, arguments []Value) (Value, error) {
8766         self := arguments[0].(*Number)
8767
8768         precision, err := ValueAsSafeInteger(arguments[1])
8769         if err != nil {
8770             return nil, NewError(nil, ctx.NewString(err.Error()))
8771         }
8772         if precision < 0 {
8773             return nil, NewError(nil, ctx.NewStringf("expected non-negative integer,
↪ received %v", arguments[1]))
8774         }
8775
8776         if math.IsNaN(self.data) || math.IsInf(self.data, 0) {
8777             return self, nil
8778         }
8779
8780         factor := math.Pow(10.0, float64(precision))
8781         fixed := math.Round(self.data*factor) / float64(factor)
8782         return ctx.NewNumber(fixed), nil
8783     })

```

```

8784 }
8785
8786 func BuiltinNumberTrunc(ctx *Context) Value {
8787     return ctx.NewBuiltin("number::trunc", []Type{TVal(NUMBER)}, func(ctx *Context,
8788 ↪ arguments []Value) (Value, error) {
8789         self := arguments[0].(*Number)
8790
8791         return ctx.NewNumber(math.Trunc(self.data)), nil
8792     })
8793 }
8794
8795 func BuiltinNumberRound(ctx *Context) Value {
8796     return ctx.NewBuiltin("number::round", []Type{TVal(NUMBER)}, func(ctx *Context,
8797 ↪ arguments []Value) (Value, error) {
8798         self := arguments[0].(*Number)
8799
8800         return ctx.NewNumber(math.Round(self.data)), nil
8801     })
8802 }
8803
8804 func BuiltinNumberFloor(ctx *Context) Value {
8805     return ctx.NewBuiltin("number::floor", []Type{TVal(NUMBER)}, func(ctx *Context,
8806 ↪ arguments []Value) (Value, error) {
8807         self := arguments[0].(*Number)
8808
8809         return ctx.NewNumber(math.Floor(self.data)), nil
8810     })
8811 }
8812
8813 func BuiltinNumberCeil(ctx *Context) Value {
8814     return ctx.NewBuiltin("number::ceil", []Type{TVal(NUMBER)}, func(ctx *Context,
8815 ↪ arguments []Value) (Value, error) {
8816         self := arguments[0].(*Number)
8817
8818         return ctx.NewNumber(math.Ceil(self.data)), nil
8819     })
8820 }
8821
8822 func BuiltinStringInit(ctx *Context) Value {
8823     return ctx.NewBuiltin("string::init", []Type{TVal(ANY)}, func(ctx *Context, arguments
8824 ↪ []Value) (Value, error) {
8825         value := arguments[0]
8826
8827         if metaFunction, ok := MetaFunction(ctx, value, ctx.constStringIntoString); ok {
8828             result, err := ctx.callMetaWithSelf(nil, metaFunction, value)
8829             if err != nil {
8830                 return nil, err
8831             }
8832             resultString, ok := result.(*String)
8833             if !ok {
8834                 return nil, NewError(
8835                     nil,
8836                     ctx.NewStringf("metafunction %s returned %v",
8837 ↪ quote(ctx.constStringIntoString.data), result),
8838                 )
8839             }
8840             return resultString.Copy(), nil
8841         }
8842
8843         if valueString, ok := value.(*String); ok {
8844             return valueString.Copy(), nil
8845         }
8846
8847         return ctx.NewString(value.String()), nil
8848     })
8849 }
8850

```

```

8845 func BuiltinStringBytes(ctx *Context) Value {
8846     return ctx.NewBuiltin("string::bytes", []Type{TVal(STRING)}, func(ctx *Context,
8847 ↪ arguments []Value) (Value, error) {
8848         self := arguments[0].(*String)
8849
8850         vector := ctx.NewVectorOrPanic(nil)
8851         bytes := []byte(self.data)
8852         for i := range bytes {
8853             err := vector.Push(ctx.NewString(string([]byte{bytes[i]})))
8854             if err != nil {
8855                 return nil, NewError(nil, ctx.NewString(err.Error()))
8856             }
8857         }
8858         return vector, nil
8859     })
8860 }
8861
8862 func BuiltinStringRunes(ctx *Context) Value {
8863     return ctx.NewBuiltin("string::runes", []Type{TVal(STRING)}, func(ctx *Context,
8864 ↪ arguments []Value) (Value, error) {
8865         self := arguments[0].(*String)
8866
8867         vector := ctx.NewVectorOrPanic(nil)
8868         for _, r := range self.data {
8869             err := vector.Push(ctx.NewString(string(r)))
8870             if err != nil {
8871                 return nil, NewError(nil, ctx.NewString(err.Error()))
8872             }
8873         }
8874         return vector, nil
8875     })
8876 }
8877
8878 func BuiltinStringCount(ctx *Context) Value {
8879     return ctx.NewBuiltin("string::count", []Type{TVal(STRING)}, func(ctx *Context,
8880 ↪ arguments []Value) (Value, error) {
8881         self := arguments[0].(*String)
8882
8883         return ctx.NewNumber(float64(len(self.data))), nil
8884     })
8885 }
8886
8887 func BuiltinStringIsEmpty(ctx *Context) Value {
8888     return ctx.NewBuiltin("string::is_empty", []Type{TVal(STRING)}, func(ctx *Context,
8889 ↪ arguments []Value) (Value, error) {
8890         self := arguments[0].(*String)
8891
8892         return ctx.NewBoolean(len(self.data) == 0), nil
8893     })
8894 }
8895
8896 func BuiltinStringContains(ctx *Context) Value {
8897     return ctx.NewBuiltin("string::contains", []Type{TVal(STRING), TVal(STRING)},
8898 ↪ func(ctx *Context, arguments []Value) (Value, error) {
8899         self := arguments[0].(*String)
8900         target := arguments[1].(*String)
8901
8902         return ctx.NewBoolean(strings.Contains(self.data, target.data)), nil
8903     })
8904 }
8905
8906 func BuiltinStringStartsWith(ctx *Context) Value {
8907     return ctx.NewBuiltin("string::starts_with", []Type{TVal(STRING), TVal(STRING)},
8908 ↪ func(ctx *Context, arguments []Value) (Value, error) {
8909         self := arguments[0].(*String)
8910         target := arguments[1].(*String)

```

```

8906         return ctx.NewBoolean(strings.HasPrefix(self.data, target.data)), nil
8907     })
8908 }
8909
8910 func BuiltinStringEndsWith(ctx *Context) Value {
8911     return ctx.NewBuiltin("string::ends_with", []Type{TVal(STRING), TVal(STRING)},
8912         ↪ func(ctx *Context, arguments []Value) (Value, error) {
8913             self := arguments[0].(*String)
8914             target := arguments[1].(*String)
8915
8916             return ctx.NewBoolean(strings.HasSuffix(self.data, target.data)), nil
8917         })
8918 }
8919
8920 func BuiltinStringTrim(ctx *Context) Value {
8921     return ctx.NewBuiltin("string::trim", []Type{TVal(STRING)}, func(ctx *Context,
8922         ↪ arguments []Value) (Value, error) {
8923             self := arguments[0].(*String)
8924
8925             return ctx.NewString(strings.TrimSpace(self.data)), nil
8926         })
8927 }
8928
8929 func BuiltinStringFind(ctx *Context) Value {
8930     return ctx.NewBuiltin("string::find", []Type{TVal(STRING), TVal(STRING)}, func(ctx
8931         ↪ *Context, arguments []Value) (Value, error) {
8932             self := arguments[0].(*String)
8933             target := arguments[1].(*String)
8934
8935             found := strings.Index(self.data, target.data)
8936             if found == -1 {
8937                 return ctx.NewNull(), nil
8938             }
8939
8940             return ctx.NewNumber(float64(found)), nil
8941         })
8942 }
8943
8944 func BuiltinStringRfind(ctx *Context) Value {
8945     return ctx.NewBuiltin("string::rfind", []Type{TVal(STRING), TVal(STRING)}, func(ctx
8946         ↪ *Context, arguments []Value) (Value, error) {
8947             self := arguments[0].(*String)
8948             target := arguments[1].(*String)
8949
8950             found := strings.LastIndex(self.data, target.data)
8951             if found == -1 {
8952                 return ctx.NewNull(), nil
8953             }
8954
8955             return ctx.NewNumber(float64(found)), nil
8956         })
8957 }
8958
8959 func BuiltinStringSlice(ctx *Context) Value {
8960     return ctx.NewBuiltin("string::slice", []Type{TVal(STRING), TVal(NUMBER),
8961         ↪ TVal(NUMBER)}, func(ctx *Context, arguments []Value) (Value, error) {
8962             self := arguments[0].(*String)
8963
8964             bgn := arguments[1].(*Number)
8965             bgn_index, err := ValueAsIndex(bgn)
8966             if err != nil {
8967                 return nil, NewError(nil, ctx.NewStringf("attempted string::slice with
8968                 ↪ invalid begin index %v (%s)", bgn, err.Error()))
8969             }
8970             if bgn_index > len(self.data) {
8971                 return nil, NewError(nil, ctx.NewStringf("attempted string::slice with
8972                 ↪ invalid begin index %v (string has a count of %v)", bgn, len(self.data)))

```

```

8966     }
8967
8968     end := arguments[2].(*Number)
8969     end_index, err := ValueAsIndex(end)
8970     if err != nil {
8971         return nil, NewError(nil, ctx.NewStringf("attempted string::slice with
↪ invalid end index %v (%s)", end, err.Error()))
8972     }
8973     if end_index > len(self.data) {
8974         return nil, NewError(nil, ctx.NewStringf("attempted string::slice with
↪ invalid end index %v (string has a count of %v)", end, len(self.data)))
8975     }
8976
8977     if end_index < bgn_index {
8978         return nil, NewError(nil, ctx.NewString("attempted string::slice with invalid
↪ indices (slice end is less than slice begin)"))
8979     }
8980
8981     return ctx.NewString(self.data[bgn_index:end_index]), nil
8982 })
8983 }
8984
8985 func BuiltinStringSplit(ctx *Context) Value {
8986     return ctx.NewBuiltin("string::split", []Type{TVal(STRING), TVal(STRING)}, func(ctx
↪ *Context, arguments []Value) (Value, error) {
8987         self := arguments[0].(*String)
8988         target := arguments[1].(*String)
8989
8990         vector := ctx.NewVectorOrPanic(nil)
8991         if len(target.data) == 0 {
8992             for _, r := range self.data {
8993                 err := vector.Push(ctx.NewString(string(r)))
8994                 if err != nil {
8995                     return nil, NewError(nil, ctx.NewString(err.Error()))
8996                 }
8997             }
8998             return vector, nil
8999         }
9000
9001         split := strings.Split(self.data, target.data)
9002         for i := range split {
9003             err := vector.Push(ctx.NewString(split[i]))
9004             if err != nil {
9005                 return nil, NewError(nil, ctx.NewString(err.Error()))
9006             }
9007         }
9008         return vector, nil
9009     })
9010 }
9011
9012 func BuiltinStringJoin(ctx *Context) Value {
9013     return ctx.NewBuiltin("string::join", []Type{TVal(STRING), TVal(VECTOR)}, func(ctx
↪ *Context, arguments []Value) (Value, error) {
9014         self := arguments[0].(*String)
9015         vector := arguments[1].(*Vector)
9016
9017         var sb strings.Builder
9018         for index, value := range vector.Elements() {
9019             string, ok := value.(*String)
9020             if !ok {
9021                 return nil, NewError(
9022                     nil,
9023                     ctx.NewStringf("expected string value for vector element at index %v,
↪ received %s", index, value.TypeName()),
9024                 )
9025             }
9026             if index != 0 {

```

```

9027         sb.WriteString(self.data)
9028     }
9029     sb.WriteString(string.data)
9030 }
9031
9032     return ctx.NewString(sb.String()), nil
9033 })
9034 }
9035
9036 func BuiltinStringCut(ctx *Context) Value {
9037     return ctx.NewBuiltin("string::cut", []Type{TVal(STRING), TVal(STRING)}, func(ctx
↵ *Context, arguments []Value) (Value, error) {
9038         self := arguments[0].(*String)
9039         target := arguments[1].(*String)
9040
9041         found := strings.Index(self.data, target.data)
9042         if found == -1 {
9043             return ctx.NewNull(), nil
9044         }
9045
9046         prefix := ctx.NewString(self.data[0:found])
9047         suffix := ctx.NewString(self.data[found+len(target.data):])
9048         result, err := ctx.NewMap([]MapPair{
9049             {ctx.NewString("prefix"), prefix},
9050             {ctx.NewString("suffix"), suffix},
9051         })
9052         if err != nil {
9053             return nil, NewError(nil, ctx.NewString(err.Error()))
9054         }
9055         return result, nil
9056     })
9057 }
9058
9059 func BuiltinStringReplace(ctx *Context) Value {
9060     return ctx.NewBuiltin("string::replace", []Type{TVal(STRING), TVal(STRING),
↵ TVal(STRING)}, func(ctx *Context, arguments []Value) (Value, error) {
9061         self := arguments[0].(*String)
9062         target := arguments[1].(*String)
9063         replacement := arguments[2].(*String)
9064
9065         return ctx.NewString(strings.ReplaceAll(self.data, target.data,
↵ replacement.data)), nil
9066     })
9067 }
9068
9069 func BuiltinStringToTitle(ctx *Context) Value {
9070     return ctx.NewBuiltin("string::to_title", []Type{TVal(STRING)}, func(ctx *Context,
↵ arguments []Value) (Value, error) {
9071         self := arguments[0].(*String)
9072
9073         return ctx.NewString(strings.Title(strings.ToLower(self.data))), nil
9074     })
9075 }
9076
9077 func BuiltinStringToUpper(ctx *Context) Value {
9078     return ctx.NewBuiltin("string::to_upper", []Type{TVal(STRING)}, func(ctx *Context,
↵ arguments []Value) (Value, error) {
9079         self := arguments[0].(*String)
9080
9081         return ctx.NewString(strings.ToUpper(self.data)), nil
9082     })
9083 }
9084
9085 func BuiltinStringToLower(ctx *Context) Value {
9086     return ctx.NewBuiltin("string::to_lower", []Type{TVal(STRING)}, func(ctx *Context,
↵ arguments []Value) (Value, error) {
9087         self := arguments[0].(*String)

```

```

9088         return ctx.NewString(strings.ToLower(self.data)), nil
9089     })
9090 }
9091
9092 func BuiltinRegexpInit(ctx *Context) Value {
9093     return ctx.NewBuiltin("regexp::init", []Type{TVal(STRING)}, func(ctx *Context,
9094 ↪ arguments []Value) (Value, error) {
9095         text := arguments[0].(*String)
9096
9097         value, err := ctx.NewRegexp(text.data)
9098         if err != nil {
9099             return nil, err
9100         }
9101
9102         return value, nil
9103     })
9104 }
9105
9106 func BuiltinRegexpSplit(ctx *Context) Value {
9107     return ctx.NewBuiltin("regexp::split", []Type{TVal(REGEXP), TVal(STRING)}, func(ctx
9108 ↪ *Context, arguments []Value) (Value, error) {
9109         self := arguments[0].(*Regexp)
9110         text := arguments[1].(*String)
9111
9112         result := ctx.NewVectorOrPanic(nil)
9113         for _, s := range self.data.Split(text.data, -1) {
9114             err := result.Push(ctx.NewString(s))
9115             if err != nil {
9116                 return nil, NewError(nil, ctx.NewString(err.Error()))
9117             }
9118         }
9119
9120         return result, nil
9121     })
9122 }
9123
9124 func BuiltinRegexpReplace(ctx *Context) Value {
9125     return ctx.NewBuiltin("regexp::replace", []Type{TVal(REGEXP), TVal(STRING),
9126 ↪ TVal(STRING)}, func(ctx *Context, arguments []Value) (Value, error) {
9127         self := arguments[0].(*Regexp)
9128         text := arguments[1].(*String)
9129         replacement := arguments[2].(*String)
9130
9131         return ctx.NewString(self.data.ReplaceAllLiteralString(text.data,
9132 ↪ replacement.data)), nil
9133     })
9134 }
9135
9136 func BuiltinVectorInit(ctx *Context) Value {
9137     return ctx.NewBuiltin("vector::init", []Type{TVal(ANY)}, func(ctx *Context, arguments
9138 ↪ []Value) (Value, error) {
9139         value := arguments[0]
9140
9141         if metaFunction, ok := MetaFunction(ctx, value, ctx.constStringNext); ok {
9142             if !callableSelfIsPassedByReference(metaFunction) {
9143                 return nil, NewError(nil, ctx.NewString("iterator next must receive self
9144 ↪ by reference (declared with function.&(self))"))
9145             }
9146             reference := ctx.newReference(value)
9147             defer unmarkReferenced(value)
9148             result := ctx.NewVectorOrPanic(nil)
9149             for {
9150                 iterated, err := call(ctx, nil, metaFunction, []Value{reference})
9151                 if err != nil {
9152                     if error, ok := err.(Error); ok {
9153                         if _, ok := error.Value.(*Null); ok {

```

```

9149             break // end-of-iteration
9150         }
9151     }
9152     return nil, err
9153 }
9154 err = result.Push(iterated.Copy())
9155 if err != nil {
9156     return nil, NewError(nil, ctx.NewString(err.Error()))
9157 }
9158 }
9159 return result, nil
9160 }
9161
9162 if valueNumber, ok := value.(*Number); ok {
9163     integer, err := ValueAsSafeInteger(valueNumber)
9164     if err != nil {
9165         return nil, NewError(nil, ctx.NewString(err.Error()))
9166     }
9167
9168     result := ctx.NewVectorOrPanic(nil)
9169     for i := int64(0); i < integer; i += 1 {
9170         err := result.Push(ctx.NewNull())
9171         if err != nil {
9172             return nil, NewError(nil, ctx.NewString(err.Error()))
9173         }
9174     }
9175     return result, nil
9176 }
9177
9178 if valueVector, ok := value.(*Vector); ok {
9179     result := ctx.NewVectorOrPanic(nil)
9180     for _, element := range valueVector.Elements() {
9181         err := result.Push(element.Copy())
9182         if err != nil {
9183             return nil, NewError(nil, ctx.NewString(err.Error()))
9184         }
9185     }
9186     return result, nil
9187 }
9188
9189 if valueMap, ok := value.(*Map); ok {
9190     result := ctx.NewVectorOrPanic(nil)
9191     for _, pair := range valueMap.Pairs() {
9192         err := result.Push(ctx.NewVectorOrPanic([]Value{pair.Key.Copy(),
↪ pair.Value.Copy()}))
9193         if err != nil {
9194             return nil, NewError(nil, ctx.NewString(err.Error()))
9195         }
9196     }
9197     return result, nil
9198 }
9199
9200 if valueSet, ok := value.(*Set); ok {
9201     result := ctx.NewVectorOrPanic(nil)
9202     for _, element := range valueSet.Elements() {
9203         err := result.Push(element.Copy())
9204         if err != nil {
9205             return nil, NewError(nil, ctx.NewString(err.Error()))
9206         }
9207     }
9208     return result, nil
9209 }
9210
9211 return nil, NewError(nil, ctx.NewStringf("cannot convert value %v to vector",
↪ value))
9212 })
9213 }

```

```

9214
9215 func BuiltinVectorCount(ctx *Context) Value {
9216     return ctx.NewBuiltin("vector::count", []Type{TVal(VECTOR)}, func(ctx *Context,
9217 ↪ arguments []Value) (Value, error) {
9218         self := arguments[0].(*Vector)
9219
9220         return ctx.NewNumber(float64(self.Count())), nil
9221     })
9222 }
9223
9224 func BuiltinVectorIsEmpty(ctx *Context) Value {
9225     return ctx.NewBuiltin("vector::is_empty", []Type{TVal(VECTOR)}, func(ctx *Context,
9226 ↪ arguments []Value) (Value, error) {
9227         self := arguments[0].(*Vector)
9228
9229         return ctx.NewBoolean(self.Count() == 0), nil
9230     })
9231 }
9232
9233 func BuiltinVectorContains(ctx *Context) Value {
9234     return ctx.NewBuiltin("vector::contains", []Type{TVal(VECTOR), TVal(ANY)}, func(ctx
9235 ↪ *Context, arguments []Value) (Value, error) {
9236         self := arguments[0].(*Vector)
9237         target := arguments[1]
9238
9239         for _, element := range self.Elements() {
9240             if element.Equal(target) {
9241                 return ctx.NewBoolean(true), nil
9242             }
9243         }
9244
9245         return ctx.NewBoolean(false), nil
9246     })
9247 }
9248
9249 func BuiltinVectorInsert(ctx *Context) Value {
9250     return ctx.NewBuiltinWithSelfByReference("vector::insert", []Type{TRef(TVal(VECTOR)),
9251 ↪ TVal(NUMBER), TVal(ANY)}, func(ctx *Context, arguments []Value) (Value, error) {
9252         self := arguments[0].(*Reference)
9253         delf := self.data.(*Vector)
9254
9255         index, err := ValueAsIndex(arguments[1])
9256         if err != nil {
9257             return nil, NewError(nil, ctx.NewStringf("attempted vector::insert with
9258 ↪ invalid index %v (%s)", arguments[1], err.Error()))
9259         }
9260
9261         if index > delf.Count() {
9262             return nil, NewError(nil, ctx.NewStringf("attempted vector::insert with
9263 ↪ invalid index %v (vector has a count of %v)", index, delf.Count()))
9264         }
9265
9266         err = delf.Insert(index, arguments[2].Copy())
9267         if err != nil {
9268             return nil, NewError(nil, ctx.NewStringf("invalid vector::insert operation
9269 ↪ (%s)", err.Error()))
9270         }
9271
9272         return ctx.NewNull(), nil
9273     })
9274 }
9275
9276 func BuiltinVectorRemove(ctx *Context) Value {
9277     return ctx.NewBuiltinWithSelfByReference("vector::remove", []Type{TRef(TVal(VECTOR)),
9278 ↪ TVal(NUMBER)}, func(ctx *Context, arguments []Value) (Value, error) {
9279         self := arguments[0].(*Reference)
9280         delf := self.data.(*Vector)

```

```

9273         index, err := ValueAsIndex(arguments[1])
9274         if err != nil {
9275             return nil, NewError(nil, ctx.NewStringf("attempted vector::remove with
9276 ↪ invalid index %v (%s)", arguments[1], err.Error()))
9277         }
9278
9279         if index >= delf.Count() {
9280             return nil, NewError(nil, ctx.NewStringf("attempted vector::remove with
9281 ↪ invalid index %v (vector has a count of %v)", index, delf.Count()))
9282         }
9283
9284         value, err := delf.Remove(index)
9285         if err != nil {
9286             return nil, NewError(nil, ctx.NewStringf("invalid vector::remove operation
9287 ↪ (%s)", err.Error()))
9288         }
9289         return value, nil
9290     })
9291 }
9292
9293 func BuiltinVectorPush(ctx *Context) Value {
9294     return ctx.NewBuiltinWithSelfByReference("vector::push", []Type{TRef(TVal(VECTOR)),
9295 ↪ TVal(ANY)}, func(ctx *Context, arguments []Value) (Value, error) {
9296         self := arguments[0].(*Reference)
9297         delf := self.data.(*Vector)
9298
9299         err := delf.Push(arguments[1].Copy())
9300         if err != nil {
9301             return nil, NewError(nil, ctx.NewStringf("invalid vector::push operation
9302 ↪ (%s)", err.Error()))
9303         }
9304
9305         return ctx.NewNull(), nil
9306     })
9307 }
9308
9309 func BuiltinVectorPop(ctx *Context) Value {
9310     return ctx.NewBuiltinWithSelfByReference("vector::pop", []Type{TRef(TVal(VECTOR))},
9311 ↪ func(ctx *Context, arguments []Value) (Value, error) {
9312         self := arguments[0].(*Reference)
9313         delf := self.data.(*Vector)
9314
9315         if delf.Count() == 0 {
9316             return nil, NewError(nil, ctx.NewString("attempted vector::pop on an empty
9317 ↪ vector"))
9318         }
9319
9320         value, err := delf.Pop()
9321         if err != nil {
9322             return nil, NewError(nil, ctx.NewStringf("invalid vector::pop operation
9323 ↪ (%s)", err.Error()))
9324         }
9325
9326         return value, nil
9327     })
9328 }
9329
9330 func BuiltinVectorAny(ctx *Context) Value {
9331     return ctx.NewBuiltin("vector::any", []Type{TVal(VECTOR), TVal(CALLABLE)}, func(ctx
9332 ↪ *Context, arguments []Value) (Value, error) {
9333         self := arguments[0].(*Vector)
9334
9335         for _, element := range self.Elements() {
9336             result, err := call(ctx, nil, arguments[1], []Value{element.Copy()})
9337             if err != nil {

```

```

9331         return nil, err
9332     }
9333
9334     resultBoolean, ok := result.(*Boolean)
9335     if !ok {
9336         ↪ return nil, NewError(nil, ctx.NewStringf("expected function %s to return
a boolean (received %s %s)", arguments[1], result.TypeName(), result))
9337     }
9338
9339     if resultBoolean.data {
9340         return ctx.NewBoolean(true), nil
9341     }
9342 }
9343
9344 return ctx.NewBoolean(false), nil
9345 })
9346 }
9347
9348 func BuiltinVectorAll(ctx *Context) Value {
9349     ↪ return ctx.NewBuiltin("vector::all", []Type{TVal(VECTOR), TVal(CALLABLE)}, func(ctx
*Context, arguments []Value) (Value, error) {
9350         self := arguments[0].(*Vector)
9351
9352         for _, element := range self.Elements() {
9353             result, err := call(ctx, nil, arguments[1], []Value{element.Copy()})
9354             if err != nil {
9355                 return nil, err
9356             }
9357
9358             resultBoolean, ok := result.(*Boolean)
9359             if !ok {
9360                 ↪ return nil, NewError(nil, ctx.NewStringf("expected function %s to return
a boolean (received %s %s)", arguments[1], result.TypeName(), result))
9361             }
9362
9363             if !resultBoolean.data {
9364                 return ctx.NewBoolean(false), nil
9365             }
9366         }
9367
9368         return ctx.NewBoolean(true), nil
9369     })
9370 }
9371
9372 func BuiltinVectorMap(ctx *Context) Value {
9373     ↪ return ctx.NewBuiltin("vector::map", []Type{TVal(VECTOR), TVal(CALLABLE)}, func(ctx
*Context, arguments []Value) (Value, error) {
9374         self := arguments[0].(*Vector)
9375
9376         mapped := []Value{}
9377         for _, element := range self.Elements() {
9378             result, err := call(ctx, nil, arguments[1], []Value{element.Copy()})
9379             if err != nil {
9380                 return nil, err
9381             }
9382
9383             mapped = append(mapped, result.Copy())
9384         }
9385
9386         return ctx.NewVectorOrPanic(mapped), nil
9387     })
9388 }
9389
9390 func BuiltinVectorFilter(ctx *Context) Value {
9391     ↪ return ctx.NewBuiltin("vector::filter", []Type{TVal(VECTOR), TVal(CALLABLE)},
func(ctx *Context, arguments []Value) (Value, error) {
9392         self := arguments[0].(*Vector)

```

```

9393
9394     filtered := []Value{}
9395     for _, element := range self.Elements() {
9396         result, err := call(ctx, nil, arguments[1], []Value{element.Copy()})
9397         if err != nil {
9398             return nil, err
9399         }
9400
9401         resultBoolean, ok := result.(*Boolean)
9402         if !ok {
9403             ↪ a boolean (received %s %s)", arguments[1], result.TypeName(), result))
9404         }
9405
9406         if resultBoolean.data {
9407             filtered = append(filtered, element.Copy())
9408         }
9409     }
9410
9411     return ctx.NewVectorOrPanic(filtered), nil
9412 })
9413 }
9414
9415 func BuiltinVectorReduce(ctx *Context) Value {
9416     return ctx.NewBuiltin("vector::reduce", []Type{TVal(VECTOR), TVal(CALLABLE)},
9417     ↪ func(ctx *Context, arguments []Value) (Value, error) {
9418         self := arguments[0].(*Vector)
9419
9420         if self.Count() == 0 {
9421             ↪ empty vector"))
9422         }
9423
9424         if self.Count() == 1 {
9425             return self.Get(0).Copy(), nil
9426         }
9427
9428         accumulator := self.Get(0).Copy()
9429         for _, element := range self.Elements()[1:] {
9430             ↪ element.Copy())
9431             if err != nil {
9432                 return nil, err
9433             }
9434             accumulator = result
9435         }
9436
9437         return moveOrCopy(accumulator), nil
9438     })
9439 }
9440
9441 func BuiltinVectorFind(ctx *Context) Value {
9442     return ctx.NewBuiltin("vector::find", []Type{TVal(VECTOR), TVal(ANY)}, func(ctx
9443     ↪ *Context, arguments []Value) (Value, error) {
9444         self := arguments[0].(*Vector)
9445         target := arguments[1]
9446
9447         elements := self.Elements()
9448         for index := range elements {
9449             if elements[index].Equal(target) {
9450                 return ctx.NewNumber(float64(index)), nil
9451             }
9452         }
9453
9454         return ctx.NewNull(), nil
9455     })
9456 }

```

```

9455
9456 func BuiltinVectorRfind(ctx *Context) Value {
9457     return ctx.NewBuiltin("vector::rfind", []Type{TVal(VECTOR), TVal(ANY)}, func(ctx
↪ *Context, arguments []Value) (Value, error) {
9458         self := arguments[0].(*Vector)
9459         target := arguments[1]
9460
9461         elements := self.Elements()
9462         for i := range elements {
9463             index := len(elements) - i - 1
9464             if elements[index].Equal(target) {
9465                 return ctx.NewNumber(float64(index)), nil
9466             }
9467         }
9468
9469         return ctx.NewNull(), nil
9470     })
9471 }
9472
9473 func BuiltinVectorSlice(ctx *Context) Value {
9474     return ctx.NewBuiltin("vector::slice", []Type{TVal(VECTOR), TVal(NUMBER),
↪ TVal(NUMBER)}, func(ctx *Context, arguments []Value) (Value, error) {
9475         self := arguments[0].(*Vector)
9476
9477         bgn := arguments[1].(*Number)
9478         bgn_index, err := ValueAsIndex(bgn)
9479         if err != nil {
9480             return nil, NewError(nil, ctx.NewStringf("attempted vector::slice with
↪ invalid begin index %v (%s)", bgn, err.Error()))
9481         }
9482         if bgn_index > self.Count() {
9483             return nil, NewError(nil, ctx.NewStringf("attempted vector::slice with
↪ invalid begin index %v (vector has a count of %v)", bgn, self.Count()))
9484         }
9485
9486         end := arguments[2].(*Number)
9487         end_index, err := ValueAsIndex(end)
9488         if err != nil {
9489             return nil, NewError(nil, ctx.NewStringf("attempted vector::slice with
↪ invalid end index %v (%s)", end, err.Error()))
9490         }
9491         if end_index > self.Count() {
9492             return nil, NewError(nil, ctx.NewStringf("attempted vector::slice with
↪ invalid end index %v (vector has a count of %v)", end, self.Count()))
9493         }
9494
9495         if end_index < bgn_index {
9496             return nil, NewError(nil, ctx.NewString("attempted vector::slice with invalid
↪ indices (slice end is less than slice begin)"))
9497         }
9498
9499         result := ctx.NewVectorOrPanic(nil)
9500         for i := bgn_index; i < end_index; i += 1 {
9501             err := result.Push(self.Get(i).Copy())
9502             if err != nil {
9503                 return nil, NewError(nil, ctx.NewString(err.Error()))
9504             }
9505         }
9506
9507         return result, nil
9508     })
9509 }
9510
9511 func BuiltinVectorReversed(ctx *Context) Value {
9512     return ctx.NewBuiltin("vector::reversed", []Type{TVal(VECTOR)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
9513         self := arguments[0].(*Vector)

```

```

9514         elements := []Value{}
9515         for _, element := range self.Elements() {
9516             elements = append(elements, element.Copy())
9517         }
9518         slices.Reverse(elements)
9519         return ctx.NewVectorOrPanic(elements), nil
9520     })
9521 }
9522 }
9523
9524 func BuiltinVectorSorted(ctx *Context) Value {
9525     function := ctx.NewValueFromSourceOrPanic("vector::sorted", `
9526 let sort = function(x) {
9527     if x.count() <= 1 {
9528         return x;
9529     }
9530     let mid = (x.count() / 2).trunc();
9531     let lo = sort(x.slice(0, mid));
9532     let hi = sort(x.slice(mid, x.count()));
9533     let lo_index = 0;
9534     let hi_index = 0;
9535     let result = [];
9536     for _ in x.count() {
9537         if lo_index == lo.count() {
9538             result.push(hi[hi_index]);
9539             hi_index = hi_index + 1;
9540             continue;
9541         }
9542         if hi_index == hi.count() {
9543             result.push(lo[lo_index]);
9544             lo_index = lo_index + 1;
9545             continue;
9546         }
9547         if lo[lo_index] < hi[hi_index] {
9548             result.push(lo[lo_index]);
9549             lo_index = lo_index + 1;
9550             continue;
9551         }
9552         if lo[lo_index] > hi[hi_index] {
9553             result.push(hi[hi_index]);
9554             hi_index = hi_index + 1;
9555             continue;
9556         }
9557         # When equal, take from the lower vector by convention.
9558         result.push(lo[lo_index]);
9559         lo_index = lo_index + 1;
9560     }
9561     return result;
9562 };
9563 return function(self) {
9564     if not ty::is_vector(self) {
9565         error $"expected vector value for argument 1, received {typename(self)}";
9566     }
9567     try { return sort(self); } catch err { error err; }
9568 };
9569 `)
9570
9571     return ctx.NewBuiltin("vector::sorted", []Type{TVal(VECTOR)}, func(ctx *Context,
9572 ↪ arguments []Value) (Value, error) {
9573         return CallBuiltinFromSource(ctx, function, arguments)
9574     })
9575 }
9576
9577 func BuiltinVectorSortedBy(ctx *Context) Value {
9578     function := ctx.NewValueFromSourceOrPanic("vector::sorted_by", `
9579 let sort = function(x, compare) {
9580     if x.count() <= 1 {

```

```

9580         return x;
9581     }
9582     let mid = (x.count() / 2).trunc();
9583     let lo = sort(x.slice(0, mid), compare);
9584     let hi = sort(x.slice(mid, x.count()), compare);
9585     let lo_index = 0;
9586     let hi_index = 0;
9587     let result = [];
9588     for _ in x.count() {
9589         if lo_index == lo.count() {
9590             result.push(hi[hi_index]);
9591             hi_index = hi_index + 1;
9592             continue;
9593         }
9594         if hi_index == hi.count() {
9595             result.push(lo[lo_index]);
9596             lo_index = lo_index + 1;
9597             continue;
9598         }
9599
9600         let cmp = compare(lo[lo_index], hi[hi_index]);
9601         if cmp < 0 {
9602             result.push(lo[lo_index]);
9603             lo_index = lo_index + 1;
9604             continue;
9605         }
9606         if cmp > 0 {
9607             result.push(hi[hi_index]);
9608             hi_index = hi_index + 1;
9609             continue;
9610         }
9611         # When equal, take from the lower vector by convention.
9612         result.push(lo[lo_index]);
9613         lo_index = lo_index + 1;
9614     }
9615     return result;
9616 };
9617 return function(self, compare) {
9618     if not ty::is_vector(self) {
9619         error $"expected vector value for argument 1, received {typename(self)}";
9620     }
9621     try { return sort(self, compare); } catch err { error err; }
9622 };
9623 `)
9624
9625     return ctx.NewBuiltin("vector::sorted_by", []Type{TVal(VECTOR), TVal(CALLABLE)},
9626 ↪ func(ctx *Context, arguments []Value) (Value, error) {
9627         return CallBuiltinFromSource(ctx, function, arguments)
9628     })
9629 }
9630
9631 func BuiltinVectorIntoIterator(ctx *Context) Value {
9632     function := ctx.NewValueFromSourceOrPanic("vector::into_iterator", `
9633 return function(self) {
9634     let vector_iterator = type extends iterator {
9635         .next = function.&(self) {
9636             if self.index >= self.vector.count() {
9637                 return iterator::eoi();
9638             }
9639             let current = self.vector[self.index];
9640             self.index = self.index + 1;
9641             return current;
9642         },
9643     };
9644     return new vector_iterator {
9645         .vector = self,
9646         .index = 0,

```

```

9646     };
9647 };
9648 `)
9649
9650     return ctx.NewBuiltin("vector::into_iterator", []Type{TVAl(VECTOR)}, func(ctx
↪ *Context, arguments []Value) (Value, error) {
9651         return CallBuiltinFromSource(ctx, function, arguments)
9652     })
9653 }
9654
9655 func BuiltinMapInit(ctx *Context) Value {
9656     return ctx.NewBuiltin("map::init", []Type{TVAl(ANY)}, func(ctx *Context, arguments
↪ []Value) (Value, error) {
9657         value := arguments[0]
9658
9659         if metaFunction, ok := MetaFunction(ctx, value, ctx.constStringNext); ok {
9660             if !callableSelfIsPassedByReference(metaFunction) {
9661                 return nil, NewError(nil, ctx.NewString("iterator next must receive self
↪ by reference (declared with function.&(self))"))
9662             }
9663             reference := ctx.newReference(value)
9664             defer unmarkReferenced(value)
9665             result := ctx.NewMapOrPanic(nil)
9666             for {
9667                 iterated, err := call(ctx, nil, metaFunction, []Value{reference})
9668                 if err != nil {
9669                     if error, ok := err.(Error); ok {
9670                         if _, ok := error.Value.(*Null); ok {
9671                             break // end-of-iteration
9672                         }
9673                     }
9674                     return nil, err
9675                 }
9676                 iteratedVector, ok := iterated.(*Vector)
9677                 if !ok || iteratedVector.Count() != 2 {
9678                     return nil, NewError(nil, ctx.NewStringf("iterator next must return a
↪ two element vector, received %s", iterated))
9679                 }
9680                 err = result.Insert(iteratedVector.Get(0).Copy(),
↪ iteratedVector.Get(1).Copy())
9681                 if err != nil {
9682                     return nil, NewError(nil, ctx.NewString(err.Error()))
9683                 }
9684             }
9685             return result, nil
9686         }
9687
9688         if valueVector, ok := value.(*Vector); ok {
9689             result := ctx.NewMapOrPanic(nil)
9690             for _, element := range valueVector.Elements() {
9691                 elementVector, ok := element.(*Vector)
9692                 if !ok || elementVector.Count() != 2 {
9693                     return nil, NewError(nil, ctx.NewStringf("expected vector element to
↪ be a two element vector, received %s", element))
9694                 }
9695                 err := result.Insert(elementVector.Get(0).Copy(),
↪ elementVector.Get(1).Copy())
9696                 if err != nil {
9697                     return nil, NewError(nil, ctx.NewString(err.Error()))
9698                 }
9699             }
9700             return result, nil
9701         }
9702
9703         if valueMap, ok := value.(*Map); ok {
9704             result := ctx.NewMapOrPanic(nil)
9705             for _, pair := range valueMap.Pairs() {

```

```

9706         err := result.Insert(pair.Key.Copy(), pair.Value.Copy())
9707         if err != nil {
9708             return nil, NewError(nil, ctx.NewString(err.Error()))
9709         }
9710     }
9711     return result, nil
9712 }
9713
9714     if valueSet, ok := value.(*Set); ok {
9715         result := ctx.NewMapOrPanic(nil)
9716         for _, element := range valueSet.Elements() {
9717             elementVector, ok := element.(*Vector)
9718             if !ok || elementVector.Count() != 2 {
9719                 return nil, NewError(nil, ctx.NewStringf("expected set element to be
↪ a two element vector, received %s", element))
9720             }
9721             err := result.Insert(elementVector.Get(0).Copy(),
↪ elementVector.Get(1).Copy())
9722             if err != nil {
9723                 return nil, NewError(nil, ctx.NewString(err.Error()))
9724             }
9725         }
9726         return result, nil
9727     }
9728
9729     return nil, NewError(nil, ctx.NewStringf("cannot convert value %v to map",
↪ value))
9730 })
9731 }
9732
9733 func BuiltinMapCount(ctx *Context) Value {
9734     return ctx.NewBuiltin("map::count", []Type{TVal(MAP)}, func(ctx *Context, arguments
↪ []Value) (Value, error) {
9735         self := arguments[0].(*Map)
9736
9737         return ctx.NewNumber(float64(self.Count())), nil
9738     })
9739 }
9740
9741 func BuiltinMapIsEmpty(ctx *Context) Value {
9742     return ctx.NewBuiltin("map::is_empty", []Type{TVal(MAP)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
9743         self := arguments[0].(*Map)
9744
9745         return ctx.NewBoolean(self.Count() == 0), nil
9746     })
9747 }
9748
9749 func BuiltinMapContains(ctx *Context) Value {
9750     return ctx.NewBuiltin("map::contains", []Type{TVal(MAP), TVal(ANY)}, func(ctx
↪ *Context, arguments []Value) (Value, error) {
9751         self := arguments[0].(*Map)
9752         target := arguments[1]
9753
9754         _, ok := self.Lookup(target)
9755
9756         return ctx.NewBoolean(ok), nil
9757     })
9758 }
9759
9760 func BuiltinMapInsert(ctx *Context) Value {
9761     return ctx.NewBuiltinWithSelfByReference("map::insert", []Type{TRef(TVal(MAP)),
↪ TVal(ANY), TVal(ANY)}, func(ctx *Context, arguments []Value) (Value, error) {
9762         self := arguments[0].(*Reference)
9763         delf := self.data.(*Map)
9764
9765         k := arguments[1]

```

```

9766         v := arguments[2]
9767
9768         err := delf.Insert(k.Copy(), v.Copy())
9769         if err != nil {
9770             return nil, NewError(nil, ctx.NewStringf("invalid map::insert operation
↪ (%s)", err.Error()))
9771         }
9772
9773         return ctx.NewNull(), nil
9774     })
9775 }
9776
9777 func BuiltinMapRemove(ctx *Context) Value {
9778     return ctx.NewBuiltinWithSelfByReference("map::remove", []Type{TRef(TVal(MAP))},
↪ TVal(ANY)}, func(ctx *Context, arguments []Value) (Value, error) {
9779         self := arguments[0].(*Reference)
9780         delf := self.data.(*Map)
9781
9782         k := arguments[1]
9783
9784         lookup, ok := delf.Lookup(k)
9785         if !ok {
9786             return nil, NewError(nil, ctx.NewStringf("attempted map::remove on a map
↪ without key %v", k))
9787         }
9788
9789         err := delf.Remove(k)
9790         if err != nil {
9791             return nil, NewError(nil, ctx.NewStringf("invalid map::remove operation
↪ (%s)", err.Error()))
9792         }
9793
9794         return lookup.Copy(), nil
9795     })
9796 }
9797
9798 func BuiltinMapGet(ctx *Context) Value {
9799     return ctx.NewBuiltin("map::get", []Type{TVal(MAP), TVal(ANY), TVal(ANY)}, func(ctx
↪ *Context, arguments []Value) (Value, error) {
9800         self := arguments[0].(*Map)
9801         k := arguments[1]
9802         d := arguments[2]
9803
9804         lookup, ok := self.Lookup(k)
9805         if ok {
9806             return lookup.Copy(), nil
9807         }
9808
9809         return d.Copy(), nil
9810     })
9811 }
9812
9813 func BuiltinMapKeys(ctx *Context) Value {
9814     return ctx.NewBuiltin("map::keys", []Type{TVal(MAP)}, func(ctx *Context, arguments
↪ []Value) (Value, error) {
9815         self := arguments[0].(*Map)
9816
9817         pairs := self.Pairs()
9818         result := ctx.NewVectorOrPanic(nil)
9819         for _, pair := range pairs {
9820             err := result.Push(pair.Key.Copy())
9821             if err != nil {
9822                 return nil, NewError(nil, ctx.NewString(err.Error()))
9823             }
9824         }
9825
9826         return result, nil

```

```

9827     })
9828 }
9829
9830 func BuiltinMapValues(ctx *Context) Value {
9831     return ctx.NewBuiltin("map::values", []Type{TVal(MAP)}, func(ctx *Context, arguments
9832 ↪ []Value) (Value, error) {
9833         self := arguments[0].(*Map)
9834
9835         pairs := self.Pairs()
9836         result := ctx.NewVectorOrPanic(nil)
9837         for _, pair := range pairs {
9838             err := result.Push(pair.Value.Copy())
9839             if err != nil {
9840                 return nil, NewError(nil, ctx.NewString(err.Error()))
9841             }
9842         }
9843         return result, nil
9844     })
9845 }
9846
9847 func BuiltinMapPairs(ctx *Context) Value {
9848     return ctx.NewBuiltin("map::pairs", []Type{TVal(MAP)}, func(ctx *Context, arguments
9849 ↪ []Value) (Value, error) {
9850         self := arguments[0].(*Map)
9851
9852         pairs := self.Pairs()
9853         result := ctx.NewVectorOrPanic(nil)
9854         for _, pair := range pairs {
9855             pair, err := ctx.NewMap([]MapPair{
9856                 {ctx.NewString("key"), pair.Key.Copy()},
9857                 {ctx.NewString("value"), pair.Value.Copy()}},
9858             )
9859             if err != nil {
9860                 return nil, NewError(nil, ctx.NewString(err.Error()))
9861             }
9862             err = result.Push(pair)
9863             if err != nil {
9864                 return nil, NewError(nil, ctx.NewString(err.Error()))
9865             }
9866         }
9867         return result, nil
9868     })
9869 }
9870
9871 func BuiltinMapUnion(ctx *Context) Value {
9872     function := ctx.NewValueFromSourceOrPanic("map::union", `
9873 return function(a, b) {
9874     if not ty::is_map(a) or not ty::is_map(b) {
9875         error $"attempted map::union of values {repr(a)} and {repr(b)}";
9876     }
9877
9878     let result = Map{};
9879     for k, v in a {
9880         map::insert(result.&, k, v);
9881     }
9882     for k, v in b {
9883         map::insert(result.&, k, v);
9884     }
9885     return result;
9886 };
9887 `)
9888
9889     return ctx.NewBuiltin("map::union", []Type{TVal(ANY), TVal(ANY)}, func(ctx *Context,
9890 ↪ arguments []Value) (Value, error) {
9891         return CallBuiltinFromSource(ctx, function, arguments)
9892     })

```

```

9891     })
9892 }
9893
9894 func BuiltinMapIntoIterator(ctx *Context) Value {
9895     function := ctx.NewValueFromSourceOrPanic("map::into_iterator", `
9896     return function(self) {
9897         let map_iterator = type extends iterator {
9898             .next = function.&(self) {
9899                 if self.index >= self.pairs.count() {
9900                     return iterator::eoi();
9901                 }
9902                 let current = self.pairs[self.index];
9903                 self.index = self.index + 1;
9904                 return current;
9905             },
9906         };
9907         return new map_iterator {
9908             .pairs = vector::init(self),
9909             .index = 0,
9910         };
9911     };
9912 `)
9913
9914     return ctx.NewBuiltin("map::into_iterator", []Type{TVal(MAP)}, func(ctx *Context,
9915 ↪ arguments []Value) (Value, error) {
9916         return CallBuiltinFromSource(ctx, function, arguments)
9917     })
9918 }
9919
9920 func BuiltinSetInit(ctx *Context) Value {
9921     return ctx.NewBuiltin("set::init", []Type{TVal(ANY)}, func(ctx *Context, arguments
9922 ↪ []Value) (Value, error) {
9923         value := arguments[0]
9924
9925         if metaFunction, ok := MetaFunction(ctx, value, ctx.constStringNext); ok {
9926             if !callableSelfIsPassedByReference(metaFunction) {
9927                 return nil, NewError(nil, ctx.NewString("iterator next must receive self
9928 ↪ by reference (declared with function.&(self))"))
9929             }
9930             reference := ctx.newReference(value)
9931             defer unmarkReferenced(value)
9932             result := ctx.NewSetOrPanic(nil)
9933             for {
9934                 iterated, err := call(ctx, nil, metaFunction, []Value{reference})
9935                 if err != nil {
9936                     if error, ok := err.(Error); ok {
9937                         if _, ok := error.Value.(*Null); ok {
9938                             break // end-of-iteration
9939                         }
9940                     }
9941                     return nil, err
9942                 }
9943                 err = result.Insert(iterated.Copy())
9944                 if err != nil {
9945                     return nil, NewError(nil, ctx.NewString(err.Error()))
9946                 }
9947             }
9948             return result, nil
9949         }
9950
9951         if valueVector, ok := value.(*Vector); ok {
9952             result := ctx.NewSetOrPanic(nil)
9953             for _, element := range valueVector.Elements() {
9954                 err := result.Insert(element.Copy())
9955                 if err != nil {
9956                     return nil, NewError(nil, ctx.NewString(err.Error()))
9957                 }
9958             }
9959         }
9960     })

```

```

9955         }
9956         return result, nil
9957     }
9958
9959     if valueMap, ok := value.(*Map); ok {
9960         result := ctx.NewSetOrPanic(nil)
9961         for _, pair := range valueMap.Pairs() {
9962             err := result.Insert(ctx.NewVectorOrPanic([]Value{pair.Key.Copy(),
↪ pair.Value.Copy()}))
9963             if err != nil {
9964                 return nil, NewError(nil, ctx.NewString(err.Error()))
9965             }
9966         }
9967         return result, nil
9968     }
9969
9970     if valueSet, ok := value.(*Set); ok {
9971         result := ctx.NewSetOrPanic(nil)
9972         for _, element := range valueSet.Elements() {
9973             err := result.Insert(element.Copy())
9974             if err != nil {
9975                 return nil, NewError(nil, ctx.NewString(err.Error()))
9976             }
9977         }
9978         return result, nil
9979     }
9980
9981     return nil, NewError(nil, ctx.NewStringf("cannot convert value %v to set",
↪ value))
9982 })
9983 }
9984
9985 func BuiltinSetCount(ctx *Context) Value {
9986     return ctx.NewBuiltin("set::count", []Type{TVal(SET)}, func(ctx *Context, arguments
↪ []Value) (Value, error) {
9987         self := arguments[0].(*Set)
9988
9989         return ctx.NewNumber(float64(self.Count())), nil
9990     })
9991 }
9992
9993 func BuiltinSetIsEmpty(ctx *Context) Value {
9994     return ctx.NewBuiltin("set::is_empty", []Type{TVal(SET)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
9995         self := arguments[0].(*Set)
9996
9997         return ctx.NewBoolean(self.Count() == 0), nil
9998     })
9999 }
10000
10001 func BuiltinSetContains(ctx *Context) Value {
10002     return ctx.NewBuiltin("set::contains", []Type{TVal(SET), TVal(ANY)}, func(ctx
↪ *Context, arguments []Value) (Value, error) {
10003         self := arguments[0].(*Set)
10004         target := arguments[1]
10005
10006         _, ok := self.Lookup(target)
10007
10008         return ctx.NewBoolean(ok), nil
10009     })
10010 }
10011
10012 func BuiltinSetInsert(ctx *Context) Value {
10013     return ctx.NewBuiltinWithSelfByReference("set::insert", []Type{TRef(TVal(SET)),
↪ TVal(ANY)}, func(ctx *Context, arguments []Value) (Value, error) {
10014         self := arguments[0].(*Reference)
10015         delf := self.data.(*Set)

```

```

10016         element := arguments[1]
10017
10018         err := delf.Insert(element.Copy())
10019         if err != nil {
10020             return nil, NewError(nil, ctx.NewStringf("invalid set::insert operation
↪ (%s)", err.Error()))
10021         }
10022
10023         return ctx.NewNull(), nil
10024     })
10025 }
10026
10027 func BuiltinSetRemove(ctx *Context) Value {
10028     return ctx.NewBuiltinWithSelfByReference("set::remove", []Type{TRef(TVal(SET)),
↪ TVal(ANY)}, func(ctx *Context, arguments []Value) (Value, error) {
10029         self := arguments[0].(*Reference)
10030         delf := self.data.(*Set)
10031         element := arguments[1]
10032
10033         lookup, ok := delf.Lookup(element)
10034         if !ok {
10035             return nil, NewError(nil, ctx.NewStringf("attempted set::remove on a set
↪ without element %v", element))
10036         }
10037
10038         err := delf.Remove(element)
10039         if err != nil {
10040             return nil, NewError(nil, ctx.NewStringf("invalid set::remove operation
↪ (%s)", err.Error()))
10041         }
10042
10043         return lookup.Copy(), nil
10044     })
10045 }
10046
10047 func BuiltinSetUnion(ctx *Context) Value {
10048     function := ctx.NewValueFromSourceOrPanic("set::union", `
10049 return function(a, b) {
10050     if not ty::is_set(a) or not ty::is_set(b) {
10051         error $"attempted set::union of values {repr(a)} and {repr(b)}";
10052     }
10053
10054     let result = Set{};
10055     for x in a {
10056         set::insert(result.&, x);
10057     }
10058     for x in b {
10059         set::insert(result.&, x);
10060     }
10061     return result;
10062 };
10063 `)
10064
10065     return ctx.NewBuiltin("set::union", []Type{TVal(ANY), TVal(ANY)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
10066         return CallBuiltinFromSource(ctx, function, arguments)
10067     })
10068 }
10069
10070 func BuiltinSetIntersection(ctx *Context) Value {
10071     function := ctx.NewValueFromSourceOrPanic("set::intersection", `
10072 return function(a, b) {
10073     if not ty::is_set(a) or not ty::is_set(b) {
10074         error $"attempted set::intersection of values {repr(a)} and {repr(b)}";
10075     }
10076
10077     let result = Set{};

```

```

10078     for x in a {
10079         if b.contains(x) {
10080             set::insert(result.&, x);
10081         }
10082     }
10083     return result;
10084 };
10085 `)
10086
10087     return ctx.NewBuiltin("set::intersection", []Type{TVal(ANY), TVal(ANY)}, func(ctx
↵ *Context, arguments []Value) (Value, error) {
10088         return CallBuiltinFromSource(ctx, function, arguments)
10089     })
10090 }
10091
10092 func BuiltinSetDifference(ctx *Context) Value {
10093     function := ctx.NewValueFromSourceOrPanic("set::difference", `
10094 return function(a, b) {
10095     if not ty::is_set(a) or not ty::is_set(b) {
10096         error $"attempted set::difference of values {repr(a)} and {repr(b)}";
10097     }
10098
10099     let result = Set{};
10100     for x in a {
10101         if not b.contains(x) {
10102             set::insert(result.&, x);
10103         }
10104     }
10105     return result;
10106 };
10107 `)
10108
10109     return ctx.NewBuiltin("set::difference", []Type{TVal(ANY), TVal(ANY)}, func(ctx
↵ *Context, arguments []Value) (Value, error) {
10110         return CallBuiltinFromSource(ctx, function, arguments)
10111     })
10112 }
10113
10114 func BuiltinSetIntoIterator(ctx *Context) Value {
10115     function := ctx.NewValueFromSourceOrPanic("set::into_iterator", `
10116 return function(self) {
10117     let set_iterator = type extends iterator {
10118         .next = function.&(self) {
10119             if self.index >= self.elements.count() {
10120                 return iterator::eoi();
10121             }
10122             let current = self.elements[self.index];
10123             self.index = self.index + 1;
10124             return current;
10125         },
10126     };
10127     return new set_iterator {
10128         .elements = vector::init(self),
10129         .index = 0,
10130     };
10131 };
10132 `)
10133
10134     return ctx.NewBuiltin("set::into_iterator", []Type{TVal(SET)}, func(ctx *Context,
↵ arguments []Value) (Value, error) {
10135         return CallBuiltinFromSource(ctx, function, arguments)
10136     })
10137 }
10138
10139 func BuiltinAssert(ctx *Context) Value {
10140     function := ctx.NewValueFromSourceOrPanic("assert", `
10141 let assert = function(condition) {

```

```

10142     if not condition {
10143         error "assertion failure";
10144     }
10145 };
10146 return assert;
10147 `)
10148
10149     return ctx.NewBuiltin("assert", []Type{TVal(BOOLEAN)}, func(ctx *Context, arguments
10150 ↪ []Value) (Value, error) {
10151         return CallBuiltinFromSource(ctx, function, arguments)
10152     })
10153 }
10154
10155 func BuiltinTypeof(ctx *Context) Value {
10156     return ctx.NewBuiltin("typeof", []Type{TVal(ANY)}, func(ctx *Context, arguments
10157 ↪ []Value) (Value, error) {
10158         value := arguments[0]
10159
10160         if value.Meta(ctx) == nil {
10161             return ctx.NewNull(), nil
10162         }
10163
10164         return value.Meta(ctx), nil
10165     })
10166 }
10167
10168 func BuiltinTypename(ctx *Context) Value {
10169     return ctx.NewBuiltin("typename", []Type{TVal(ANY)}, func(ctx *Context, arguments
10170 ↪ []Value) (Value, error) {
10171         return ctx.NewString(arguments[0].Typename()), nil
10172     })
10173 }
10174
10175 func BuiltinRepr(ctx *Context) Value {
10176     return ctx.NewBuiltin("repr", []Type{TVal(ANY)}, func(ctx *Context, arguments
10177 ↪ []Value) (Value, error) {
10178         return ctx.NewStringf("%v", arguments[0]), nil
10179     })
10180 }
10181
10182 func BuiltinDump(ctx *Context) Value {
10183     return ctx.NewBuiltin("dump", []Type{TVal(ANY)}, func(ctx *Context, arguments
10184 ↪ []Value) (Value, error) {
10185         fmt.Fprintf(ctx.Stdout, "%v", arguments[0])
10186         return ctx.NewNull(), nil
10187     })
10188 }
10189
10190 func BuiltinDumpln(ctx *Context) Value {
10191     return ctx.NewBuiltin("dumpln", []Type{TVal(ANY)}, func(ctx *Context, arguments
10192 ↪ []Value) (Value, error) {
10193         fmt.Fprintf(ctx.Stdout, "%v\n", arguments[0])
10194         return ctx.NewNull(), nil
10195     })
10196 }
10197
10198 func BuiltinPrint(ctx *Context) Value {
10199     return ctx.NewBuiltin("print", []Type{TVal(ANY)}, func(ctx *Context, arguments
10200 ↪ []Value) (Value, error) {
10201         if metaFunction, ok := MetaFunction(ctx, arguments[0],
10202 ↪ ctx.constStringIntoString); ok {
10203             result, err := ctx.callMetaWithSelf(nil, metaFunction, arguments[0])
10204             if err != nil {
10205                 return nil, err
10206             }
10207             resultString, ok := result.(*String)
10208             if !ok {

```

```

10201         return nil, NewError(
10202             nil,
10203             ctx.NewStringf("metafunction %s returned %v",
↪ quote(ctx.constStringIntoString.data), result),
10204         )
10205     }
10206     fmt.Fprintf(ctx.Stdout, "%s", resultString.data)
10207     return ctx.NewNull(), nil
10208 }
10209
10210     if string, ok := arguments[0].(*String); ok {
10211         fmt.Fprintf(ctx.Stdout, "%s", string.data)
10212         return ctx.NewNull(), nil
10213     }
10214
10215     fmt.Fprintf(ctx.Stdout, "%v", arguments[0])
10216     return ctx.NewNull(), nil
10217 })
10218 }
10219
10220 func BuiltinPrintln(ctx *Context) Value {
10221     return ctx.NewBuiltin("println", []Type{TVal(ANY)}, func(ctx *Context, arguments
↪ []Value) (Value, error) {
10222         if metaFunction, ok := MetaFunction(ctx, arguments[0],
↪ ctx.constStringIntoString); ok {
10223             result, err := ctx.callMetaWithSelf(nil, metaFunction, arguments[0])
10224             if err != nil {
10225                 return nil, err
10226             }
10227             resultString, ok := result.(*String)
10228             if !ok {
10229                 return nil, NewError(
10230                     nil,
10231                     ctx.NewStringf("metafunction %s returned %v",
↪ quote(ctx.constStringIntoString.data), result),
10232                 )
10233             }
10234             fmt.Fprintf(ctx.Stdout, "%s\n", resultString.data)
10235             return ctx.NewNull(), nil
10236         }
10237
10238         if string, ok := arguments[0].(*String); ok {
10239             fmt.Fprintf(ctx.Stdout, "%s\n", string.data)
10240             return ctx.NewNull(), nil
10241         }
10242
10243         fmt.Fprintf(ctx.Stdout, "%v\n", arguments[0])
10244         return ctx.NewNull(), nil
10245     })
10246 }
10247
10248 func BuiltinEprint(ctx *Context) Value {
10249     return ctx.NewBuiltin("eprint", []Type{TVal(ANY)}, func(ctx *Context, arguments
↪ []Value) (Value, error) {
10250         if metaFunction, ok := MetaFunction(ctx, arguments[0],
↪ ctx.constStringIntoString); ok {
10251             result, err := ctx.callMetaWithSelf(nil, metaFunction, arguments[0])
10252             if err != nil {
10253                 return nil, err
10254             }
10255             resultString, ok := result.(*String)
10256             if !ok {
10257                 return nil, NewError(
10258                     nil,
10259                     ctx.NewStringf("metafunction %s returned %v",
↪ quote(ctx.constStringIntoString.data), result),
10260                 )

```

```

10261     }
10262     fmt.Fprintf(ctx.Stderr, "%s", resultString.data)
10263     return ctx.NewNull(), nil
10264 }
10265
10266 if string, ok := arguments[0].(*String); ok {
10267     fmt.Fprintf(ctx.Stderr, "%s", string.data)
10268     return ctx.NewNull(), nil
10269 }
10270
10271 fmt.Fprintf(ctx.Stderr, "%v", arguments[0])
10272 return ctx.NewNull(), nil
10273 })
10274 }
10275
10276 func BuiltinEprintln(ctx *Context) Value {
10277     return ctx.NewBuiltin("eprintln", []Type{TVAl(ANY)}, func(ctx *Context, arguments
↪ []Value) (Value, error) {
10278         if metaFunction, ok := MetaFunction(ctx, arguments[0],
↪ ctx.constStringIntoString); ok {
10279             result, err := ctx.callMetaWithSelf(nil, metaFunction, arguments[0])
10280             if err != nil {
10281                 return nil, err
10282             }
10283             resultString, ok := result.(*String)
10284             if !ok {
10285                 return nil, NewError(
10286                     nil,
10287                     ctx.NewStringf("metafunction %s returned %v",
↪ quote(ctx.constStringIntoString.data), result),
10288                 )
10289             }
10290             fmt.Fprintf(ctx.Stderr, "%s\n", resultString.data)
10291             return ctx.NewNull(), nil
10292         }
10293
10294         if string, ok := arguments[0].(*String); ok {
10295             fmt.Fprintf(ctx.Stderr, "%s\n", string.data)
10296             return ctx.NewNull(), nil
10297         }
10298
10299         fmt.Fprintf(ctx.Stderr, "%v\n", arguments[0])
10300         return ctx.NewNull(), nil
10301     })
10302 }
10303
10304 func BuiltinRange(ctx *Context) Value {
10305     function := ctx.NewValueFromSourceOrPanic("range", `
10306 let range_iterator = type extends iterator {
10307     "init": function(bgn, end) {
10308         if end < bgn {
10309             error $"end-of-range {repr(end)} is less than beginning-of-range
↪ {repr(bgn)}";
10310         }
10311         return new range_iterator {
10312             .cur = bgn,
10313             .end = end,
10314         };
10315     },
10316     "next": function.&(self) {
10317         if self.cur >= self.end {
10318             error null; # end-of-iteration
10319         }
10320         let result = self.cur;
10321         self.cur = self.cur + 1;
10322         return result;
10323     },

```

```

10324 };
10325 let range = function(bgn, end) {
10326     return range_iterator::init(bgn, end);
10327 };
10328 return range;
10329 `)
10330
10331     return ctx.NewBuiltin("range", []Type{TVal(NUMBER), TVal(NUMBER)}, func(ctx *Context,
10332 ↪ arguments []Value) (Value, error) {
10333         return CallBuiltinFromSource(ctx, function, arguments)
10334     })
10335 }
10336
10337 func BuiltinMin(ctx *Context) Value {
10338     function := ctx.NewValueFromSourceOrPanic("min", `
10339 let min = function(a, b) {
10340     if a <= b {
10341         return a;
10342     }
10343     return b;
10344 };
10345 return min;
10346 `)
10347     return ctx.NewBuiltin("min", []Type{TVal(ANY), TVal(ANY)}, func(ctx *Context,
10348 ↪ arguments []Value) (Value, error) {
10349         return CallBuiltinFromSource(ctx, function, arguments)
10350     })
10351 }
10352
10353 func BuiltinMax(ctx *Context) Value {
10354     function := ctx.NewValueFromSourceOrPanic("max", `
10355 let max = function(a, b) {
10356     if a >= b {
10357         return a;
10358     }
10359     return b;
10360 };
10361 return max;
10362 `)
10363     return ctx.NewBuiltin("max", []Type{TVal(ANY), TVal(ANY)}, func(ctx *Context,
10364 ↪ arguments []Value) (Value, error) {
10365         return CallBuiltinFromSource(ctx, function, arguments)
10366     })
10367 }
10368
10369 // XXX: Parse, don't validate! Currently, this implementation validates an
10370 // already parsed AST (1) to avoid duplicating parsing code, and (2) because it
10371 // is unclear what the best architecture for breaking out comb encoding would be
10372 // (separate parse mode? separate parser type?).
10373 func combValidate(ctx *Context, expr AstExpression) error {
10374     validationError := func() error {
10375         found := reflect.TypeOf(expr).Elem().Name()
10376         if x, ok := expr.(*AstExpressionIdentifier); ok {
10377             found = x.Name.data
10378         }
10379         return NewError(expr.ExpressionLocation(), ctx.NewStringf("expected expression,
10380 ↪ found %s", found))
10381     }
10382
10383     if _, ok := expr.(*AstExpressionNull); ok {
10384         return nil
10385     }
10386
10387     if _, ok := expr.(*AstExpressionBoolean); ok {
10388         return nil
10389     }

```

```

10387     }
10388
10389     if _, ok := expr.(*AstExpressionNumber); ok {
10390         return nil
10391     }
10392
10393     if _, ok := expr.(*AstExpressionString); ok {
10394         return nil
10395     }
10396
10397     if x, ok := expr.(*AstExpressionVector); ok {
10398         for _, element := range x.Elements {
10399             if err := combValidate(ctx, element); err != nil {
10400                 return err
10401             }
10402         }
10403         return nil
10404     }
10405
10406     if x, ok := expr.(*AstExpressionMap); ok {
10407         for _, pair := range x.Elements {
10408             if err := combValidate(ctx, pair.Key); err != nil {
10409                 return err
10410             }
10411             if err := combValidate(ctx, pair.Value); err != nil {
10412                 return err
10413             }
10414         }
10415         return nil
10416     }
10417
10418     if x, ok := expr.(*AstExpressionSet); ok {
10419         for _, element := range x.Elements {
10420             if err := combValidate(ctx, element); err != nil {
10421                 return err
10422             }
10423         }
10424         return nil
10425     }
10426
10427     if x, ok := expr.(*AstExpressionPositive); ok {
10428         if err := combValidate(ctx, x.Expression); err != nil {
10429             return err
10430         }
10431         if _, isNumber := x.Expression.(*AstExpressionNumber); !isNumber {
10432             return validationError()
10433         }
10434         return nil
10435     }
10436
10437     if x, ok := expr.(*AstExpressionNegative); ok {
10438         if err := combValidate(ctx, x.Expression); err != nil {
10439             return err
10440         }
10441         if _, isNumber := x.Expression.(*AstExpressionNumber); !isNumber {
10442             return validationError()
10443         }
10444         return nil
10445     }
10446
10447     return validationError()
10448 }
10449
10450 func BuiltinCombDecode(ctx *Context) Value {
10451     return ctx.NewBuiltin("comb::decode", []Type{TVal(STRING)}, func(ctx *Context,
10452 ↪ arguments []Value) (Value, error) {
10453         encoded := arguments[0].(*String)

```

```

10453     lexer := NewLexer(ctx, encoded.data, &SourceLocation{"comb::decode", 1})
10454     parser, err := NewParser(&lexer)
10455     if err != nil {
10456         return nil, err
10457     }
10458     expr, err := parser.ParseExpression()
10459     if err != nil {
10460         if e, ok := err.(ParseError); ok {
10461             return nil, NewError(e.Location, ctx.NewString(e.Error()))
10462         }
10463         return nil, err
10464     }
10465     if parser.currentToken.Kind != TOKEN_EOF {
10466         return nil, NewError(
10467             parser.currentToken.Location,
10468             ctx.NewStringf("expected %s, found %v", TOKEN_EOF, parser.currentToken),
10469         )
10470     }
10471     err = combValidate(ctx, expr)
10472     if err != nil {
10473         return nil, err
10474     }
10475     env := NewEnvironment(nil)
10476     result, err := expr.Eval(ctx, env)
10477     env.tryReuse()
10478     return result, err
10479 })
10480 }
10481
10482 func BuiltinCombEncode(ctx *Context) Value {
10483     return ctx.NewBuiltin("comb::encode", []Type{TVal(ANY)}, func(ctx *Context, arguments
10484 ↪ []Value) (Value, error) {
10485         var sb strings.Builder
10486         encoder := NewCombEncoder(&sb)
10487         err := arguments[0].CombEncode(encoder)
10488         if err != nil {
10489             return nil, NewError(nil, ctx.NewString(err.Error()))
10490         }
10491         return ctx.NewString(sb.String()), nil
10492     })
10493 }
10494
10495 func BuiltinCombEncodeEx(ctx *Context) Value {
10496     return ctx.NewBuiltin("comb::encode_ex", []Type{TVal(ANY), TVal(MAP)}, func(ctx
10497 ↪ *Context, arguments []Value) (Value, error) {
10498         options := arguments[1].(*Map)
10499         var indent *string = nil // optional
10500         var separator string = ""
10501         for _, pair := range options.Pairs() {
10502             if k, ok := pair.Key.(*String); ok && k.data == "indent" {
10503                 if v, ok := pair.Value.(*Number); ok {
10504                     vInt, err := ValueAsSafeInteger(v)
10505                     if err == nil && vInt >= 0 {
10506                         empty := ""
10507                         indent = &empty
10508                         for _ = range vInt {
10509                             *indent = *indent + " "
10510                         }
10511                         continue
10512                     }
10513                 }
10514             }
10515             if v, ok := pair.Value.(*String); ok {
10516                 indent = &v.data
10517                 continue

```

```

10518         }
10519
10520         return nil, NewError(nil, ctx.NewStringf("expected non-negative integer
↳ or string indent, received %v", pair.Value))
10521     }
10522
10523     if k, ok := pair.Key.(*String); ok && k.data == "separator" {
10524         if v, ok := pair.Value.(*String); ok {
10525             separator = v.data
10526             continue
10527         }
10528
10529         return nil, NewError(nil, ctx.NewStringf("expected string separator,
↳ received %v", pair.Value))
10530     }
10531
10532     return nil, NewError(nil, ctx.NewStringf("unknown option %v", pair.Key))
10533 }
10534
10535 var sb strings.Builder
10536 encoder := NewCombEncoder(&sb)
10537 encoder.Indent = indent
10538 encoder.Separator = separator
10539 err := arguments[0].CombEncode(encoder)
10540 if err != nil {
10541     return nil, NewError(nil, ctx.NewString(err.Error()))
10542 }
10543
10544 return ctx.NewString(sb.String()), nil
10545 })
10546 }
10547
10548 func BuiltinHtmlEscape(ctx *Context) Value {
10549     return ctx.NewBuiltin("html::escape", []Type{TVal(STRING)}, func(ctx *Context,
↳ arguments []Value) (Value, error) {
10550         text := arguments[0].(*String)
10551
10552         // Although &apos; is not a part of the HTML4 standard, this function
10553         // chooses to normalize on HTML5 rather than keep compatibility with
10554         // legacy browsers like IE.
10555         escaper := strings.NewReplacer(
10556             `&`, "&amp;",
10557             `\'`, "&apos;",
10558             `\"`, "&quot;",
10559             `<`, "&lt;",
10560             `>`, "&gt;",
10561         )
10562
10563         return ctx.NewString(escaper.Replace(text.data)), nil
10564     })
10565 }
10566
10567 func jsonDecodeFromTokens(ctx *Context, dec *json.Decoder) (Value, error) {
10568     t, err := dec.Token()
10569     if err != nil {
10570         return nil, err
10571     }
10572
10573     if t == nil {
10574         return ctx.NewNull(), nil
10575     }
10576
10577     switch v := t.(type) {
10578     case bool:
10579         return ctx.NewBoolean(v), nil
10580     case float64:
10581         return ctx.NewNumber(v), nil

```

```

10582     case string:
10583         return ctx.NewString(v), nil
10584     case json.Delim:
10585         switch v {
10586             case '[':
10587                 elements := []Value{}
10588                 for dec.More() {
10589                     element, err := jsonDecodeFromTokens(ctx, dec)
10590                     if err != nil {
10591                         return nil, err
10592                     }
10593                     elements = append(elements, element)
10594                 }
10595                 // consume the closing ']'
10596                 if _, err := dec.Token(); err != nil {
10597                     return nil, err
10598                 }
10599                 return ctx.NewVectorOrPanic(elements), nil
10600             case '{':
10601                 pairs := []MapPair{}
10602                 for dec.More() {
10603                     keyToken, err := dec.Token()
10604                     if err != nil {
10605                         return nil, err
10606                     }
10607                     keyStr, ok := keyToken.(string)
10608                     if !ok {
10609                         return nil, fmt.Errorf("expected string key in JSON object")
10610                     }
10611                     key := ctx.NewString(keyStr)
10612                     value, err := jsonDecodeFromTokens(ctx, dec)
10613                     if err != nil {
10614                         return nil, err
10615                     }
10616                     pairs = append(pairs, MapPair{key, value})
10617                 }
10618                 // consume the closing '}'
10619                 if _, err := dec.Token(); err != nil {
10620                     return nil, err
10621                 }
10622                 return ctx.NewMap(pairs)
10623             }
10624         }
10625
10626     return nil, fmt.Errorf("cannot JSON-decode type %T", t)
10627 }
10628
10629 func BuiltinJsonDecode(ctx *Context) Value {
10630     return ctx.NewBuiltin("json::decode", []Type{TVal(STRING)}, func(ctx *Context,
10631 ↪ arguments []Value) (Value, error) {
10632         encoded := arguments[0].(*String)
10633
10634         dec := json.NewDecoder(strings.NewReader(encoded.data))
10635         decoded, err := jsonDecodeFromTokens(ctx, dec)
10636         if err != nil {
10637             return nil, NewError(nil, ctx.NewStringf("cannot JSON-decode string %v",
10638 ↪ encoded))
10639         }
10640
10641         // Check for trailing data after the first JSON value
10642         if dec.More() {
10643             return nil, NewError(nil, ctx.NewStringf("cannot JSON-decode string %v",
10644 ↪ encoded))
10645         }
10646
10647         return decoded, nil
10648     })
10649 }

```

```

10646 }
10647
10648 // orderedMapMarshaler preserves key order when marshaling JSON,
10649 // solving the non-deterministic map iteration of encoding/json.
10650 type orderedMapMarshaler []orderedMapEntry
10651
10652 type orderedMapEntry struct {
10653     Key    string
10654     Value  any
10655 }
10656
10657 func (m orderedMapMarshaler) MarshalJSON() ([]byte, error) {
10658     var buf strings.Builder
10659     buf.WriteByte('{')
10660     for i, entry := range m {
10661         if i > 0 {
10662             buf.WriteByte(',')
10663         }
10664         keyBytes, err := json.Marshal(entry.Key)
10665         if err != nil {
10666             return nil, err
10667         }
10668         buf.Write(keyBytes)
10669         buf.WriteByte(':')
10670         valBytes, err := json.Marshal(entry.Value)
10671         if err != nil {
10672             return nil, err
10673         }
10674         buf.Write(valBytes)
10675     }
10676     buf.WriteByte('}')
10677     return []byte(buf.String()), nil
10678 }
10679
10680 func jsonEncode(value Value) (any, error) {
10681     if _, ok := value.(*Null); ok {
10682         return nil, nil
10683     }
10684
10685     if x, ok := value.(*Boolean); ok {
10686         if x.data {
10687             return true, nil
10688         }
10689         return false, nil
10690     }
10691
10692     if x, ok := value.(*Number); ok {
10693         if math.IsNaN(x.data) || math.IsInf(x.data, 0) {
10694             return nil, fmt.Errorf("cannot JSON-encode value %v", value)
10695         }
10696         return x.data, nil
10697     }
10698
10699     if x, ok := value.(*String); ok {
10700         if !utf8.Valid([]byte(x.data)) {
10701             return nil, fmt.Errorf("cannot JSON-encode string with invalid UTF-8 encoding
10702 ↪ %v", value)
10703         }
10704         return x.data, nil
10705     }
10706
10707     if x, ok := value.(*Vector); ok {
10708         result := []any{}
10709         for _, element := range x.Elements() {
10710             j, err := jsonEncode(element)
10711             if err != nil {

```

```

10712         }
10713         result = append(result, j)
10714     }
10715     return result, nil
10716 }
10717
10718 if x, ok := value.(*Map); ok {
10719     result := orderedMapMarshaler{}
10720     for _, pair := range x.Pairs() {
10721         _, ok := pair.Key.(*String)
10722         if !ok {
10723             return nil, fmt.Errorf("cannot JSON-encode map with key %v", pair.Key)
10724         }
10725
10726         k, err := jsonEncode(pair.Key)
10727         if err != nil {
10728             return nil, err
10729         }
10730         v, err := jsonEncode(pair.Value)
10731         if err != nil {
10732             return nil, err
10733         }
10734
10735         ks, ok := k.(string)
10736         if !ok {
10737             return nil, fmt.Errorf("unexpected string JSON-encoding %v", ks)
10738         }
10739
10740         result = append(result, orderedMapEntry{ks, v})
10741     }
10742     return result, nil
10743 }
10744
10745 return nil, fmt.Errorf("cannot JSON-encode value %v of type %s", value,
↪ value.TypeName())
10746 }
10747
10748 func BuiltinJsonEncode(ctx *Context) Value {
10749     return ctx.NewBuiltin("json::encode", []Type{TVal(ANY)}, func(ctx *Context, arguments
↪ []Value) (Value, error) {
10750         j, err := jsonEncode(arguments[0])
10751         if err != nil {
10752             return nil, NewError(nil, ctx.NewString(err.Error()))
10753         }
10754         encoded, err := json.Marshal(j)
10755         if err != nil {
10756             return nil, NewError(nil, ctx.NewString(err.Error()))
10757         }
10758         return ctx.NewString(string(encoded)), nil
10759     })
10760 }
10761
10762 func BuiltinMathIsNaN(ctx *Context) Value {
10763     return ctx.NewBuiltin("math::is_nan", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
10764         return ctx.NewBoolean(math.IsNaN(arguments[0].(*Number).data)), nil
10765     })
10766 }
10767
10768 func BuiltinMathIsInf(ctx *Context) Value {
10769     return ctx.NewBuiltin("math::is_inf", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
10770         return ctx.NewBoolean(math.IsInf(arguments[0].(*Number).data, 0)), nil
10771     })
10772 }
10773
10774 func BuiltinMathIsFinite(ctx *Context) Value {

```

```

10775     return ctx.NewBuiltin("math::is_finite", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
10776         value := arguments[0].(*Number)
10777
10778         return ctx.NewBoolean(!math.IsInf(value.data, 0) && !math.IsNaN(value.data)), nil
10779     })
10780 }
10781
10782 func BuiltinMathIsInteger(ctx *Context) Value {
10783     return ctx.NewBuiltin("math::is_integer", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
10784         value := arguments[0].(*Number)
10785
10786         if math.IsInf(value.data, 0) || math.IsNaN(value.data) {
10787             return ctx.NewBoolean(false), nil
10788         }
10789
10790         return ctx.NewBoolean(math.Trunc(value.data) == value.data), nil
10791     })
10792 }
10793
10794 func BuiltinMathSign(ctx *Context) Value {
10795     return ctx.NewBuiltin("math::sign", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
10796         value := arguments[0].(*Number)
10797
10798         if value.data > 0 {
10799             return ctx.NewNumber(+1), nil
10800         }
10801         if value.data < 0 {
10802             return ctx.NewNumber(-1), nil
10803         }
10804         if value.data == 0 {
10805             return ctx.NewNumber(math.Copysign(0, value.data)), nil
10806         }
10807
10808         return ctx.NewNumber(math.NaN()), nil
10809     })
10810 }
10811
10812 func BuiltinMathCopySign(ctx *Context) Value {
10813     return ctx.NewBuiltin("math::copy_sign", []Type{TVal(NUMBER), TVal(NUMBER)}, func(ctx
↪ *Context, arguments []Value) (Value, error) {
10814         value := arguments[0].(*Number)
10815         sign := arguments[1].(*Number)
10816
10817         return ctx.NewNumber(math.Copysign(value.data, sign.data)), nil
10818     })
10819 }
10820
10821 func BuiltinMathTrunc(ctx *Context) Value {
10822     return ctx.NewBuiltin("math::trunc", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
10823         return ctx.NewNumber(math.Trunc(arguments[0].(*Number).data)), nil
10824     })
10825 }
10826
10827 func BuiltinMathRound(ctx *Context) Value {
10828     return ctx.NewBuiltin("math::round", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
10829         return ctx.NewNumber(math.Round(arguments[0].(*Number).data)), nil
10830     })
10831 }
10832
10833 func BuiltinMathFloor(ctx *Context) Value {
10834     return ctx.NewBuiltin("math::floor", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {

```

```

10835         return ctx.NewNumber(math.Floor(arguments[0].(*Number).data)), nil
10836     })
10837 }
10838
10839 func BuiltinMathCeil(ctx *Context) Value {
10840     return ctx.NewBuiltin("math::ceil", []Type{TVal(NUMBER)}, func(ctx *Context,
10841 ↪ arguments []Value) (Value, error) {
10842         return ctx.NewNumber(math.Ceil(arguments[0].(*Number).data)), nil
10843     })
10844 }
10845
10846 func BuiltinMathAbs(ctx *Context) Value {
10847     return ctx.NewBuiltin("math::abs", []Type{TVal(NUMBER)}, func(ctx *Context, arguments
10848 ↪ []Value) (Value, error) {
10849         return ctx.NewNumber(math.Abs(arguments[0].(*Number).data)), nil
10850     })
10851 }
10852
10853 func BuiltinMathMod(ctx *Context) Value {
10854     function := ctx.NewValueFromSourceOrPanic("math::mod", `
10855     return function(n, divisor) {
10856         if n.is_finite() and divisor.is_inf() {
10857             if n == 0 {
10858                 return math::copy_sign(0, divisor);
10859             }
10860             if math::sign(n) == math::sign(divisor) {
10861                 return n;
10862             }
10863             return divisor;
10864         }
10865         return ((n % divisor) + divisor) % divisor;
10866     };
10867 `)
10868     return ctx.NewBuiltin("math::mod", []Type{TVal(NUMBER), TVal(NUMBER)}, func(ctx
10869 ↪ *Context, arguments []Value) (Value, error) {
10870         return CallBuiltinFromSource(ctx, function, arguments)
10871     })
10872 }
10873
10874 func BuiltinMathRem(ctx *Context) Value {
10875     function := ctx.NewValueFromSourceOrPanic("math::rem", `
10876     return function(n, divisor) {
10877         return n % divisor;
10878     };
10879 `)
10880     return ctx.NewBuiltin("math::rem", []Type{TVal(NUMBER), TVal(NUMBER)}, func(ctx
10881 ↪ *Context, arguments []Value) (Value, error) {
10882         return CallBuiltinFromSource(ctx, function, arguments)
10883     })
10884 }
10885
10886 func BuiltinMathExp(ctx *Context) Value {
10887     return ctx.NewBuiltin("math::exp", []Type{TVal(NUMBER)}, func(ctx *Context, arguments
10888 ↪ []Value) (Value, error) {
10889         return ctx.NewNumber(math.Exp(arguments[0].(*Number).data)), nil
10890     })
10891 }
10892
10893 func BuiltinMathExp2(ctx *Context) Value {
10894     return ctx.NewBuiltin("math::exp2", []Type{TVal(NUMBER)}, func(ctx *Context,
10895 ↪ arguments []Value) (Value, error) {
10896         return ctx.NewNumber(math.Exp2(arguments[0].(*Number).data)), nil
10897     })
10898 }
10899
10900 }
10901
10902 }
10903
10904 }
10905

```

```

10896 func BuiltinMathExp10(ctx *Context) Value {
10897     return ctx.NewBuiltin("math::exp10", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
10898         return ctx.NewNumber(math.Pow(10, arguments[0].(*Number).data)), nil
10899     })
10900 }
10901
10902 func BuiltinMathLog(ctx *Context) Value {
10903     return ctx.NewBuiltin("math::log", []Type{TVal(NUMBER)}, func(ctx *Context, arguments
↪ []Value) (Value, error) {
10904         return ctx.NewNumber(math.Log(arguments[0].(*Number).data)), nil
10905     })
10906 }
10907
10908 func BuiltinMathLog2(ctx *Context) Value {
10909     return ctx.NewBuiltin("math::log2", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
10910         return ctx.NewNumber(math.Log2(arguments[0].(*Number).data)), nil
10911     })
10912 }
10913
10914 func BuiltinMathLog10(ctx *Context) Value {
10915     return ctx.NewBuiltin("math::log10", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
10916         return ctx.NewNumber(math.Log10(arguments[0].(*Number).data)), nil
10917     })
10918 }
10919
10920 func BuiltinMathPow(ctx *Context) Value {
10921     return ctx.NewBuiltin("math::pow", []Type{TVal(NUMBER), TVal(NUMBER)}, func(ctx
↪ *Context, arguments []Value) (Value, error) {
10922         return ctx.NewNumber(math.Pow(arguments[0].(*Number).data,
↪ arguments[1].(*Number).data)), nil
10923     })
10924 }
10925
10926 func BuiltinMathSqrt(ctx *Context) Value {
10927     return ctx.NewBuiltin("math::sqrt", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
10928         return ctx.NewNumber(math.Sqrt(arguments[0].(*Number).data)), nil
10929     })
10930 }
10931
10932 func BuiltinMathCbrt(ctx *Context) Value {
10933     return ctx.NewBuiltin("math::cbrt", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
10934         return ctx.NewNumber(math.Cbrt(arguments[0].(*Number).data)), nil
10935     })
10936 }
10937
10938 func BuiltinMathClamp(ctx *Context) Value {
10939     function := ctx.NewValueFromSourceOrPanic("math::clamp", `
10940 let clamp = function(value, min, max) {
10941     if min.is_nan() {
10942         error $"min is {min}";
10943     }
10944     if max.is_nan() {
10945         error $"max is {max}";
10946     }
10947     if min > max {
10948         error $"min > max ({min} > {max})";
10949     }
10950     if value.is_nan() {
10951         return NaN;
10952     }
10953     if value < min {
10954         return min;

```

```

10955     }
10956     if value > max {
10957         return max;
10958     }
10959     return value;
10960 };
10961 return clamp;
10962 `)
10963
10964     return ctx.NewBuiltin("math::clamp", []Type{TVal(NUMBER), TVal(NUMBER), TVal(NUMBER)},
↪ func(ctx *Context, arguments []Value) (Value, error) {
10965         return CallBuiltinFromSource(ctx, function, arguments)
10966     })
10967 }
10968
10969 func BuiltinMathSin(ctx *Context) Value {
10970     return ctx.NewBuiltin("math::sin", []Type{TVal(NUMBER)}, func(ctx *Context, arguments
↪ []Value) (Value, error) {
10971         return ctx.NewNumber(math.Sin(arguments[0].(*Number).data)), nil
10972     })
10973 }
10974
10975 func BuiltinMathCos(ctx *Context) Value {
10976     return ctx.NewBuiltin("math::cos", []Type{TVal(NUMBER)}, func(ctx *Context, arguments
↪ []Value) (Value, error) {
10977         return ctx.NewNumber(math.Cos(arguments[0].(*Number).data)), nil
10978     })
10979 }
10980
10981 func BuiltinMathTan(ctx *Context) Value {
10982     return ctx.NewBuiltin("math::tan", []Type{TVal(NUMBER)}, func(ctx *Context, arguments
↪ []Value) (Value, error) {
10983         return ctx.NewNumber(math.Tan(arguments[0].(*Number).data)), nil
10984     })
10985 }
10986
10987 func BuiltinMathAsin(ctx *Context) Value {
10988     return ctx.NewBuiltin("math::asin", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
10989         return ctx.NewNumber(math.Asin(arguments[0].(*Number).data)), nil
10990     })
10991 }
10992
10993 func BuiltinMathAcos(ctx *Context) Value {
10994     return ctx.NewBuiltin("math::acos", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
10995         return ctx.NewNumber(math.Acos(arguments[0].(*Number).data)), nil
10996     })
10997 }
10998
10999 func BuiltinMathAtan(ctx *Context) Value {
11000     return ctx.NewBuiltin("math::atan", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
11001         return ctx.NewNumber(math.Atan(arguments[0].(*Number).data)), nil
11002     })
11003 }
11004
11005 func BuiltinMathAtan2(ctx *Context) Value {
11006     return ctx.NewBuiltin("math::atan2", []Type{TVal(NUMBER), TVal(NUMBER)}, func(ctx
↪ *Context, arguments []Value) (Value, error) {
11007         return ctx.NewNumber(math.Atan2(arguments[0].(*Number).data,
↪ arguments[1].(*Number).data)), nil
11008     })
11009 }
11010
11011 func BuiltinMathSinh(ctx *Context) Value {

```

```

11012     return ctx.NewBuiltin("math::sinh", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
11013         return ctx.NewNumber(math.Sinh(arguments[0].(*Number).data)), nil
11014     })
11015 }
11016
11017 func BuiltinMathCosh(ctx *Context) Value {
11018     return ctx.NewBuiltin("math::cosh", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
11019         return ctx.NewNumber(math.Cosh(arguments[0].(*Number).data)), nil
11020     })
11021 }
11022
11023 func BuiltinMathTanh(ctx *Context) Value {
11024     return ctx.NewBuiltin("math::tanh", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
11025         return ctx.NewNumber(math.Tanh(arguments[0].(*Number).data)), nil
11026     })
11027 }
11028
11029 func BuiltinMathAsinh(ctx *Context) Value {
11030     return ctx.NewBuiltin("math::asinh", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
11031         return ctx.NewNumber(math.Asinh(arguments[0].(*Number).data)), nil
11032     })
11033 }
11034
11035 func BuiltinMathAcosh(ctx *Context) Value {
11036     return ctx.NewBuiltin("math::acosh", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
11037         return ctx.NewNumber(math.Acosh(arguments[0].(*Number).data)), nil
11038     })
11039 }
11040
11041 func BuiltinMathAtanh(ctx *Context) Value {
11042     return ctx.NewBuiltin("math::atanh", []Type{TVal(NUMBER)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
11043         return ctx.NewNumber(math.Atanh(arguments[0].(*Number).data)), nil
11044     })
11045 }
11046
11047 func BuiltinMathSum(ctx *Context) Value {
11048     function := ctx.NewValueFromSourceOrPanic("math::sum", `
11049 let add = function(lhs, rhs) {
11050     return lhs + rhs;
11051 };
11052 return function(iterable) {
11053     return iterable.reduce(add);
11054 };
11055 `)
11056
11057     return ctx.NewBuiltin("math::sum", []Type{TVal(ANY)}, func(ctx *Context, arguments
↪ []Value) (Value, error) {
11058         return CallBuiltinFromSource(ctx, function, arguments)
11059     })
11060 }
11061
11062 func BuiltinMathProduct(ctx *Context) Value {
11063     function := ctx.NewValueFromSourceOrPanic("math::product", `
11064 let multiply = function(lhs, rhs) {
11065     return lhs * rhs;
11066 };
11067 return function(iterable) {
11068     return iterable.reduce(multiply);
11069 };
11070 `)
11071

```

```

11072     return ctx.NewBuiltin("math::product", []Type{TVal(ANY)}, func(ctx *Context,
↪ arguments []Value) (Value, error) {
11073         return CallBuiltinFromSource(ctx, function, arguments)
11074     })
11075 }
11076
11077 func BuiltinRandomSeed(ctx *Context) Value {
11078     return ctx.NewBuiltin("random::seed", []Type{TVal(ANY)}, func(ctx *Context, arguments
↪ []Value) (Value, error) {
11079         seed := arguments[0].Hash()
11080         ctx.rng = rand.New(rand.NewPCG(seed, seed))
11081         return ctx.NewNull(), nil
11082     })
11083 }
11084
11085 func BuiltinRandomInteger(ctx *Context) Value {
11086     return ctx.NewBuiltin("random::integer", []Type{TVal(NUMBER), TVal(NUMBER)}, func(ctx
↪ *Context, arguments []Value) (Value, error) {
11087         min := arguments[0].(*Number)
11088         if math.IsInf(min.data, 0) || math.IsNaN(min.data) || math.Trunc(min.data) !=
↪ min.data {
11089             return nil, NewError(nil, ctx.NewStringf("expected integer lower bound,
↪ received %v", min))
11090         }
11091
11092         max := arguments[1].(*Number)
11093         if math.IsInf(max.data, 0) || math.IsNaN(max.data) || math.Trunc(max.data) !=
↪ max.data {
11094             return nil, NewError(nil, ctx.NewStringf("expected integer upper bound,
↪ received %v", max))
11095         }
11096
11097         if min.data > max.data {
11098             return nil, NewError(nil, ctx.NewStringf("min > max (%v > %v)", min, max))
11099         }
11100         if min.data < float64(MIN_SAFE_INTEGER) || min.data > float64(MAX_SAFE_INTEGER) {
11101             return nil, NewError(nil, ctx.NewStringf("integer %v is outside the safe
↪ integer range", min))
11102         }
11103         if max.data < float64(MIN_SAFE_INTEGER) || max.data > float64(MAX_SAFE_INTEGER) {
11104             return nil, NewError(nil, ctx.NewStringf("integer %v is outside the safe
↪ integer range", max))
11105         }
11106         minInteger := int64(min.data)
11107         maxInteger := int64(max.data)
11108
11109         return ctx.NewNumber(float64(ctx.rng.Int64N(maxInteger-minInteger+1) +
↪ minInteger)), nil
11110     })
11111 }
11112
11113 func BuiltinReSplit(ctx *Context) Value {
11114     function := ctx.NewValueFromSourceOrPanic("re::split", `
11115     return function(string, regexp) {
11116         return regexp.split(string);
11117     };
11118     `)
11119
11120     return ctx.NewBuiltin("re::split", []Type{TVal(STRING), TVal(REGEXP)}, func(ctx
↪ *Context, arguments []Value) (Value, error) {
11121         return CallBuiltinFromSource(ctx, function, arguments)
11122     })
11123 }
11124
11125 func BuiltinReReplace(ctx *Context) Value {
11126     function := ctx.NewValueFromSourceOrPanic("re::replace", `
11127     return function(string, regexp, replacement) {

```

```

11128     return regexp.replace(string, replacement);
11129 };
11130 `)
11131
11132     return ctx.NewBuiltin("re::replace", []Type{TVal(STRING), TVal(REGEXP), TVal(STRING)},
11133 ↪ func(ctx *Context, arguments []Value) (Value, error) {
11134         return CallBuiltinFromSource(ctx, function, arguments)
11135     })
11136 }
11137
11138 func BuiltinTyIs(ctx *Context) Value {
11139     return ctx.NewBuiltin("ty::is", []Type{TVal(ANY), TVal(ANY)}, func(ctx *Context,
11140 ↪ arguments []Value) (Value, error) {
11141         value := arguments[0]
11142         ty := arguments[1]
11143
11144         if _, ok := ty.(*Null); ok {
11145             return ctx.NewBoolean(value.Meta(ctx) == nil), nil
11146         }
11147
11148         if tyMap, ok := ty.(*Map); ok && tyMap.IsImmutable() && tyMap.name != nil {
11149             return ctx.NewBoolean(value.Meta(ctx) == tyMap), nil
11150         }
11151
11152         return nil, NewError(nil, ctx.NewStringf("expected null or map value created with
11153 ↪ the %s keyword, received %v", quote(TOKEN_TYPE), ty))
11154     })
11155 }
11156
11157 func BuiltinTyIsNull(ctx *Context) Value {
11158     return ctx.NewBuiltin("ty::is_null", []Type{TVal(ANY)}, func(ctx *Context, arguments
11159 ↪ []Value) (Value, error) {
11160         _, ok := arguments[0].(*Null)
11161         return ctx.NewBoolean(ok), nil
11162     })
11163 }
11164
11165 func BuiltinTyIsBoolean(ctx *Context) Value {
11166     return ctx.NewBuiltin("ty::is_boolean", []Type{TVal(ANY)}, func(ctx *Context,
11167 ↪ arguments []Value) (Value, error) {
11168         _, ok := arguments[0].(*Boolean)
11169         return ctx.NewBoolean(ok), nil
11170     })
11171 }
11172
11173 func BuiltinTyIsNumber(ctx *Context) Value {
11174     return ctx.NewBuiltin("ty::is_number", []Type{TVal(ANY)}, func(ctx *Context,
11175 ↪ arguments []Value) (Value, error) {
11176         _, ok := arguments[0].(*Number)
11177         return ctx.NewBoolean(ok), nil
11178     })
11179 }
11180
11181 func BuiltinTyIsString(ctx *Context) Value {
11182     return ctx.NewBuiltin("ty::is_string", []Type{TVal(ANY)}, func(ctx *Context,
11183 ↪ arguments []Value) (Value, error) {
11184         _, ok := arguments[0].(*String)
11185         return ctx.NewBoolean(ok), nil
11186     })
11187 }
11188
11189 func BuiltinTyIsRegex(ctx *Context) Value {
11190     return ctx.NewBuiltin("ty::is_regex", []Type{TVal(ANY)}, func(ctx *Context,
11191 ↪ arguments []Value) (Value, error) {
11192         _, ok := arguments[0].(*Regex)
11193         return ctx.NewBoolean(ok), nil
11194     })
11195 }

```

```

11187 }
11188
11189 func BuiltinTyIsVector(ctx *Context) Value {
11190     return ctx.NewBuiltin("ty::is_vector", []Type{TVal(ANY)}, func(ctx *Context,
11191 ↪ arguments []Value) (Value, error) {
11192         _, ok := arguments[0].(*Vector)
11193         return ctx.NewBoolean(ok), nil
11194     })
11195 }
11196
11197 func BuiltinTyIsMap(ctx *Context) Value {
11198     return ctx.NewBuiltin("ty::is_map", []Type{TVal(ANY)}, func(ctx *Context, arguments
11199 ↪ []Value) (Value, error) {
11200         _, ok := arguments[0].(*Map)
11201         return ctx.NewBoolean(ok), nil
11202     })
11203 }
11204
11205 func BuiltinTyIsSet(ctx *Context) Value {
11206     return ctx.NewBuiltin("ty::is_set", []Type{TVal(ANY)}, func(ctx *Context, arguments
11207 ↪ []Value) (Value, error) {
11208         _, ok := arguments[0].(*Set)
11209         return ctx.NewBoolean(ok), nil
11210     })
11211 }
11212
11213 func BuiltinTyIsReference(ctx *Context) Value {
11214     return ctx.NewBuiltin("ty::is_reference", []Type{TVal(ANY)}, func(ctx *Context,
11215 ↪ arguments []Value) (Value, error) {
11216         _, ok := arguments[0].(*Reference)
11217         return ctx.NewBoolean(ok), nil
11218     })
11219 }
11220
11221 func BuiltinTyIsFunction(ctx *Context) Value {
11222     return ctx.NewBuiltin("ty::is_function", []Type{TVal(ANY)}, func(ctx *Context,
11223 ↪ arguments []Value) (Value, error) {
11224         if _, ok := arguments[0].(*Function); ok {
11225             return ctx.NewBoolean(true), nil
11226         }
11227         if _, ok := arguments[0].(*Builtin); ok {
11228             return ctx.NewBoolean(true), nil
11229         }
11230         return ctx.NewBoolean(false), nil
11231     })
11232 }
11233
11234 func BuiltinUTF8IsValid(ctx *Context) Value {
11235     return ctx.NewBuiltin("utf8::is_valid", []Type{TVal(STRING)}, func(ctx *Context,
11236 ↪ arguments []Value) (Value, error) {
11237         return ctx.NewBoolean(utf8.ValidString(arguments[0].(*String).data)), nil
11238     })
11239 }
11240
11241 func BuiltinUTF8CodePoint(ctx *Context) Value {
11242     return ctx.NewBuiltin("utf8::code_point", []Type{TVal(STRING)}, func(ctx *Context,
11243 ↪ arguments []Value) (Value, error) {
11244         s := arguments[0].(*String)
11245         if !utf8.ValidString(s.data) {
11246             return nil, NewError(nil, ctx.NewStringf("expected valid UTF-8 string,
11247 ↪ received %v", s))
11248         }
11249         r := []rune(s.data)
11250         if len(r) != 1 {
11251             return nil, NewError(nil, ctx.NewStringf("expected UTF-8 string with 1 code
11252 ↪ point, received %v with %v code points", s, len(r)))

```

```

11245         }
11246
11247         return ctx.NewNumber(float64(r[0])), nil
11248     })
11249 }
11250
11251 func BuiltinUTF8CodePoints(ctx *Context) Value {
11252     return ctx.NewBuiltin("utf8::code_points", []Type{TVal(STRING)}, func(ctx *Context,
11253 ↪ arguments []Value) (Value, error) {
11254         s := arguments[0].(*String)
11255         if !utf8.ValidString(s.data) {
11256             return nil, NewError(nil, ctx.NewStringf("expected valid UTF-8 string,
11257 ↪ received %v", s))
11258         }
11259
11260         elements := []Value{}
11261         for _, n := range []rune(s.data) {
11262             elements = append(elements, ctx.NewNumber(float64(n)))
11263         }
11264
11265         return ctx.NewVectorOrPanic(elements), nil
11266     })
11267 }
11268
11269 func BuiltinUTF8String(ctx *Context) Value {
11270     return ctx.NewBuiltin("utf8::string", []Type{TVal(ANY)}, func(ctx *Context, arguments
11271 ↪ []Value) (Value, error) {
11272         value := arguments[0]
11273
11274         if valueNumber, ok := value.(*Number); ok {
11275             n, err := ValueAsSafeInteger(valueNumber)
11276             if err != nil {
11277                 return nil, NewError(nil, ctx.NewString(err.Error()))
11278             }
11279             if !(0 <= n && n <= utf8.MaxRune) {
11280                 return nil, NewError(nil, ctx.NewStringf("expected Unicode code point,
11281 ↪ received out-of-range integer %#x", n))
11282             }
11283             result := ctx.NewString(string(rune(n)))
11284             if !utf8.ValidString(result.data) {
11285                 return nil, NewError(nil, ctx.NewStringf("expected valid UTF-8 string,
11286 ↪ received %v", result))
11287             }
11288             return result, nil
11289         }
11290
11291         if valueVector, ok := value.(*Vector); ok {
11292             runes := []rune{}
11293             for _, element := range valueVector.Elements() {
11294                 n, err := ValueAsSafeInteger(element)
11295                 if err != nil {
11296                     return nil, NewError(nil, ctx.NewString(err.Error()))
11297                 }
11298                 if !(0 <= n && n <= utf8.MaxRune) {
11299                     return nil, NewError(nil, ctx.NewStringf("expected Unicode code
11300 ↪ point, received out-of-range integer %#x", n))
11301                 }
11302                 runes = append(runes, rune(n))
11303             }
11304             result := ctx.NewString(string(runes))
11305             if !utf8.ValidString(result.data) {
11306                 return nil, NewError(nil, ctx.NewStringf("expected valid UTF-8 string,
11307 ↪ received %v", result))
11308             }
11309             return result, nil
11310         }
11311     }
11312 }

```

mellifera.go

```
11305         return nil, NewError(nil, ctx.NewStringf("expected Unicode code point or vector
↪ of Unicode code points, received %v", value))
11306     })
11307 }
```

cmd/mf/mf.go

```
1 package main
2
3 import (
4     "bytes"
5     "errors"
6     "fmt"
7     "io"
8     "os"
9     "os/exec"
10    "path/filepath"
11    "regexp"
12    "strings"
13    "syscall"
14
15    "ashn.dev/mellifera"
16 )
17
18 func BuiltinExit(ctx *mellifera.Context) mellifera.Value {
19     return ctx.NewBuiltin("exit", []mellifera.Type{
20         mellifera.TVal(mellifera.NUMBER),
21     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
22     ↪ {
23         integer, err := mellifera.ValueAsSafeInteger(arguments[0])
24         if err != nil {
25             return nil, mellifera.NewError(nil, ctx.NewStringf("expected integer exit
26 ↪ code, received %v", arguments[0]))
27         }
28         os.Exit(int(integer))
29         return ctx.NewNull(), nil
30     })
31 }
32
33 func BuiltinImport(ctx *mellifera.Context) mellifera.Value {
34     return ctx.NewBuiltin("import", []mellifera.Type{
35         mellifera.TVal(mellifera.STRING),
36     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
37     ↪ {
38         target := arguments[0].(*mellifera.String)
39
40         module, err := ctx.BaseEnvironment.Get("module")
41         if err != nil {
42             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
43         }
44         moduleMap, ok := module.(*mellifera.Map)
45         if !ok {
46             return nil, mellifera.NewError(nil, ctx.NewStringf("expected map module
47 ↪ value, received %v", module.TypeName()))
48         }
49         _, modulePathOk := moduleMap.Lookup(ctx.NewString("path"))
50         _, moduleFileOk := moduleMap.Lookup(ctx.NewString("file"))
51         moduleDirectory, moduleDirectoryOk :=
52 ↪ moduleMap.Lookup(ctx.NewString("directory"))
53         if !modulePathOk || !moduleFileOk || !moduleDirectoryOk {
54             return nil, mellifera.NewError(nil, ctx.NewStringf("expected module map to
55 ↪ contain `path`, `file` and `directory` values, received %v", module))
56         }
57         moduleDirectoryString, ok := moduleDirectory.(*mellifera.String)
58         if !ok {
```

```

53         return nil, mellifera.NewError(nil, ctx.NewStringf("expected string module
↪ directory value, received %v", moduleDirectory))
54     }
55
56     // Always restore module fields.
57     defer (func() {
58         _ = ctx.BaseEnvironment.Set("module", moduleMap)
59     })()
60
61     var result mellifera.Value = nil
62     paths := []string{}
63     if filepath.IsAbs(target.Data()) {
64         paths = append(paths, target.Data()) // direct path
65     } else {
66         paths = append(paths, filepath.Join(moduleDirectoryString.Data(),
↪ target.Data())) // current module directory
67         if search, ok := os.LookupEnv("MELLIFERA_SEARCH_PATH"); ok {
68             for _, p := range strings.Split(search, ":") {
69                 paths = append(paths, filepath.Join(p, target.Data()))
70             }
71         }
72     }
73     for _, p := range paths {
74         stat, err := os.Stat(p)
75         if err != nil {
76             continue
77         }
78         if stat.IsDir() {
79             // If the path is a directory, such as in the case of a
80             // library, load the entry point to the library and/or group of
81             // files, using the name <directory>/lib.mf by convention.
82             p = filepath.Join(p, "lib.mf")
83         }
84         absolute, err := filepath.Abs(p)
85         if err != nil {
86             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
87         }
88
89         if cached, ok := ctx.GetModule(absolute); ok {
90             return cached, nil
91         }
92
93         importModuleMap := ctx.NewMapOrPanic([]mellifera.MapPair{
94             {ctx.NewString("path"), ctx.NewString(absolute)},
95             {ctx.NewString("file"), ctx.NewString(filepath.Base(absolute))},
96             {ctx.NewString("directory"), ctx.NewString(filepath.Dir(absolute))},
97         }).Freeze()
98         if err := ctx.BaseEnvironment.Set("module", importModuleMap); err != nil {
99             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
100         }
101
102         bytes, err := os.ReadFile(absolute)
103         if err != nil {
104             continue
105         }
106         source := string(bytes)
107
108         lexer := mellifera.NewLexer(ctx, source, &mellifera.SourceLocation{p, 1})
109         parser, err := mellifera.NewParser(&lexer)
110         if err != nil {
111             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
112         }
113         program, err := parser.ParseProgram()
114         if err != nil {
115             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
116         }

```

```

117         result, err = program.Eval(ctx,
↪ mellifera.NewEnvironment(ctx.BaseEnvironment))
118         if err != nil {
119             if e, ok := err.(mellifera.Error); ok {
120                 return nil, e
121             }
122             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
123         }
124         ctx.SetModule(absolute, result) // cache import
125         break
126     }
127
128     if result == nil {
129         return nil, mellifera.NewError(nil, ctx.NewStringf("module %v not found",
↪ target))
130     }
131     return result, nil
132 })
133 }
134
135 func BuiltinInput(ctx *mellifera.Context) mellifera.Value {
136     return ctx.NewBuiltin("input", []mellifera.Type{}, func(ctx *mellifera.Context,
↪ arguments []mellifera.Value) (mellifera.Value, error) {
137         data, err := io.ReadAll(ctx.Stdin)
138         if err != nil {
139             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
140         }
141         return ctx.NewString(string(data)), nil
142     })
143 }
144
145 func BuiltinInputln(ctx *mellifera.Context) mellifera.Value {
146     return ctx.NewBuiltin("inputln", []mellifera.Type{}, func(ctx *mellifera.Context,
↪ arguments []mellifera.Value) (mellifera.Value, error) {
147         data := []byte{}
148         buf := make([]byte, 1)
149         for {
150             nbytes, err := ctx.Stdin.Read(buf)
151             if nbytes != 0 {
152                 if buf[0] == '\n' {
153                     break
154                 }
155                 data = append(data, buf[0])
156             }
157             if err != nil {
158                 if err == io.EOF {
159                     if len(data) == 0 {
160                         // The end of input was reached before any line data
161                         // was read. Produce null so that the end of input is
162                         // distinguishable from an empty line.
163                         return ctx.NewNull(), nil
164                     }
165                     break
166                 }
167                 return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
168             }
169         }
170         return ctx.NewString(string(data)), nil
171     })
172 }
173
174 func BuiltinEnvGet(ctx *mellifera.Context) mellifera.Value {
175     return ctx.NewBuiltin("env::get", []mellifera.Type{
176         mellifera.TVal(mellifera.STRING),
177     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
178         variable := arguments[0].(*mellifera.String)

```

```

179         lookup, ok := os.LookupEnv(variable.Data())
180         if !ok {
181             return ctx.NewNull(), nil
182         }
183         return ctx.NewString(lookup), nil
184     })
185 }
186 }
187
188 func BuiltinEnvSet(ctx *mellifera.Context) mellifera.Value {
189     return ctx.NewBuiltin("env::set", []mellifera.Type{
190         mellifera.TVal(mellifera.STRING),
191         mellifera.TVal(mellifera.STRING),
192     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
193     ↪ {
194         variable := arguments[0].(*mellifera.String)
195         value := arguments[1].(*mellifera.String)
196
197         err := os.Setenv(variable.Data(), value.Data())
198         if err != nil {
199             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
200         }
201         return ctx.NewNull(), nil
202     })
203 }
204
205 func BuiltinEnvUnset(ctx *mellifera.Context) mellifera.Value {
206     return ctx.NewBuiltin("env::unset", []mellifera.Type{
207         mellifera.TVal(mellifera.STRING),
208     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
209     ↪ {
210         variable := arguments[0].(*mellifera.String)
211
212         err := os.Unsetenv(variable.Data())
213         if err != nil {
214             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
215         }
216         return ctx.NewNull(), nil
217     })
218 }
219
220 func BuiltinFsRead(ctx *mellifera.Context) mellifera.Value {
221     return ctx.NewBuiltin("fs::read", []mellifera.Type{
222         mellifera.TVal(mellifera.STRING),
223     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
224     ↪ {
225         path := arguments[0].(*mellifera.String)
226
227         data, err := os.ReadFile(path.Data())
228         if err != nil {
229             var pathErr *os.PathError
230             if errors.As(err, &pathErr) {
231                 if errors.Is(pathErr.Err, os.ErrNotExist) {
232                     return nil, mellifera.NewError(nil, ctx.NewStringf("failed to read
233 ↪ file %v (file not found)", path))
234                 }
235             }
236             return nil, mellifera.NewError(nil, ctx.NewStringf("failed to read file %v
237 ↪ (%s)", path, err.Error()))
238         }
239         return ctx.NewString(string(data)), nil
240     })
241 }
242
243 func BuiltinFsWrite(ctx *mellifera.Context) mellifera.Value {
244     return ctx.NewBuiltin("fs::write", []mellifera.Type{
245         mellifera.TVal(mellifera.STRING),

```

```

241     mellifera.TVal(mellifera.STRING),
242 }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
243     path := arguments[0].(*mellifera.String)
244     data := arguments[1].(*mellifera.String)
245
246     f, err := os.OpenFile(path.Data(), os.O_WRONLY|os.O_TRUNC|os.O_CREATE, 0644)
247     if err != nil {
248         return nil, mellifera.NewError(nil, ctx.NewStringf("failed write to file %v
↪ (%s)", path, err.Error()))
249     }
250     defer f.Close()
251     if _, err := f.Write([]byte(data.Data())); err != nil {
252         return nil, mellifera.NewError(nil, ctx.NewStringf("failed write to file %v
↪ (%s)", path, err.Error()))
253     }
254     return ctx.NewNull(), nil
255 })
256 }
257
258 func BuiltinFsAppend(ctx *mellifera.Context) mellifera.Value {
259     return ctx.NewBuiltin("fs::append", []mellifera.Type{
260         mellifera.TVal(mellifera.STRING),
261         mellifera.TVal(mellifera.STRING),
262     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
263         path := arguments[0].(*mellifera.String)
264         data := arguments[1].(*mellifera.String)
265
266         f, err := os.OpenFile(path.Data(), os.O_WRONLY|os.O_APPEND|os.O_CREATE, 0644)
267         if err != nil {
268             return nil, mellifera.NewError(nil, ctx.NewStringf("failed append to file %v
↪ (%s)", path, err.Error()))
269         }
270         defer f.Close()
271         if _, err := f.Write([]byte(data.Data())); err != nil {
272             return nil, mellifera.NewError(nil, ctx.NewStringf("failed append to file %v
↪ (%s)", path, err.Error()))
273         }
274         return ctx.NewNull(), nil
275     })
276 }
277
278 func shRun(ctx *mellifera.Context, command string, stdin *string) (*mellifera.Map, error,
↪ int) {
279     found := strings.Index(command, "\x00")
280     if found != -1 {
281         return nil, mellifera.NewError(nil, ctx.NewStringf("invalid null byte in POSIX
↪ shell string")), -1
282     }
283
284     var stdout bytes.Buffer
285     var stderr bytes.Buffer
286     cmd := exec.Command("/bin/sh", "-c", command)
287     if stdin != nil {
288         cmd.Stdin = strings.NewReader(*stdin)
289     }
290     cmd.Stdout = &stdout
291     cmd.Stderr = &stderr
292
293     status := 0
294     if err := cmd.Run(); err != nil {
295         // > https://pkg.go.dev/os/exec@go1.26.7#Cmd.Run
296         // > =====
297         // > Run starts the specified command and waits for it to complete.
298         // >
299         // > The returned error is nil if the command runs, has no problems

```

```

300 // > copying stdin, stdout, and stderr, and exits with a zero exit
301 // > status.
302 // >
303 // > If the command starts but does not complete successfully, the
304 // > error is of type *ExitError. Other error types may be returned
305 // > for other situations.
306 if exitError, ok := err.(*exec.ExitError); ok {
307     waitStatus := exitError.Sys().(syscall.WaitStatus)
308     if waitStatus.Signal() {
309         // Use the POSIX shell convention of 128 + signum for $?
310         status = 128 + int(waitStatus.Signal())
311     } else {
312         status = waitStatus.ExitStatus()
313     }
314 } else {
315     return nil, mellifera.NewError(nil, ctx.NewString(err.Error())), -1
316 }
317 }
318
319 return ctx.NewMapOrPanic([]mellifera.MapPair{
320     {ctx.NewString("status"), ctx.NewNumber(float64(status))},
321     {ctx.NewString("stdout"), ctx.NewString(stdout.String())},
322     {ctx.NewString("stderr"), ctx.NewString(stderr.String())},
323 }, nil, status
324 }
325
326 func BuiltinShRun(ctx *mellifera.Context) mellifera.Value {
327     return ctx.NewBuiltin("sh::run", []mellifera.Type{
328         mellifera.TVal(mellifera.STRING),
329     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
330     ↪ {
331         command := arguments[0].(*mellifera.String)
332
333         result, err, _ := shRun(ctx, command.Data(), nil)
334         return result, err
335     })
336 }
337
338 func BuiltinShRunEx(ctx *mellifera.Context) mellifera.Value {
339     return ctx.NewBuiltin("sh::run_ex", []mellifera.Type{
340         mellifera.TVal(mellifera.STRING),
341         mellifera.TVal(mellifera.MAP),
342     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
343     ↪ {
344         command := arguments[0].(*mellifera.String)
345         options := arguments[1].(*mellifera.Map)
346
347         var stdin *string = nil // optional
348         for _, pair := range options.Pairs() {
349             if k, ok := pair.Key.(*mellifera.String); ok && k.Data() == "stdin" {
350                 if v, ok := pair.Value.(*mellifera.String); ok {
351                     data := v.Data()
352                     stdin = &data
353                     continue
354                 }
355             }
356
357             return nil, mellifera.NewError(nil, ctx.NewStringf("expected string
358             ↪ standard input, received %v", pair.Value))
359         }
360
361         return nil, mellifera.NewError(nil, ctx.NewStringf("unknown option %v",
362             ↪ pair.Key))
363     }
364
365     result, err, _ := shRun(ctx, command.Data(), stdin)
366     return result, err
367 })
368 }

```

```

363 }
364
365 func BuiltinShRunx(ctx *mellifera.Context) mellifera.Value {
366     return ctx.NewBuiltin("sh::runx", []mellifera.Type{
367         mellifera.TVal(mellifera.STRING),
368     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
369     ↪ {
370         command := arguments[0].(*mellifera.String)
371
372         result, err, status := shRun(ctx, command.Data(), nil)
373         if err != nil {
374             return nil, err
375         }
376         if status != 0 {
377             return nil, mellifera.NewError(nil, result)
378         }
379         return result, err
380     })
381 }
382
383 func BuiltinShRunEx(ctx *mellifera.Context) mellifera.Value {
384     return ctx.NewBuiltin("sh::runx_ex", []mellifera.Type{
385         mellifera.TVal(mellifera.STRING),
386         mellifera.TVal(mellifera.MAP),
387     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
388     ↪ {
389         command := arguments[0].(*mellifera.String)
390         options := arguments[1].(*mellifera.Map)
391
392         var stdin *string = nil // optional
393         for _, pair := range options.Pairs() {
394             if k, ok := pair.Key.(*mellifera.String); ok && k.Data() == "stdin" {
395                 if v, ok := pair.Value.(*mellifera.String); ok {
396                     data := v.Data()
397                     stdin = &data
398                     continue
399                 }
400             }
401
402             return nil, mellifera.NewError(nil, ctx.NewStringf("expected string
403             ↪ standard input, received %v", pair.Value))
404         }
405
406         return nil, mellifera.NewError(nil, ctx.NewStringf("unknown option %v",
407         ↪ pair.Key))
408     })
409 }
410
411 result, err, status := shRun(ctx, command.Data(), stdin)
412 if err != nil {
413     return nil, err
414 }
415 if status != 0 {
416     return nil, mellifera.NewError(nil, result)
417 }
418 return result, err
419 })
420 }
421
422 func BuiltinShQuote(ctx *mellifera.Context) mellifera.Value {
423     return ctx.NewBuiltin("sh::quote", []mellifera.Type{
424         mellifera.TVal(mellifera.STRING),
425     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
426     ↪ {
427         s := arguments[0].(*mellifera.String)
428
429         found := strings.Index(s.Data(), "\x00")
430         if found != -1 {

```

```

424         return nil, mellifera.NewError(nil, ctx.NewStringf("invalid null byte in
↳ POSIX shell string"))
425     }
426
427     // 'foo bar'
428     // 'foo\'\'bar'
429     return ctx.NewStringf("%s'", strings.ReplaceAll(s.Data(), "'", "'\\'")), nil
430 })
431 }
432
433 func dumpTokensSource(ctx *mellifera.Context, source string, location
↳ *mellifera.SourceLocation) error {
434     tokens := ctx.NewVectorOrPanic(nil)
435     lexer := mellifera.NewLexer(ctx, source, location)
436
437     token, err := lexer.NextToken()
438     if err != nil {
439         return err
440     }
441     for token.Kind != mellifera.TOKEN_EOF {
442         err = tokens.Push(token.IntoValue(ctx))
443         if err != nil {
444             return err
445         }
446         token, err = lexer.NextToken()
447         if err != nil {
448             return err
449         }
450     }
451
452     var sb strings.Builder
453     indent := "    "
454     encoder := mellifera.NewCombEncoder(&sb)
455     encoder.Indent = &indent
456     encoder.Separator = " "
457     err = tokens.CombEncode(encoder)
458     if err != nil {
459         return err
460     }
461     fmt.Println(sb.String())
462     return nil
463 }
464
465 func dumpTokensFile(ctx *mellifera.Context, path string) error {
466     bytes, err := os.ReadFile(path)
467     if err != nil {
468         return err
469     }
470     return dumpTokensSource(ctx, string(bytes), &mellifera.SourceLocation{path, 1})
471 }
472
473 func dumpAstSource(ctx *mellifera.Context, source string, location
↳ *mellifera.SourceLocation) error {
474     lexer := mellifera.NewLexer(ctx, source, location)
475     parser, err := mellifera.NewParser(&lexer)
476     if err != nil {
477         return err
478     }
479     program, err := parser.ParseProgram()
480     if err != nil {
481         return err
482     }
483
484     var sb strings.Builder
485     indent := "    "
486     encoder := mellifera.NewCombEncoder(&sb)
487     encoder.Indent = &indent

```

```

488     encoder.Separator = " "
489     err = program.IntoValue(ctx).CombEncode(encoder)
490     if err != nil {
491         return err
492     }
493     fmt.Println(sb.String())
494     return nil
495 }
496
497 func dumpAstFile(ctx *mellifera.Context, path string) error {
498     bytes, err := os.ReadFile(path)
499     if err != nil {
500         return err
501     }
502     return dumpAstSource(ctx, string(bytes), &mellifera.SourceLocation{path, 1})
503 }
504
505 func evalSource(ctx *mellifera.Context, source string, location
506 ↪ *mellifera.SourceLocation) (mellifera.Value, error) {
507     lexer := mellifera.NewLexer(ctx, source, location)
508     parser, err := mellifera.NewParser(&lexer)
509     if err != nil {
510         return nil, err
511     }
512     program, err := parser.ParseProgram()
513     if err != nil {
514         return nil, err
515     }
516     env := mellifera.NewEnvironment(ctx.BaseEnvironment)
517     return program.Eval(ctx, env)
518 }
519
520 func evalFile(ctx *mellifera.Context, path string) (mellifera.Value, error) {
521     bytes, err := os.ReadFile(path)
522     if err != nil {
523         return nil, err
524     }
525     return evalSource(ctx, string(bytes), &mellifera.SourceLocation{path, 1})
526 }
527
528 func mfenv(w io.Writer) {
529     fmt.Fprintf(w, "MELLIFERA_HOME=%s\n", os.Getenv("MELLIFERA_HOME"))
530     fmt.Fprintf(w, "MELLIFERA_SEARCH_PATH=%s\n", os.Getenv("MELLIFERA_SEARCH_PATH"))
531 }
532
533 func usage(w io.Writer) {
534     program := os.Args[0]
535     fmt.Fprintf(w, `usage:
536 %s FILE [ARGS...]
537 %s [-c|--command] COMMAND [ARGS...]
538
539 options:
540 -c, --command      Execute the provided command.
541 --dump-tokens      Dump a comb-encoded vector of lexed tokens to stdout.
542 --dump-ast         Dump a comb-encoded abstract syntax tree to stdout.
543 -e, --env          Display the Mellifera environment and exit.
544 -h, --help         Display this help text and exit.
545 `, program, program)
546 }
547
548 func main() {
549     reCommand := regexp.MustCompile(`^+c(?:ommand)?(?:=(. *))?$`)
550     reDumpTokens := regexp.MustCompile(`^+dump-tokens$`)
551     reDumpAst := regexp.MustCompile(`^+dump-ast$`)
552     reEnv := regexp.MustCompile(`^+e(?:nv)?$`)
553     reHelp := regexp.MustCompile(`^+h(?:elp)?$`)

```

```

554     envMELLIFERA_HOME, ok := os.LookupEnv("MELLIFERA_HOME")
555     if !ok {
556         // $MELLIFERA_HOME/bin/mf
557         exe, err := os.Executable()
558         if err != nil {
559             fmt.Fprintf(os.Stderr, "error: %s\n", err.Error())
560             os.Exit(1)
561         }
562         // $MELLIFERA_HOME/bin
563         bin := filepath.Dir(exe)
564         // $MELLIFERA_HOME
565         envMELLIFERA_HOME = filepath.Dir(bin)
566         if err = os.Setenv("MELLIFERA_HOME", envMELLIFERA_HOME); err != nil {
567             fmt.Fprintf(os.Stderr, "error: %s\n", err.Error())
568             os.Exit(1)
569         }
570     }
571     envMELLIFERA_SEARCH_PATH, ok := os.LookupEnv("MELLIFERA_SEARCH_PATH")
572     if !ok {
573         envMELLIFERA_SEARCH_PATH = fmt.Sprintf("%s/lib", envMELLIFERA_HOME)
574         if err := os.Setenv("MELLIFERA_SEARCH_PATH", envMELLIFERA_SEARCH_PATH); err !=
575         ↪ nil {
576             fmt.Fprintf(os.Stderr, "error: %s\n", err.Error())
577             os.Exit(1)
578         }
579     }
580     verbatim := false
581     var cmds *string
582     var file *string
583     var argv []string
584     dumpTokens := false
585     dumpAst := false
586     argi := 1
587     for argi < len(os.Args) {
588         arg := os.Args[argi]
589
590         positional := func() {
591             if cmds == nil && file == nil {
592                 file = &arg
593                 argv = append(argv, arg)
594                 verbatim = true
595                 argi += 1
596                 return
597             }
598
599             argv = append(argv, arg)
600             argi += 1
601         }
602
603         if verbatim {
604             positional()
605             continue
606         }
607
608         // Remaining args are processed verbatim.
609         if arg == "--" {
610             verbatim = true
611             argi += 1
612             continue
613         }
614
615         // -c, -command
616         if m := reCommand.FindStringSubmatch(arg); m != nil {
617             // -c='println("hello 🐼");'
618             if m[1] != "" {
619                 cmds = &m[1]

```

```

620         argv = append([]string{os.Args[0]}, os.Args[argi+1:]...)
621         break
622     }
623
624     // -c 'println("hello 🐝");'
625     if argi+1 < len(os.Args) {
626         cmds = &os.Args[argi+1]
627         argv = append([]string{os.Args[0]}, os.Args[argi+2:]...)
628         break
629     }
630
631     fmt.Fprintf(os.Stderr, "error: expected command argument\n")
632     usage(os.Stderr)
633     os.Exit(1)
634 }
635
636 // -dump-tokens
637 if m := reDumpTokens.FindStringSubmatch(arg); m != nil {
638     dumpTokens = true
639     argi += 1
640     continue
641 }
642
643 // -dump-ast
644 if m := reDumpAst.FindStringSubmatch(arg); m != nil {
645     dumpAst = true
646     argi += 1
647     continue
648 }
649
650 // -e, -env
651 if m := reEnv.FindStringSubmatch(arg); m != nil {
652     mfenv(os.Stdout)
653     os.Exit(0)
654 }
655
656 // -h, -help
657 if m := reHelp.FindStringSubmatch(arg); m != nil {
658     usage(os.Stdout)
659     os.Exit(0)
660 }
661
662 if strings.HasPrefix(arg, "-") {
663     fmt.Fprintf(os.Stderr, "error: unknown flag %s\n", arg)
664     usage(os.Stderr)
665     os.Exit(1)
666 }
667
668 positional()
669 }
670
671 var err error
672 ctx := mellifera.NewContext()
673 ctx.BaseEnvironment.Let("exit", BuiltinExit(ctx))
674 ctx.BaseEnvironment.Let("import", BuiltinImport(ctx))
675 ctx.BaseEnvironment.Let("input", BuiltinInput(ctx))
676 ctx.BaseEnvironment.Let("inputln", BuiltinInputln(ctx))
677 ctx.BaseEnvironment.Let("env", ctx.NewMapOrPanic([]mellifera.MapPair{
678     {ctx.NewString("get"), BuiltinEnvGet(ctx)},
679     {ctx.NewString("set"), BuiltinEnvSet(ctx)},
680     {ctx.NewString("unset"), BuiltinEnvUnset(ctx)},
681 }).Freeze())
682 ctx.BaseEnvironment.Let("fs", ctx.NewMapOrPanic([]mellifera.MapPair{
683     {ctx.NewString("read"), BuiltinFsRead(ctx)},
684     {ctx.NewString("write"), BuiltinFsWrite(ctx)},
685     {ctx.NewString("append"), BuiltinFsAppend(ctx)},
686 }).Freeze())

```

```

687     ctx.BaseEnvironment.Let("sh", ctx.NewMapOrPanic([]mellifera.MapPair{
688         {ctx.NewString("run"), BuiltinShRun(ctx)},
689         {ctx.NewString("run_ex"), BuiltinShRunEx(ctx)},
690         {ctx.NewString("runx"), BuiltinShRunx(ctx)},
691         {ctx.NewString("runx_ex"), BuiltinShRunxEx(ctx)},
692         {ctx.NewString("quote"), BuiltinShQuote(ctx)},
693     }).Freeze())
694
695     argvIntoValue := func() mellifera.Value {
696         result := ctx.NewVectorOrPanic(nil)
697         for _, arg := range argv {
698             _ = result.Push(ctx.NewString(arg))
699         }
700         return result.Freeze()
701     }
702     ctx.BaseEnvironment.Let("argv", argvIntoValue())
703
704     var path string
705     if file != nil {
706         path, err = filepath.Abs(*file)
707         if err != nil {
708             fmt.Fprintf(os.Stderr, "error: %s\n", err.Error())
709             os.Exit(1)
710         }
711     } else {
712         path, err = filepath.Abs(os.Args[0])
713         if err != nil {
714             fmt.Fprintf(os.Stderr, "error: %s\n", err.Error())
715             os.Exit(1)
716         }
717     }
718     module := ctx.NewMapOrPanic([]mellifera.MapPair{
719         {ctx.NewString("path"), ctx.NewString(path)},
720         {ctx.NewString("file"), ctx.NewString(filepath.Base(path))},
721         {ctx.NewString("directory"), ctx.NewString(filepath.Dir(path))},
722     }).Freeze()
723     if err = ctx.BaseEnvironment.Set("module", module); err != nil {
724         fmt.Fprintf(os.Stderr, "error: %s\n", err.Error())
725         os.Exit(1)
726     }
727
728     if dumpTokens && dumpAst {
729         ↪ fmt.Fprintf(os.Stderr, "error: requested token dump and AST dump which are
730         mutually exclusive\n")
731         os.Exit(1)
732     } else if cmds != nil || file != nil {
733         if cmds != nil && dumpTokens {
734             err = dumpTokensSource(ctx, *cmds, &mellifera.SourceLocation{"<command>", 1})
735         } else if cmds != nil && dumpAst {
736             err = dumpAstSource(ctx, *cmds, &mellifera.SourceLocation{"<command>", 1})
737         } else if cmds != nil {
738             _, err = evalSource(ctx, *cmds, &mellifera.SourceLocation{"<command>", 1})
739         } else if file != nil && dumpTokens {
740             err = dumpTokensFile(ctx, *file)
741         } else if file != nil && dumpAst {
742             err = dumpAstFile(ctx, *file)
743         } else if file != nil {
744             _, err = evalFile(ctx, *file)
745         } else {
746             err = fmt.Errorf("unreachable")
747         }
748     } else if dumpTokens {
749         ↪ fmt.Fprintf(os.Stderr, "error: requested token dump without a command or file
750         path\n")
751         os.Exit(1)
752     } else if dumpAst {

```

```

751         fmt.Fprintf(os.Stderr, "error: requested AST dump without a command or file
↪ path\n")
752         os.Exit(1)
753     } else {
754         usage(os.Stderr)
755         os.Exit(1)
756     }
757
758     if err != nil {
759         if e, ok := err.(mellifera.ParseError); ok && e.Location != nil {
760             fmt.Fprintf(os.Stderr, "[%v, line %v] error: %v\n", e.Location.File,
↪ e.Location.Line, err)
761         } else if e, ok := err.(mellifera.Error); ok {
762             if e.Location != nil {
763                 fmt.Fprintf(os.Stderr, "[%v, line %v] error: %v\n", e.Location.File,
↪ e.Location.Line, err)
764             } else {
765                 fmt.Fprintf(os.Stderr, "error: %v\n", err)
766             }
767             for _, element := range e.Trace {
768                 s := fmt.Sprintf("...within %v", element.FuncName)
769                 if element.Location != nil {
770                     s += fmt.Sprintf(" called from %s, line %v", element.Location.File,
↪ element.Location.Line)
771                 }
772                 fmt.Fprintf(os.Stderr, "%s\n", s)
773             }
774         } else {
775             fmt.Fprintf(os.Stderr, "%v\n", err)
776         }
777         os.Exit(1)
778     }
779 }

```


cmd/wasm/wasm.go

```
1 package main
2
3 import (
4     "fmt"
5     "io"
6     "os"
7     "path"
8     "runtime/debug"
9     "strings"
10    "syscall/js"
11
12    "ashn.dev/mellifera"
13 )
14
15 var isDebug = false
16
17 func debugf(format string, args ...any) {
18     if isDebug {
19         fmt.Fprintf(os.Stderr, format, args...)
20     }
21 }
22
23 type contextCacheEntry struct {
24     ctx *mellifera.Context
25     env *mellifera.Environment
26 }
27
28 var contextCache = make(map[string]contextCacheEntry)
29
30 type textarea struct {
31     id string
32 }
33
34 func (self textarea) clear() {
35     element := js.Global().Get("document").Call("getElementById", self.id)
36     element.Set("value", "")
37 }
38
39 func (self textarea) Write(p []byte) (n int, err error) {
40     element := js.Global().Get("document").Call("getElementById", self.id)
41     value := element.Get("value").String()
42     element.Set("value", value+string(p))
43     element.Set("scrollTop", element.Get("scrollHeight")) // focus on bottom
44     return len(p), nil
45 }
46
47 func JsValueFromMelliferaValue(ctx *mellifera.Context, mfValue mellifera.Value) (js.Value,
48     ↵ error) {
49     switch v := mfValue.(type) {
50     case *mellifera.Null:
51         return js.Null(), nil
52     case *mellifera.Boolean:
53         return js.ValueOf(v.Data()), nil
54     case *mellifera.Number:
55         return js.ValueOf(v.Data()), nil
56     case *mellifera.String:
57         return js.ValueOf(v.Data()), nil
58     case *mellifera.Vector:
59         result := js.Global().Get("Array").New(v.Count())
```

```

59         for i, element := range v.Elements() {
60             elementJsValue, err := JsValueFromMelliferaValue(ctx, element)
61             if err != nil {
62                 return js.Null(), mellifera.NewError(nil, ctx.NewStringf("unable to
↪ represent Mellifera vector element %v at index %v in JavaScript", element, i))
63             }
64             result.SetIndex(i, elementJsValue)
65         }
66         return result, nil
67     case *mellifera.Map:
68         result := js.Global().Get("Object").New()
69         for _, pair := range v.Pairs() {
70             keyString, ok := pair.Key.(*mellifera.String)
71             if !ok {
72                 return js.Null(), mellifera.NewError(nil, ctx.NewStringf("unable to
↪ represent Mellifera map key %v as JavaScript string key", pair.Key))
73             }
74             valueJsValue, err := JsValueFromMelliferaValue(ctx, pair.Value)
75             if err != nil {
76                 return js.Null(), mellifera.NewError(nil, ctx.NewStringf("unable to
↪ represent Mellifera map value %v in JavaScript", pair.Value))
77             }
78             result.Set(keyString.Data(), valueJsValue)
79         }
80         return result, nil
81     default:
82         return js.Null(), mellifera.NewError(nil, ctx.NewStringf("unable to represent
↪ Mellifera value %v in JavaScript", mfValue))
83     }
84 }
85
86 func JsValueIntoMelliferaValue(ctx *mellifera.Context, jsValue js.Value) (mellifera.Value,
↪ error) {
87     switch jsValue.Type() {
88     case js.TypeNull:
89         return ctx.NewNull(), nil
90     case js.TypeBoolean:
91         return ctx.NewBoolean(jsValue.Bool()), nil
92     case js.TypeNumber:
93         return ctx.NewNumber(jsValue.Float()), nil
94     case js.TypeString:
95         return ctx.NewString(jsValue.String()), nil
96     case js.TypeObject:
97         isArray := js.Global().Get("Array").Call("isArray", jsValue).Bool()
98         if isArray {
99             elements := []mellifera.Value{}
100             for i := range jsValue.Length() {
101                 element, err := JsValueIntoMelliferaValue(ctx, jsValue.Index(i))
102                 if err != nil {
103                     return nil, mellifera.NewError(nil, ctx.NewStringf("unable to
↪ represent JavaScript array element %v at index %v in Mellifera", jsValue.Index(i),
↪ i))
104                 }
105                 elements = append(elements, element)
106             }
107             return ctx.NewVectorOrPanic(elements), nil
108         }
109         pairs := []mellifera.MapPair{}
110         entries := js.Global().Get("Object").Call("entries", jsValue)
111         for i := range entries.Length() {
112             keyJsValue := entries.Index(i).Index(0)
113             keyMfValue, err := JsValueIntoMelliferaValue(ctx, keyJsValue)
114             if err != nil {
115                 return nil, mellifera.NewError(nil, ctx.NewStringf("unable to represent
↪ JavaScript object key %v in Mellifera", keyJsValue))
116             }
117             valueJsValue := entries.Index(i).Index(1)

```

```

118         valueMfValue, err := JsValueIntoMelliferaValue(ctx, valueJsValue)
119         if err != nil {
120             return nil, mellifera.NewError(nil, ctx.NewStringf("unable to represent
↪ JavaScript object value %v in Mellifera", valueJsValue))
121         }
122         pairs = append(pairs, mellifera.MapPair{keyMfValue, valueMfValue})
123     }
124     return ctx.NewMap(pairs)
125 default:
126     return nil, mellifera.NewError(nil, ctx.NewStringf("unable to represent
↪ JavaScript value %v in Mellifera", jsValue))
127 }
128 }
129
130 func BuiltinJsFromMellifera(ctx *mellifera.Context) *mellifera.Builtin {
131     return ctx.NewBuiltin("js::from_mellifera", []mellifera.Type{
132         mellifera.TVal(mellifera.ANY),
133     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
134         jsValue, err := JsValueFromMelliferaValue(ctx, arguments[0])
135         if err != nil {
136             return nil, err
137         }
138         return
↪ ctx.NewExternalWithType(ctx.BaseEnvironment.GetOrPanic("@js::value").(*mellifera.Map),
↪ jsValue), nil
139     })
140 }
141
142 func BuiltinJsIntoMellifera(ctx *mellifera.Context) *mellifera.Builtin {
143     return ctx.NewBuiltin("js::into_mellifera", []mellifera.Type{
144         mellifera.TVal(mellifera.EXTERNAL),
145     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
146         jsValue, ok := arguments[0].(*mellifera.External).Data().(js.Value)
147         if !ok {
148             return nil, mellifera.NewError(nil, ctx.NewStringf("invalid JavaScript value
↪ %v", arguments[0]))
149         }
150         return JsValueIntoMelliferaValue(ctx, jsValue)
151     })
152 }
153
154 func BuiltinJsIsUndefined(ctx *mellifera.Context) *mellifera.Builtin {
155     return ctx.NewBuiltin("js::is_undefined", []mellifera.Type{
156         mellifera.TVal(mellifera.EXTERNAL),
157     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
158         jsValue, ok := arguments[0].(*mellifera.External).Data().(js.Value)
159         if !ok {
160             return nil, mellifera.NewError(nil, ctx.NewStringf("invalid JavaScript value
↪ %v", arguments[0]))
161         }
162         return ctx.NewBoolean(jsValue.Type() == js.TypeUndefined), nil
163     })
164 }
165
166 func BuiltinJsIsNull(ctx *mellifera.Context) *mellifera.Builtin {
167     return ctx.NewBuiltin("js::is_null", []mellifera.Type{
168         mellifera.TVal(mellifera.EXTERNAL),
169     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
170         jsValue, ok := arguments[0].(*mellifera.External).Data().(js.Value)
171         if !ok {
172             return nil, mellifera.NewError(nil, ctx.NewStringf("invalid JavaScript value
↪ %v", arguments[0]))
173         }

```

```

174         return ctx.NewBoolean(jsValue.Type() == js.TypeNull), nil
175     })
176 }
177
178 func BuiltinJsIsBoolean(ctx *mellifera.Context) *mellifera.Builtin {
179     return ctx.NewBuiltin("js::is_boolean", []mellifera.Type{
180         mellifera.TVal(mellifera.EXTERNAL),
181     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
182     ↪ {
183         jsValue, ok := arguments[0].(*mellifera.External).Data().(js.Value)
184         if !ok {
185             return nil, mellifera.NewError(nil, ctx.NewStringf("invalid JavaScript value
186             ↪ %v", arguments[0]))
187         }
188         return ctx.NewBoolean(jsValue.Type() == js.TypeBoolean), nil
189     })
190 }
191
192 func BuiltinJsIsNumber(ctx *mellifera.Context) *mellifera.Builtin {
193     return ctx.NewBuiltin("js::is_number", []mellifera.Type{
194         mellifera.TVal(mellifera.EXTERNAL),
195     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
196     ↪ {
197         jsValue, ok := arguments[0].(*mellifera.External).Data().(js.Value)
198         if !ok {
199             return nil, mellifera.NewError(nil, ctx.NewStringf("invalid JavaScript value
200             ↪ %v", arguments[0]))
201         }
202         return ctx.NewBoolean(jsValue.Type() == js.TypeNumber), nil
203     })
204 }
205
206 func BuiltinJsIsString(ctx *mellifera.Context) *mellifera.Builtin {
207     return ctx.NewBuiltin("js::is_string", []mellifera.Type{
208         mellifera.TVal(mellifera.EXTERNAL),
209     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
210     ↪ {
211         jsValue, ok := arguments[0].(*mellifera.External).Data().(js.Value)
212         if !ok {
213             return nil, mellifera.NewError(nil, ctx.NewStringf("invalid JavaScript value
214             ↪ %v", arguments[0]))
215         }
216         return ctx.NewBoolean(jsValue.Type() == js.TypeString), nil
217     })
218 }
219
220 func BuiltinJsIsSymbol(ctx *mellifera.Context) *mellifera.Builtin {
221     return ctx.NewBuiltin("js::is_symbol", []mellifera.Type{
222         mellifera.TVal(mellifera.EXTERNAL),
223     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
224     ↪ {
225         jsValue, ok := arguments[0].(*mellifera.External).Data().(js.Value)
226         if !ok {
227             return nil, mellifera.NewError(nil, ctx.NewStringf("invalid JavaScript value
228             ↪ %v", arguments[0]))
229         }
230         return ctx.NewBoolean(jsValue.Type() == js.TypeSymbol), nil
231     })
232 }
233
234 func BuiltinJsIsArray(ctx *mellifera.Context) *mellifera.Builtin {
235     return ctx.NewBuiltin("js::is_array", []mellifera.Type{
236         mellifera.TVal(mellifera.EXTERNAL),
237     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
238     ↪ {
239         jsValue, ok := arguments[0].(*mellifera.External).Data().(js.Value)
240         if !ok {

```

```

232         return nil, mellifera.NewError(nil, ctx.NewStringf("invalid JavaScript value
↪ %v", arguments[0]))
233     }
234     return ctx.NewBoolean(js.Global().Get("Array").Call("isArray", jsValue).Bool()),
↪ nil
235 })
236 }
237
238 func BuiltinJsIsObject(ctx *mellifera.Context) *mellifera.Builtin {
239     return ctx.NewBuiltin("js::is_object", []mellifera.Type{
240         mellifera.TVal(mellifera.EXTERNAL),
241     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
242         jsValue, ok := arguments[0].(*mellifera.External).Data().(js.Value)
243         if !ok {
244             return nil, mellifera.NewError(nil, ctx.NewStringf("invalid JavaScript value
↪ %v", arguments[0]))
245         }
246         return ctx.NewBoolean(jsValue.Type() == js.TypeObject), nil
247     })
248 }
249
250 func BuiltinJsIsFunction(ctx *mellifera.Context) *mellifera.Builtin {
251     return ctx.NewBuiltin("js::is_function", []mellifera.Type{
252         mellifera.TVal(mellifera.EXTERNAL),
253     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
254         jsValue, ok := arguments[0].(*mellifera.External).Data().(js.Value)
255         if !ok {
256             return nil, mellifera.NewError(nil, ctx.NewStringf("invalid JavaScript value
↪ %v", arguments[0]))
257         }
258         return ctx.NewBoolean(jsValue.Type() == js.TypeFunction), nil
259     })
260 }
261
262 func BuiltinJsValueGet(ctx *mellifera.Context) *mellifera.Builtin {
263     return ctx.NewBuiltinWithSelfByReference("js::value::get", []mellifera.Type{
264         mellifera.TRef(mellifera.TVal(mellifera.EXTERNAL)),
265         mellifera.TVal(mellifera.STRING),
266     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
267         self := arguments[0].(*mellifera.Reference)
268         delf := self.Data().(*mellifera.External)
269         property := arguments[1].(*mellifera.String)
270
271         delfJsValue, ok := delf.Data().(js.Value)
272         if !ok {
273             return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
↪ a JavaScript value", delf))
274         }
275
276         if delfJsValue.Type() != js.TypeObject {
277             return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
↪ a JavaScript object", delf))
278         }
279
280         lookup := delfJsValue.Get(property.Data())
281         return
↪ ctx.NewExternalWithType(ctx.BaseEnvironment.GetOrPanic("@js::value").(*mellifera.Map),
↪ lookup), nil
282     })
283 }
284
285 func BuiltinJsValueSet(ctx *mellifera.Context) *mellifera.Builtin {
286     return ctx.NewBuiltinWithSelfByReference("js::value::set", []mellifera.Type{
287         mellifera.TRef(mellifera.TVal(mellifera.EXTERNAL)),

```

```

288     mellifera.TVal(mellifera.STRING),
289     mellifera.TVal(mellifera.EXTERNAL),
290 }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
291     self := arguments[0].(*mellifera.Reference)
292     delf := self.Data().(*mellifera.External)
293     property := arguments[1].(*mellifera.String)
294     value := arguments[2].(*mellifera.External)
295
296     delfJsValue, ok := delf.Data().(js.Value)
297     if !ok {
298         return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
↪ a JavaScript value", delf))
299     }
300
301     if delfJsValue.Type() != js.TypeObject {
302         return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
↪ a JavaScript object", delf))
303     }
304
305     valueJsValue, ok := value.Data().(js.Value)
306     if !ok {
307         return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
↪ a JavaScript value", value))
308     }
309
310     delfJsValue.Set(property.Data(), valueJsValue)
311     return ctx.NewNull(), nil
312 })
313 }
314
315 func BuiltinJsValueDelete(ctx *mellifera.Context) *mellifera.Builtin {
316     return ctx.NewBuiltinWithSelfByReference("js::value::delete", []mellifera.Type{
317         mellifera.TRef(mellifera.TVal(mellifera.EXTERNAL)),
318         mellifera.TVal(mellifera.STRING),
319     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
320         self := arguments[0].(*mellifera.Reference)
321         delf := self.Data().(*mellifera.External)
322         property := arguments[1].(*mellifera.String)
323
324         delfJsValue, ok := delf.Data().(js.Value)
325         if !ok {
326             return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
↪ a JavaScript value", delf))
327         }
328
329         if delfJsValue.Type() != js.TypeObject {
330             return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
↪ a JavaScript object", delf))
331         }
332
333         delfJsValue.Delete(property.Data())
334         return ctx.NewNull(), nil
335     })
336 }
337
338 func BuiltinJsValueGetIndex(ctx *mellifera.Context) *mellifera.Builtin {
339     return ctx.NewBuiltinWithSelfByReference("js::value::get_index", []mellifera.Type{
340         mellifera.TRef(mellifera.TVal(mellifera.EXTERNAL)),
341         mellifera.TVal(mellifera.NUMBER),
342     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
343         self := arguments[0].(*mellifera.Reference)
344         delf := self.Data().(*mellifera.External)
345         index := arguments[1].(*mellifera.Number)
346

```

```

347         delfJsValue, ok := delf.Data().(js.Value)
348         if !ok {
349             return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
↪ a JavaScript value", delf))
350         }
351
352         if delfJsValue.Type() != js.TypeObject {
353             return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
↪ a JavaScript object", delf))
354         }
355
356         indexInt, err := mellifera.ValueAsIndex(index)
357         if err != nil {
358             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
359         }
360
361         lookup := delfJsValue.Index(indexInt)
362         return
↪ ctx.NewExternalWithType(ctx.BaseEnvironment.GetOrPanic("@js::value").(*mellifera.Map),
↪ lookup), nil
363     })
364 }
365
366 func BuiltinJsValueSetIndex(ctx *mellifera.Context) *mellifera.Builtin {
367     return ctx.NewBuiltinWithSelfByReference("js::value::set_index", []mellifera.Type{
368         mellifera.TRef(mellifera.TVal(mellifera.EXTERNAL)),
369         mellifera.TVal(mellifera.NUMBER),
370         mellifera.TVal(mellifera.EXTERNAL),
371     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
372         self := arguments[0].(*mellifera.Reference)
373         delf := self.Data().(*mellifera.External)
374         index := arguments[1].(*mellifera.Number)
375         value := arguments[2].(*mellifera.External)
376
377         delfJsValue, ok := delf.Data().(js.Value)
378         if !ok {
379             return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
↪ a JavaScript value", delf))
380         }
381
382         valueJsValue, ok := value.Data().(js.Value)
383         if !ok {
384             return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
↪ a JavaScript value", value))
385         }
386
387         if delfJsValue.Type() != js.TypeObject {
388             return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
↪ a JavaScript object", delf))
389         }
390
391         indexInt, err := mellifera.ValueAsIndex(index)
392         if err != nil {
393             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
394         }
395
396         delfJsValue.SetIndex(indexInt, valueJsValue)
397         return ctx.NewNull(), nil
398     })
399 }
400
401 func BuiltinJsValueCall(ctx *mellifera.Context) *mellifera.Builtin {
402     return ctx.NewBuiltinWithSelfByReference("js::value::call", []mellifera.Type{
403         mellifera.TRef(mellifera.TVal(mellifera.EXTERNAL)),
404         mellifera.TVal(mellifera.STRING),
405         mellifera.TVal(mellifera.VECTOR),

```

```

406     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
407         self := arguments[0].(*mellifera.Reference)
408         delf := self.Data().(*mellifera.External)
409         method := arguments[1].(*mellifera.String)
410         args := arguments[2].(*mellifera.Vector)
411
412         delfJsValue, ok := delf.Data().(js.Value)
413         if !ok {
414             return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
↪ a JavaScript value", delf))
415         }
416
417         if delfJsValue.Type() != js.TypeObject {
418             return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
↪ a JavaScript object", delf))
419         }
420
421         methodJsValue := delfJsValue.Get(method.Data())
422         if methodJsValue.Type() != js.TypeFunction {
423             return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v does
↪ not have method %s (found %v)", delf, method.Data(),
↪ ctx.NewExternalWithType(ctx.BaseEnvironment.GetOrPanic("@js::value").(*mellifera.Map),
↪ methodJsValue)))
424         }
425
426         argsSlice := []any{}
427         for i, arg := range args.Elements() {
428             argExternal, ok := arg.(*mellifera.External)
429             if !ok {
430                 return nil, mellifera.NewError(nil, ctx.NewStringf("argument %v with
↪ value %v is not an external JavaScript value", i, arg))
431             }
432             argJsValue, ok := argExternal.Data().(js.Value)
433             if !ok {
434                 return nil, mellifera.NewError(nil, ctx.NewStringf("argument %v with
↪ value %v is not an external JavaScript value", i, arg))
435             }
436             argsSlice = append(argsSlice, argJsValue)
437         }
438
439         result := delfJsValue.Call(method.Data(), argsSlice...)
440         return
↪ ctx.NewExternalWithType(ctx.BaseEnvironment.GetOrPanic("@js::value").(*mellifera.Map),
↪ result), nil
441     })
442 }
443
444 func BuiltinJsCall(ctx *mellifera.Context) *mellifera.Builtin {
445     return ctx.NewBuiltin("js::call", []mellifera.Type{
446         mellifera.TVal(mellifera.EXTERNAL),
447         mellifera.TVal(mellifera.VECTOR),
448     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
449         function := arguments[0].(*mellifera.External)
450         args := arguments[1].(*mellifera.Vector)
451
452         functionJsValue, ok := function.Data().(js.Value)
453         if !ok {
454             return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
↪ a JavaScript value", function))
455         }
456
457         if functionJsValue.Type() != js.TypeFunction {
458             return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
↪ a JavaScript function", function))
459         }

```

```

460         argsSlice := []any{}
461         for i, arg := range args.Elements() {
462             argExternal, ok := arg.(*mellifera.External)
463             if !ok {
464                 return nil, mellifera.NewError(nil, ctx.NewStringf("argument %v with
465 ↪ value %v is not an external JavaScript value", i, arg))
466             }
467             argJsValue, ok := argExternal.Data().(js.Value)
468             if !ok {
469                 return nil, mellifera.NewError(nil, ctx.NewStringf("argument %v with
470 ↪ value %v is not an external JavaScript value", i, arg))
471             }
472             argsSlice = append(argsSlice, argJsValue)
473         }
474         result := functionJsValue.Invoke(argsSlice...)
475         return
476 ↪ ctx.NewExternalWithType(ctx.BaseEnvironment.GetOrPanic("@js::value").(*mellifera.Map),
477 ↪ result), nil
478     })
479 }
480
481 func BuiltinJsCallNew(ctx *mellifera.Context) *mellifera.Builtin {
482     return ctx.NewBuiltin("js::call_new", []mellifera.Type{
483         mellifera.TVal(mellifera.EXTERNAL),
484         mellifera.TVal(mellifera.VECTOR),
485     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
486 ↪ {
487         function := arguments[0].(*mellifera.External)
488         args := arguments[1].(*mellifera.Vector)
489
490         functionJsValue, ok := function.Data().(js.Value)
491         if !ok {
492             return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
493 ↪ a JavaScript value", function))
494         }
495
496         if functionJsValue.Type() != js.TypeFunction {
497             return nil, mellifera.NewError(nil, ctx.NewStringf("external value %v is not
498 ↪ a JavaScript function", functionJsValue))
499         }
500
501         argsSlice := []any{}
502         for i, arg := range args.Elements() {
503             argExternal, ok := arg.(*mellifera.External)
504             if !ok {
505                 return nil, mellifera.NewError(nil, ctx.NewStringf("argument %v with
506 ↪ value %v is not an external JavaScript value", i, arg))
507             }
508             argJsValue, ok := argExternal.Data().(js.Value)
509             if !ok {
510                 return nil, mellifera.NewError(nil, ctx.NewStringf("argument %v with
511 ↪ value %v is not an external JavaScript value", i, arg))
512             }
513             argsSlice = append(argsSlice, argJsValue)
514         }
515
516         result := functionJsValue.New(argsSlice...)
517         return
518 ↪ ctx.NewExternalWithType(ctx.BaseEnvironment.GetOrPanic("@js::value").(*mellifera.Map),
519 ↪ result), nil
520     })
521 }
522
523 func BuiltinJsGlobal(ctx *mellifera.Context) *mellifera.Builtin {

```

```

515     return ctx.NewBuiltin("js::global", []mellifera.Type{}, func(ctx *mellifera.Context,
↪ arguments []mellifera.Value) (mellifera.Value, error) {
516         return
↪ ctx.NewExternalWithType(ctx.BaseEnvironment.GetOrPanic("@js::value").(*mellifera.Map),
↪ js.Global()), nil
517     })
518 }
519
520 func BuiltinJsTypeof(ctx *mellifera.Context) *mellifera.Builtin {
521     return ctx.NewBuiltin("js::typeof", []mellifera.Type{
522         mellifera.TVal(mellifera.EXTERNAL),
523     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
524         jsValue, ok := arguments[0].(*mellifera.External).Data().(js.Value)
525         if !ok {
526             return nil, mellifera.NewError(nil, ctx.NewStringf("invalid JavaScript value
↪ %v", arguments[0]))
527         }
528
529         // XXX: At the time of writing (Go version 1.26.2) attempting to call
530         // js.Value.Type() on a BigInt instance will cause a runtime panic[1],
531         // so switching on jsValue.Type() below has the potential to blow up
532         // the current goroutine. It is basically impossible to work around
533         // this issue at the moment, as preemptively checking the instance's
534         // .constructor property with:
535         //
536         // constructor := jsValue.Get("constructor")
537         // if constructor.Equal(js.Global().Get("BigInt")) {
538         //     return ctx.NewString("bigint"), nil
539         // }
540         //
541         // will blow up if a non-object type is passed in, and attempting to
542         // manually check if jsValue instanceof BigInt with:
543         //
544         // jsValue.InstanceOf(js.Global().Get("BigInt"))
545         //
546         // will fail with the Go runtime erroneously returning false for this
547         // expression.
548         //
549         // [1]: https://github.com/golang/go/issues/72050#issuecomment-4346819262
550
551         //
↪ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof#description
552         switch jsValue.Type() {
553             case js.TypeUndefined:
554                 return ctx.NewString("undefined"), nil
555             case js.TypeNull:
556                 // https://developer.mozilla.org/en-US/docs/Glossary/Null
557                 // Due to historical reasons, typeof null is "object".
558                 return ctx.NewString("object"), nil
559             case js.TypeBoolean:
560                 return ctx.NewString("boolean"), nil
561             case js.TypeNumber:
562                 return ctx.NewString("number"), nil
563             case js.TypeString:
564                 return ctx.NewString("string"), nil
565             case js.TypeSymbol:
566                 return ctx.NewString("symbol"), nil
567             case js.TypeFunction:
568                 return ctx.NewString("function"), nil
569             case js.TypeObject:
570                 return ctx.NewString("object"), nil
571             default:
572                 panic("unknown JavaScript type")
573         }
574     })
575 }

```

```

576
577 func BuiltinFsRead(ctx *mellifera.Context) mellifera.Value {
578     return ctx.NewBuiltin("fs::read", []mellifera.Type{
579         mellifera.TVal(mellifera.STRING),
580     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪     {
581         file := arguments[0].(*mellifera.String)
582
583         fs, err := ctx.BaseEnvironment.Get("@fs")
584         if err != nil {
585             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
586         }
587         fsMap, ok := fs.(*mellifera.Map)
588         if !ok {
589             return nil, mellifera.NewError(nil, ctx.NewStringf("expected @fs map value,
↪ found %v", fs))
590         }
591
592         // Directly map string filenames to text content. This is a flat
593         // mapping that ignores the concept of directory structure.
594         text, ok := fsMap.Lookup(file)
595         if !ok {
596             return nil, mellifera.NewError(nil, ctx.NewStringf("failed to read file %v
↪ (file not found)", file))
597         }
598         _, ok = text.(*mellifera.String)
599         if !ok {
600             return nil, mellifera.NewError(nil, ctx.NewStringf("expected @fs file string
↪ value, found %v", text))
601         }
602
603         return text, nil
604     })
605 }
606
607 func BuiltinFsWrite(ctx *mellifera.Context) mellifera.Value {
608     return ctx.NewBuiltin("fs::write", []mellifera.Type{
609         mellifera.TVal(mellifera.STRING),
610         mellifera.TVal(mellifera.STRING),
611     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪     {
612         file := arguments[0].(*mellifera.String)
613         data := arguments[1].(*mellifera.String)
614
615         fs, err := ctx.BaseEnvironment.Get("@fs")
616         if err != nil {
617             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
618         }
619         fsMap, ok := fs.(*mellifera.Map)
620         if !ok {
621             return nil, mellifera.NewError(nil, ctx.NewStringf("expected @fs map value,
↪ found %v", fs))
622         }
623
624         // Directly map string filenames to text content. This is a flat
625         // mapping that ignores the concept of directory structure.
626         err = fsMap.Insert(file, data)
627         if err != nil {
628             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
629         }
630
631         return ctx.NewNull(), nil
632     })
633 }
634
635 func BuiltinFsAppend(ctx *mellifera.Context) mellifera.Value {
636     return ctx.NewBuiltin("fs::append", []mellifera.Type{

```

```

637         mellifera.TVal(mellifera.STRING),
638         mellifera.TVal(mellifera.STRING),
639     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
640         file := arguments[0].(*mellifera.String)
641         data := arguments[1].(*mellifera.String)
642
643         fs, err := ctx.BaseEnvironment.Get("@fs")
644         if err != nil {
645             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
646         }
647         fsMap, ok := fs.(*mellifera.Map)
648         if !ok {
649             return nil, mellifera.NewError(nil, ctx.NewStringf("expected @fs map value,
↪ found %v", fs))
650         }
651
652         // Directly map string filenames to text content. This is a flat
653         // mapping that ignores the concept of directory structure.
654         text, ok := fsMap.Lookup(file)
655         if !ok {
656             err = fsMap.Insert(file, data)
657             if err != nil {
658                 return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
659             }
660             return ctx.NewNull(), nil
661         }
662         textString, ok := text.(*mellifera.String)
663         if !ok {
664             return nil, mellifera.NewError(nil, ctx.NewStringf("expected @fs file string
↪ value, found %v", text))
665         }
666         err = fsMap.Insert(file, ctx.NewString(textString.Data()+data.Data()))
667         if err != nil {
668             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
669         }
670
671         return ctx.NewNull(), nil
672     })
673 }
674
675 func BuiltinImport(ctx *mellifera.Context) mellifera.Value {
676     return ctx.NewBuiltin("import", []mellifera.Type{
677         mellifera.TVal(mellifera.STRING),
678     }, func(ctx *mellifera.Context, arguments []mellifera.Value) (mellifera.Value, error)
↪ {
679         target := arguments[0].(*mellifera.String)
680
681         module, err := ctx.BaseEnvironment.Get("module")
682         if err != nil {
683             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
684         }
685         moduleMap, ok := module.(*mellifera.Map)
686         if !ok {
687             return nil, mellifera.NewError(nil, ctx.NewStringf("expected map module
↪ value, received %v", module.TypeName()))
688         }
689         _, modulePathOk := moduleMap.Lookup(ctx.NewString("path"))
690         _, moduleFileOk := moduleMap.Lookup(ctx.NewString("file"))
691         moduleDirectory, moduleDirectoryOk :=
↪ moduleMap.Lookup(ctx.NewString("directory"))
692         if !modulePathOk || !moduleFileOk || !moduleDirectoryOk {
693             return nil, mellifera.NewError(nil, ctx.NewStringf("expected module map to
↪ contain `path`, `file` and `directory` values, received %v", module))
694         }
695
696         // Always restore module fields.

```

```

697     defer (func() {
698         _ = ctx.BaseEnvironment.Set("module", moduleMap)
699     })()
700
701     fs, err := ctx.BaseEnvironment.Get("@fs")
702     if err != nil {
703         return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
704     }
705     fsMap, ok := fs.(*mellifera.Map)
706     if !ok {
707         return nil, mellifera.NewError(nil, ctx.NewStringf("expected @fs map value,
↪ found %v", fs))
708     }
709
710     var result mellifera.Value = nil
711     paths := []string{
712         target.Data(), // direct path
713     }
714     if moduleDirectoryString, ok := moduleDirectory.(*mellifera.String); ok {
715         paths = append(paths, moduleDirectoryString.Data()+"/"+target.Data()) // path
↪ within current module directory
716     }
717     for _, p := range paths {
718         // Strip the trailing slash off of directories.
719         p = strings.TrimSuffix(p, "/")
720         if _, ok := fsMap.Lookup(ctx.NewString(p + "/lib.mf")); ok {
721             // If the path is a directory, such as in the case of a
722             // library, load the entry point to the library and/or group of
723             // files, using the name <directory>/lib.mf by convention.
724             p = p + "/lib.mf"
725         }
726
727         if cached, ok := ctx.GetModule(p); ok {
728             return cached, nil
729         }
730
731         importModuleMap := ctx.NewMapOrPanic([]mellifera.MapPair{
732             {ctx.NewString("path"), ctx.NewString(p)},
733             {ctx.NewString("file"), ctx.NewString(path.Base(p))},
734             {ctx.NewString("directory"), ctx.NewString(path.Dir(p))},
735         }).Freeze()
736         if err := ctx.BaseEnvironment.Set("module", importModuleMap); err != nil {
737             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
738         }
739
740         // Directly map string filenames to text content. This is a flat
741         // mapping that ignores the concept of directory structure.
742         text, ok := fsMap.Lookup(ctx.NewString(p))
743         if !ok {
744             continue
745         }
746
747         textString, ok := text.(*mellifera.String)
748         if !ok {
749             return nil, mellifera.NewError(nil, ctx.NewStringf("expected @fs file
↪ string value, found %v", text))
750         }
751         source := textString.Data()
752
753         lexer := mellifera.NewLexer(ctx, source, &mellifera.SourceLocation{p, 1})
754         parser := mellifera.NewParser(&lexer)
755         if err != nil {
756             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
757         }
758         program, err := parser.ParseProgram()
759         if err != nil {
760             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))

```

```

761         }
762         result, err = program.Eval(ctx,
↪ mellifera.NewEnvironment(ctx.BaseEnvironment))
763         if err != nil {
764             if e, ok := err.(mellifera.Error); ok {
765                 return nil, e
766             }
767             return nil, mellifera.NewError(nil, ctx.NewString(err.Error()))
768         }
769         ctx.SetModule(p, result) // cache import
770         break
771     }
772
773     if result == nil {
774         return nil, mellifera.NewError(nil, ctx.NewStringf("module %v not found",
↪ target))
775     }
776     return result, nil
777 })
778 }
779
780 func main() {
781     mf := js.Global().Get("mellifera")
782     mf.Set("eval", js.FuncOf(func(this js.Value, args []js.Value) any {
783         if len(args) == 0 || len(args) > 2 {
784             fmt.Fprintf(os.Stderr, "error: expected one or two arguments for
↪ mellifera.eval(source, options), received %v arguments\n", len(args))
785             return nil
786         } else if len(args) == 1 {
787             args = append(args, js.ValueOf(map[string]any{})) // empty options
788         }
789
790         if args[0].Type() != js.TypeString {
791             fmt.Fprintf(os.Stderr, "error: expected source to be a string, received
↪ %v\n", args[0])
792             return nil
793         }
794         source := args[0].String()
795
796         if args[1].Type() != js.TypeObject {
797             fmt.Fprintf(os.Stderr, "error: expected options to be an object, received
↪ %v\n", args[1])
798             return nil
799         }
800         options := args[1]
801
802         newContext := func() (*mellifera.Context, error) {
803             ctx := mellifera.NewContext()
804
805             if !options.Get("stdout").IsUndefined() {
806                 if options.Get("stdout").Type() != js.TypeString {
807                     return nil, fmt.Errorf("error: expected options.stdout to be a
↪ string, received %v\n", options.Get("stdout"))
808                 }
809                 stdoutTextarea := textarea{options.Get("stdout").String()}
810                 stdoutTextarea.clear()
811                 ctx.Stdout = stdoutTextarea
812             }
813
814             if !options.Get("stderr").IsUndefined() {
815                 if options.Get("stderr").Type() != js.TypeString {
816                     return nil, fmt.Errorf("error: expected options.stderr to be a
↪ string, received %v\n", options.Get("stderr"))
817                 }
818                 stderrTextarea := textarea{options.Get("stderr").String()}
819                 stderrTextarea.clear()
820                 ctx.Stderr = stderrTextarea

```

```

821     }
822
823     fs := ctx.NewMapOrPanic(nil)
824     if !options.Get("fs").IsUndefined() {
825         value, err := JsValueIntoMelliferaValue(ctx, options.Get("fs"))
826         if err != nil {
827             return nil, err
828         }
829         valueMap, ok := value.(*mellifera.Map)
830         if !ok {
831             return nil, fmt.Errorf("expected options.fs to be a map, received
↪ %v\n", value)
832         }
833         fs = valueMap
834     }
835
836     ctx.BaseEnvironment.Let("@fs", fs)
837     ctx.BaseEnvironment.Let("fs", ctx.NewMapOrPanic([]mellifera.MapPair{
838         {ctx.NewString("read"), BuiltinFsRead(ctx)},
839         {ctx.NewString("write"), BuiltinFsWrite(ctx)},
840         {ctx.NewString("append"), BuiltinFsAppend(ctx)},
841     })).Freeze())
842     ctx.BaseEnvironment.Let("import", BuiltinImport(ctx))
843     ctx.BaseEnvironment.Let("@js::value", ctx.NewMetaMapOrPanic("js::value",
↪ []mellifera.MapPair{
844         {ctx.NewString("get"), BuiltinJsValueGet(ctx)},
845         {ctx.NewString("set"), BuiltinJsValueSet(ctx)},
846         {ctx.NewString("delete"), BuiltinJsValueDelete(ctx)},
847         {ctx.NewString("get_index"), BuiltinJsValueGetIndex(ctx)},
848         {ctx.NewString("set_index"), BuiltinJsValueSetIndex(ctx)},
849         {ctx.NewString("call"), BuiltinJsValueCall(ctx)},
850     })).Freeze())
851     ctx.BaseEnvironment.Let("js", ctx.NewMapOrPanic([]mellifera.MapPair{
852         {ctx.NewString("value"), ctx.BaseEnvironment.GetOrPanic("@js::value")},
853         {ctx.NewString("from_mellifera"), BuiltinJsFromMellifera(ctx)},
854         {ctx.NewString("into_mellifera"), BuiltinJsIntoMellifera(ctx)},
855         {ctx.NewString("is_undefined"), BuiltinJsIsUndefined(ctx)},
856         {ctx.NewString("is_null"), BuiltinJsIsNull(ctx)},
857         {ctx.NewString("is_boolean"), BuiltinJsIsBoolean(ctx)},
858         {ctx.NewString("is_number"), BuiltinJsIsNumber(ctx)},
859         {ctx.NewString("is_string"), BuiltinJsIsString(ctx)},
860         {ctx.NewString("is_symbol"), BuiltinJsIsSymbol(ctx)},
861         {ctx.NewString("is_array"), BuiltinJsIsArray(ctx)},
862         {ctx.NewString("is_object"), BuiltinJsIsObject(ctx)},
863         {ctx.NewString("is_function"), BuiltinJsIsFunction(ctx)},
864         {ctx.NewString("call"), BuiltinJsCall(ctx)},
865         {ctx.NewString("call_new"), BuiltinJsCallNew(ctx)},
866         {ctx.NewString("global"), BuiltinJsGlobal(ctx)},
867         {ctx.NewString("typeof"), BuiltinJsTypeof(ctx)},
868     })).Freeze())
869
870     return ctx, nil
871 }
872
873 var ctx *mellifera.Context = nil
874 var env *mellifera.Environment = nil
875 eval := func() (mellifera.Value, error) {
876     var err error = nil
877     if !options.Get("context").IsUndefined() {
878         if options.Get("context").Type() != js.TypeString {
879             return nil, fmt.Errorf("error: expected options.context to be a
↪ string, received %v\n", options.Get("context"))
880         }
881         key := options.Get("context").String()
882         cached, ok := contextCache[key]
883         if ok {
884             debugf("[mellifera.eval] using existing named context %q\n", key)

```

```

885         ctx = cached.ctx
886         env = cached.env
887     } else {
888         debugf("[mellifera.eval] using new named context %q\n", key)
889         ctx, err = newContext()
890         if err != nil {
891             return nil, err
892         }
893         env = mellifera.NewEnvironment(ctx.BaseEnvironment)
894         contextCache[key] = contextCacheEntry{ctx, env}
895     }
896 } else {
897     ctx, err = newContext()
898     if err != nil {
899         return nil, err
900     }
901     env = mellifera.NewEnvironment(ctx.BaseEnvironment)
902 }
903
904 debugf("[mellifera.eval] stdout=%+v, stderr=%+v, fs=%v\n", ctx.Stdout,
↪ ctx.Stderr, ctx.BaseEnvironment.GetOrPanic("@fs"))
905 defer func() {
906     if r := recover(); r != nil {
907         fmt.Fprintf(os.Stderr, "[mellifera.eval] encountered panic: %v\n", r)
908         debug.PrintStack()
909     }
910 }()
911
912 lexer := mellifera.NewLexer(ctx, source,
↪ &mellifera.SourceLocation{"<program>", 1})
913 parser, err := mellifera.NewParser(&lexer)
914 if err != nil {
915     return nil, err
916 }
917
918 program, err := parser.ParseProgram()
919 if err != nil {
920     return nil, err
921 }
922
923 return program.Eval(ctx, env)
924 }
925
926 _, err := eval()
927 if err != nil {
928     stderr := io.Writer(os.Stderr)
929     if ctx != nil {
930         stderr = ctx.Stderr
931     }
932
933     if e, ok := err.(mellifera.ParseError); ok && e.Location != nil {
934         fmt.Fprintf(stderr, "[%v, line %v] error: %v\n", e.Location.File,
↪ e.Location.Line, err)
935     } else if e, ok := err.(mellifera.Error); ok {
936         if e.Location != nil {
937             fmt.Fprintf(stderr, "[%v, line %v] error: %v\n", e.Location.File,
↪ e.Location.Line, err)
938         } else {
939             fmt.Fprintf(stderr, "error: %v\n", err)
940         }
941         for _, element := range e.Trace {
942             s := fmt.Sprintf("...within %v", element.FuncName)
943             if element.Location != nil {
944                 s += fmt.Sprintf(" called from %s, line %v",
↪ element.Location.File, element.Location.Line)
945             }
946             fmt.Fprintf(stderr, "%s\n", s)

```

```
947         }
948     } else {
949         fmt.Fprintf(stderr, "%v\n", err)
950     }
951 }
952
953     return nil
954 }))
955
956 // Prevent the Wasm program from exiting.
957 c := make(chan struct{}, 0)
958 <-c
959 }
```

cmd/wasm/wasm.go

bin/mf-test

```
1  #!/bin/sh
2  set -e
3  export LC_ALL=C.UTF-8
4
5  PROGNAME=$(basename "$0")
6  usage() {
7      cat <<EOF
8  Usage: ${PROGNAME} [FILE...]
9
10 Options:
11 -h          Display usage information and exit.
12 -py         Run tests using the Python reference interpreter.
13 EOF
14 }
15
16 for arg in "$@"; do
17 case "${arg}" in
18     -h)
19         usage
20         exit 0
21         ;;
22     -py|--py)
23         MELLIFERA_PROG="${MELLIFERA_HOME}/mf.py"
24         shift
25         ;;
26     *)
27         ;;
28 esac
29 done
30
31 if [ -z "${MELLIFERA_HOME}" ]; then
32     MELLIFERA_HOME=$(pwd)
33 fi
34 if [ -x "${MELLIFERA_HOME}/bin/mf" -a -z "${MELLIFERA_PROG}" ]; then
35     MELLIFERA_PROG="${MELLIFERA_HOME}/bin/mf"
36 else
37     MELLIFERA_PROG="${MELLIFERA_HOME}/mf.py"
38 fi
39
40 TESTSRUN=0
41 FAILURES=0
42
43 test() {
44     TEST="$1"
45
46     TESTSRUN=$((TESTSRUN + 1))
47     echo "[= TEST ${TEST} =]"
48
49     set +e
50     RECEIVED=$( \
51         cd "$(dirname "${TEST}")" 2>&1 && \
52         "${MELLIFERA_PROG}" "${basename "${TEST}"}" 2>&1)
53     set -e
54
55     EXPECTED=$( \
56         sed -n '/^#####\(#\)*/, $p' "${TEST}" | \
57         sed '1d' | \
58         sed -E 's/^#[[:space:]]?//g')
59 }
```

bin/mf-test

```
60     if [ "${EXPECTED}" = "${RECEIVED}" ]; then
61         echo '[= PASS =]'
62     else
63         TMPDIR=$(mktemp -d)
64         trap '{ rm -rf -- "${TMPDIR}"; }' EXIT
65         printf '%s\n' "${EXPECTED}" >"${TMPDIR}/expected"
66         printf '%s\n' "${RECEIVED}" >"${TMPDIR}/received"
67         diff "${TMPDIR}/expected" "${TMPDIR}/received" || true
68         echo '[= FAIL =]'
69         FAILURES=$((FAILURES + 1))
70     fi
71 }
72
73 TESTS= # empty
74 if [ "$#" -ne 0 ]; then
75     for arg in "$@"; do
76         if [ -d "$(realpath ${arg})" ]; then
77             FILES=$(find "$(realpath ${arg})" -name '*.test.mf' | sort)
78             TESTS=$(echo "${TESTS}" "${FILES}")
79         else
80             TESTS=$(echo "${TESTS}" "${arg}")
81         fi
82     done
83 else
84     TESTS=$(find . -name '*.test.mf' | sort)
85 fi
86
87 for t in ${TESTS}; do
88     test "${t}"
89 done
90
91 echo "TESTS RUN => ${TESTSRUN}"
92 echo "FAILURES  => ${FAILURES}"
93
94 [ "${FAILURES}" -eq 0 ] || exit 1
```

examples/basic-types-and-values.mf

```
1  # null
2  dumpLn( null );
3
4  # boolean: true or false
5  dumpLn( true );
6  dumpLn( false );
7
8  # number: IEEE-754 double
9  dumpLn( 123.456 );
10 dumpLn( Inf );
11 dumpLn( NaN );
12
13 # string: byte strings with assumed UTF-8 encoding
14 dumpLn( "hello 🍔" );
15 dumpLn( "hello\t🍔" ); # string with escape sequences
16 dumpLn( `hello\t🍔` ); # raw string where escape sequences are not processed
17 let burger = "🍔";
18 dumpLn( $"hello\t{burger}" ); # template string with string interpolation
19 dumpLn( `$hello\t{burger}` ); # template string using raw string literal
20
21 # regexp: regular expressions
22 dumpLn( r`(\w+) (\w+)` );
23
24 # vector: ordered collection of elements
25 dumpLn( [] );
26 dumpLn( ["foo", "bar", "baz"] );
27
28 # map: collection of key-value pairs with unique keys
29 dumpLn( Map{} ); # empty map requires Map to disambiguate from the empty set
30 dumpLn( {"foo": 123, "bar": 456, "baz": ["abc", 789]} );
31 dumpLn( {foo = 123, bar = 456, baz = ["abc", 789]} ); # alternate syntax
32
33 # set: collection unique elements
34 dumpLn( Set{} ); # empty set requires Set to disambiguate from the empty map
35 dumpLn( {"foo", "bar", "baz", ["abc", 789]} );
36
37 # reference: pointer-like construct to allow in-place mutation and data sharing
38 dumpLn( 123.& ); # .& is the postfix addressof operator
39
40 # function: first-class function values
41 dumpLn( function() { println("hello"); } );
42 let add = function(a, b) { return a + b; };
43 dumpLn( add );
44 println($"add(123, 456) is {add(123, 456)}");
```


examples/creating-references.mf

```
1 # Reference created by using the reference operator .& on a function argument.
2 let f = function(x) {
3     println(`${x} is {x} of type {typename(x)}`);
4     println(`${x.*} is {x.*} of type {typename(x.*)}`);
5 };
6 let v = ["foo", "bar", "baz"];
7 f(v.&);
8
9 print("\n");
10
11 # Reference created referring to each vector element in a for loop.
12 let inventory = [
13     {"item": "apples", "quantity": 12},
14     {"item": "bananas", "quantity": 30},
15     {"item": "cherries", "quantity": 8},
16 ];
17 println(`${inventory} starts as {inventory}`);
18 for stock.& in inventory {
19     println(`${Doubling the quantity of {stock.*.item}}`);
20     stock.*.quantity = stock.*.quantity * 2;
21 }
22 println(`${inventory} ends as {inventory}`);
23
24 print("\n");
25
26 # Reference created referring to each map value in a for loop.
27 let bank_accounts = {
28     "Ashley": {"checking": 350, "savings": 1500},
29     "Joseph": {"checking": 120, "savings": 45},
30 };
31 println(`${bank_accounts} starts as {bank_accounts}`);
32 for person, account.& in bank_accounts {
33     println(`${Adding $100 to {person}'s savings account}`);
34     account.*.savings = account.*.savings + 100;
35 }
36 println(`${bank_accounts} ends as {bank_accounts}`);
37
38 print("\n");
39
40 # Reference implicitly created and passed as the first argument when a method
41 # declared with function.& is invoked with the dot operator.
42 let person = type {
43     .init = function(name, age) {
44         return new person { .name = name, .age = age };
45     },
46     .birthday = function.&(self) {
47         println(`${Happy birthday {self.name}! 🎂`);
48         self.*.age = self.*.age + 1;
49     },
50 };
51 let ashley = person::init("Ashley", 21);
52 println(`${ashley} starts as {ashley}`);
53 ashley.birthday(); # ashley.& is implicitly passed as self to person::birthday()
54 println(`${ashley} ends as {ashley}`);
```


examples/depth-first-search.mf

```
1 let node = type {
2   .init = function(name, children) {
3     return new node {"name": name, "children": children};
4   },
5   .into_string = function(self) {
6     return self.name;
7   },
8 };
9
10 let graph = type {
11   .init = function(nodes) {
12     let self = new graph {"nodes": Map{}};
13     for node in nodes {
14       self.nodes[node.name] = node;
15     }
16     return self;
17   },
18   .into_string = function(self) {
19     let elements = [];
20     for node in self.nodes {
21       elements.push(string::init(node));
22     }
23     return "{" + ", ".join(elements) + "}";
24   },
25 };
26
27 let dfs = function(graph) {
28   let visited = Set{};
29   let ordered = [];
30
31   let visit = function(node) {
32     if visited.contains(node) {
33       return;
34     }
35     visited.insert(node);
36     for name in node.children {
37       visit(graph.nodes[name]);
38     }
39     ordered.insert(0, node);
40   };
41
42   for name, node in graph.nodes {
43     visit(node);
44   }
45   return ordered;
46 };
47
48 let dfs_cycle_detecting = function(graph) {
49   let visiting = Set{};
50   let visited = Set{};
51   let ordered = [];
52
53   let visit = function(node) {
54     if visited.contains(node) {
55       return;
56     }
57     if visiting.contains(node) {
58       error $"cycle detected in graph {string::init(graph)}";
59     }

```

```

60     visiting.insert(node);
61     for name in node.children {
62         visit(graph.nodes[name]);
63     }
64     visited.insert(node);
65     ordered.insert(0, node);
66 };
67
68     for name, node in graph.nodes {
69         visit(node);
70     }
71     return ordered;
72 };
73
74 # B-->E-->A--+
75 # |   |   |   |
76 # |   v   |   |
77 # +-->C<--+ |
78 #       |       v
79 #       +---->D
80
81 let a = node::init("A", {"C", "D"});
82 let b = node::init("B", {"C", "E"});
83 let c = node::init("C", {"D"});
84 let d = node::init("D", Set{});
85 let e = node::init("E", {"A", "C"});
86
87 let ordered = dfs(graph::init({a, b, c, d, e}));
88 println("Ordered:");
89 for x in ordered {
90     println(x);
91 }
92
93 # Validate that the cycle-detecting and non-cycle-detecting depth-first search
94 # produce identical results for acyclic directed graphs.
95 assert(ordered == dfs_cycle_detecting(graph::init({a, b, c, d, e})));
96
97 # +->W-->X--+
98 # |   |   |   |
99 # |   |   v   v
100 # |   +-->Y->Z
101 # |           |
102 # +-----+
103
104 let w = node::init("W", {"X", "Y"});
105 let x = node::init("X", {"Y", "Z"});
106 let y = node::init("Y", {"Z"});
107 let z = node::init("Z", {"W"});
108
109 try {
110     dfs_cycle_detecting(graph::init({w, x, y, z}));
111 }
112 catch err {
113     println($"error: {err}");
114 }

```

examples/exceptions.mf

```
1  try {
2    fs::read("/path/to/file/that/does/not/exist.txt");
3  }
4  catch err {
5    eprintln($"error: {err}");
6  }
7
8  eprint("\n");
9
10 let g = function() {
11   let f = function() {
12     error "oopsie";
13   };
14   f();
15 };
16 function() {
17   let h = function() {
18     g();
19   };
20   h();
21 }();
```


examples/explicit-references.mf

```
1 let pass_by_value = function(val) {
2   println($"[inside pass_by_value] val starts as {val}");
3
4   # Here val is a copy, so the mutation performed by `val.push` will not
5   # change the original value outside of this function's lexical scope.
6   val.push(123);
7
8   println($"[inside pass_by_value] val ends as {val}");
9 };
10
11 let pass_by_reference = function(ref) {
12   println($"[inside pass_by_reference] ref starts as {ref} with referenced value
13   ↪ {ref.*}");
14
15   # Here ref is a reference to the original value outside of this function's
16   # lexical scope, so mutation performed by `ref.*.push` will change that
17   # original value.
18   ref.*.push(123); # .* is the postfix dereference operator
19
20   println($"[inside pass_by_reference] ref ends as {ref} with referenced value
21   ↪ {ref.*}");
22 };
23
24 let x = ["foo", "bar", "baz"];
25 println($"[outside pass_by_value] x starts as {x}");
26 pass_by_value(x);
27 println($"[outside pass_by_value] x ends as {x}");
28
29 print("\n");
30
31 let y = ["abc", "def", "ghi"];
32 println($"[outside pass_by_reference] y starts as {y}");
33 pass_by_reference(y.&); # .& is the postfix reference operator
34 println($"[outside pass_by_reference] y ends as {y}");
```


examples/fibonacci-iterative.mf

```
1  # Usage:
2  #   mf examples/fibonacci-iterative.mf n
3
4  let fibonacci = function(n) {
5      if n <= 1 {
6          return n;
7      }
8      let a = 0; # fibonacci(0)
9      let b = 1; # fibonacci(1)
10
11      for _ in range(2, n+1) {
12          let c = a + b;
13          a = b;
14          b = c;
15      }
16
17      return b;
18  };
19
20  if argv.count() != 2 {
21      error "expected exactly one program argument";
22  }
23  println(fibonacci(number::init(argv[1])));
```


examples/fibonacci-recursive-memoized.mf

```
1 # Usage:
2 #   mf examples/fibonacci-recursive-memoized.mf n
3
4 let cache = {
5     0: 0, # fibonacci(0)
6     1: 1, # fibonacci(1)
7 };
8 let fibonacci = function(n) {
9     if not cache.contains(n) {
10         cache[n] = fibonacci(n - 1) + fibonacci(n - 2);
11     }
12     return cache[n];
13 };
14
15
16 if argv.count() != 2 {
17     error "expected exactly one program argument";
18 }
19 println(fibonacci(number::init(argv[1])));
```


examples/fibonacci-recursive.mf

```
1  # Usage:
2  #   mf examples/fibonacci-recursive.mf n
3
4  let fibonacci = function(n) {
5      if n <= 1 {
6          return n;
7      }
8      return fibonacci(n - 1) + fibonacci(n - 2);
9  };
10
11 if argv.count() != 2 {
12     error "expected exactly one program argument";
13 }
14 println(fibonacci(number::init(argv[1])));
```


examples/fizzbuzz.mf

```
1  let N = 30;
2  for i in N {
3      let n = i + 1;
4      if n % 3 == 0 and n % 5 == 0 {
5          println("fizzbuzz");
6          continue;
7      }
8      if n % 3 == 0 {
9          println("fizz");
10         continue;
11     }
12     if n % 5 == 0 {
13         println("buzz");
14         continue;
15     }
16     println(n);
17 }
```


examples/greeter.mf

```
1 # Greeter module used for explaining imports.
2 #
3 # let greeter = import("/path/to/greeter.mf");
4 # let greet = greeter::init("Juliet");
5 # greet(); # prints the line "Hello, Juliet!"
6 let init = function(name) {
7     return function() {
8         println($"Hello, {name}!");
9     };
10 };
11 return {"init": init};
```


examples/hello.mf

```
1 println("hello 🐝");
```

examples/hello.mf

examples/program-arguments.mf

```
1 # Usage:
2 #   mf examples/program-arguments.mf foo bar baz
3
4 println($"program arguments: {argv}");
```


examples/regular-expressions.mf

```
1 let strings = [  
2     "Johnny Appleseed",  
3     "Some long text that does not match!",  
4     "Isaac Newton"  
5 ];  
6 for s in strings {  
7     if s =~ r`^(\w+) (\w+)$` {  
8         println("First Name = {repr($1)}, Last Name = {repr($2)}");  
9     }  
10 }
```


examples/sieve-of-eratosthenes.mf

```
1  let sieve = function(n) {
2    if n < 2 {
3      return [];
4    }
5
6    let end = n + 1;
7    let primes = [];
8    for i in end {
9      primes.push(true);
10   }
11   primes[0] = false;
12   primes[1] = false;
13
14   for i in range(2, end) {
15     if not primes[i] {
16       continue;
17     }
18
19     let multiple = i * i;
20     while multiple < end {
21       primes[multiple] = false;
22       multiple = multiple + i;
23     }
24   }
25
26   let result = [];
27   for i in range(2, end) {
28     if primes[i] {
29       result.push(i);
30     }
31   }
32   return result;
33 };
34
35 if argv.count() != 2 { # program.mf n
36   error "expected exactly one program argument";
37 }
38 println(sieve(number::init(argv[1])));
```


examples/user-defined-iterators.mf

```
1  let fizzbuzz = type extends iterator {
2    .init = function(n, max) {
3      return new fizzbuzz { "n": n, "max": max };
4    },
5    .next = function.&(self) {
6      let n = self.n;
7      if n > self.max {
8        error null; # error null signals end-of-iteration
9      }
10     self.n = self.n + 1;
11     if n % 3 == 0 and n % 5 == 0 {
12       return "fizzbuzz";
13     }
14     if n % 3 == 0 {
15       return "fizz";
16     }
17     if n % 5 == 0 {
18       return "buzz";
19     }
20     return n;
21   },
22 };
23
24 let fb = fizzbuzz::init(1, 15);
25 for x in fb {
26   println(x);
27 }
```


examples/user-defined-types.mf

```
1 let vec2 = type {
2     .fixed = function(self, ndigits) {
3         return new vec2 {
4             .x = self.x.fixed(ndigits),
5             .y = self.y.fixed(ndigits),
6         };
7     },
8     .magnitude = function(self) {
9         return math::sqrt(self.x * self.x + self.y * self.y);
10    },
11 };
12
13 let va = new vec2 {.x = 3, .y = 4};
14 println($`va is {va} with type {typename(va)}`);
15 println($`va.magnitude() is {va.magnitude()}`);
16
17 print("\n");
18
19 let vb = new vec2 {.x = math::e, .y = -math::pi};
20 println($`vb is {vb} with type {typename(vb)}`);
21 println($`vb.magnitude() is {vb.magnitude()}`);
22 println($`vb.fixed(3) is {vb.fixed(3)}`);
```


examples/value-semantics-and-structural-equality.mf

```
1 let x = ["foo", {"bar": 123}, "baz"];
2 let y = x; # x is assigned to y by copy
3 println($`x is {x}`);
4 println($`y is {y}`);
5 # x and y are separate values that are structurally equal
6 println($`x == y is {x == y}`);
7
8 print("\n");
9
10 # updates to x and y do not affect each other, because they are separate values
11 x[0] = "abc";
12 y[1]["bar"] = "xyz";
13 println($`x is {x}`);
14 println($`y is {y}`);
15 # x and y are no longer structurally equal as their contents now differ
16 println($`x == y is {x == y}`);
17
18 print("\n");
19
20 let z = ["foo", {"bar": "xyz"}, "baz"];
21 println($`z is {z}`);
22 # y and z are separate values that are structurally equal
23 println($`y == z is {y == z}`);
```


examples/word-count-simple.mf

```
1  #!/usr/bin/env mf
2  # usage: cat FILE | word-count-simple.mf
3  let words = re::split(input(), r`\s+`)
4      .map(function(word) {
5          return re::replace(word.to_lower(), r`[^\\w!]'`, "");
6      })
7      .filter(function(word) {
8          return word.count() != 0;
9      });
10
11  let counts = Map{};
12  for word in words {
13      counts[word] = counts.get(word, 0) + 1;
14  }
15
16  let ordered = counts
17      .pairs()
18      .sorted_by(function(lhs, rhs) {
19          return rhs.value - lhs.value;
20      });
21  for pair in ordered {
22      println("${pair.key} {pair.value}");
23  }
```


examples/word-count.mf

```
1  #!/usr/bin/env mf
2  let help = function(writeln) {
3      writeln($``
4  usage: {argv[0]} [file]
5
6  positional arguments:
7      file          Text file to process (defaults to stdin).
8
9  optional arguments:
10     --json        Output results as a JSON object.
11     --top=n       Output only the top n words.
12     -h, --help    Display this help text and exit.
13     ```.trim());
14 };
15
16 let file = null;
17 let text = null;
18 let fmt = null;
19 let top = null;
20 let argi = 1;
21 while argi < argv.count() {
22     if argv[argi] =~ r`^--json$` {
23         fmt = "json";
24         argi = argi + 1;
25         continue;
26     }
27     if argv[argi] =~ r`^--top=(.*)?$` {
28         if $2 != null {
29             top = number::init($2);
30         }
31         else {
32             argi = argi + 1;
33             top = number::init(argv[argi]);
34         }
35         argi = argi + 1;
36         continue;
37     }
38     if argv[argi] =~ r`^--h(elp)?$` {
39         help(println);
40         exit(0);
41     }
42
43     if file != null and text != null {
44         error $"multiple input files, {file} and {argv[argi]}";
45     }
46     file = argv[argi];
47     text = fs::read(file);
48     argi = argi + 1;
49 }
50 if file == null and text == null {
51     file = "<stdin>";
52     text = input();
53 }
54
55 let occurrences = Map{};
56 let words = re::split(text, r`\s+`)
57     .into_iterator()
58     .map(function(word) {
59         return re::replace(word.to_lower(), r`[^w']`, "");
```

```
60     })
61     .filter(function(word) {
62         return word.count() != 0;
63     });
64 for word in words {
65     occurrences[word] = occurrences.get(word, 0) + 1;
66 }
67
68 let ordered = occurrences
69     .pairs()
70     .sorted_by(function(lhs, rhs) {
71         return rhs.value - lhs.value;
72     })
73     .into_iterator()
74     .map(function(pair) {
75         return {
76             .word = pair.key,
77             .count = pair.value,
78         };
79     })
80     .into_vector();
81
82 if top != null {
83     ordered = ordered.slice(0, min(top, ordered.count()));
84 }
85
86 if fmt == "json" {
87     let map = Map{};
88     for x in ordered {
89         map[x.word] = x.count;
90     }
91     println(json::encode(map));
92 }
93 else {
94     for x in ordered {
95         println("${x.word} {x.count}");
96     }
97 }
```