



enumerable, streamable, understandable

NATURAL SET

*Learning about protocols and streams by
implementing a new data type from scratch*



Luciano Ramalho
@ramalhoorg



Sets FTW!



The **MapSet** API

NaturalSet under the hood

Q&A

WHY USE SETS

Logic!

Nobody has yet discovered a branch of mathematics that has successfully resisted formalization into set theory.

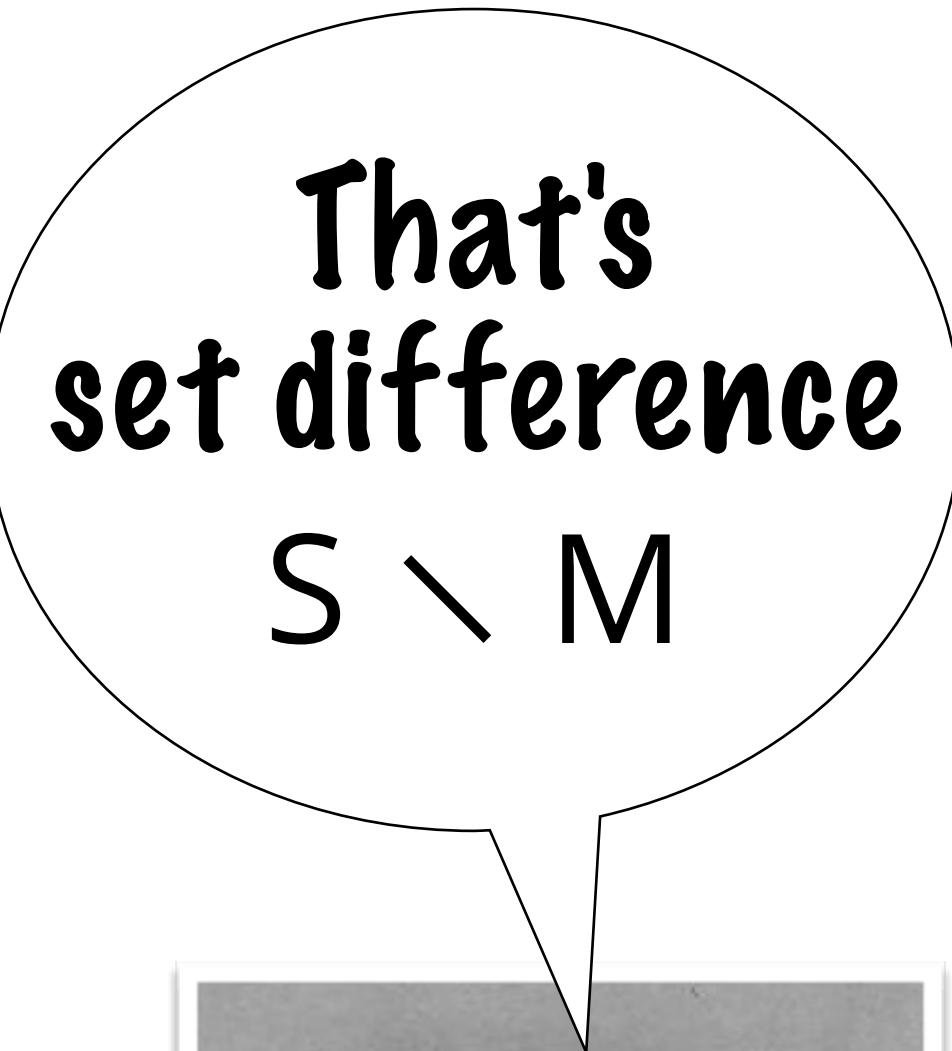
*Thomas Forster
Logic Induction and Sets, p. 167*

USE CASE #1: NEWS PAGE

The screenshot shows a web browser window for 'Business & Financial News, U.S.' at <https://www.reuters.com>. The main content area (M) displays several news headlines from 'THE WIRE' section, such as 'West accepts Moscow was right to approve COVID vaccine: Russian official' and 'Baltic states impose sanctions on Lukashenko and other Belarus officials'. Below these are articles like 'Scientists see downsides to vaccines from Russia, China' and 'Oregon state police called to Portland as officials warn of escalating violence'. A sidebar (S) on the left lists recent news items, including 'Zimbabwe says foreign white farmers can apply to get back seized land' and 'Tight security, and an Arabic greeting, on first Israel-UAE flight'. The top navigation bar includes links for Business, Markets, World, Politics, Tech, Breakingviews, Wealth, Stocks, Bonds, Commodities, and Currencies. The S&P 500 index is shown at 3,509.43 (+0.04%).

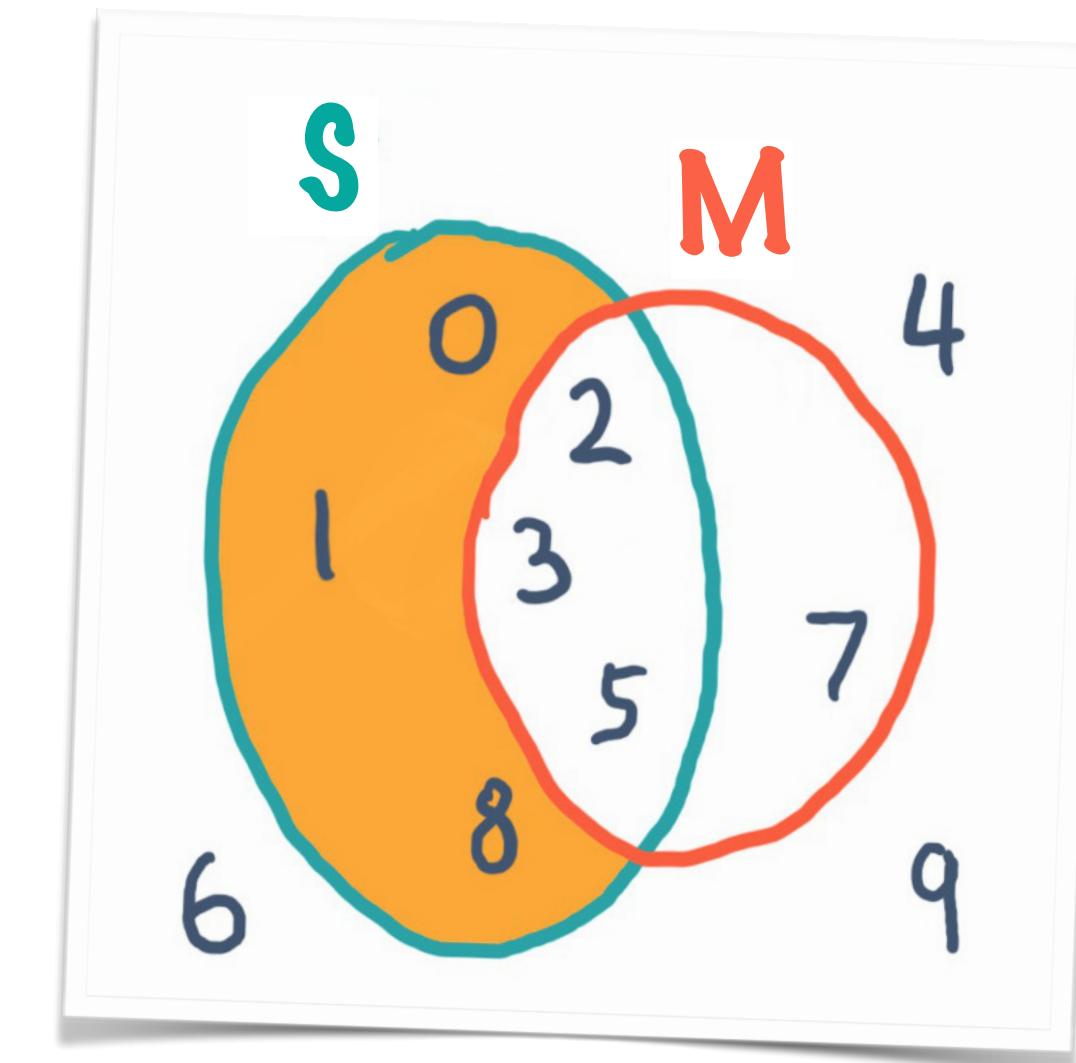
Show newest headlines
on side bar S, excluding
headlines shown in the
main content area M.

USE CASE #1: NEWS PAGE



That's
set difference
 $S \setminus M$

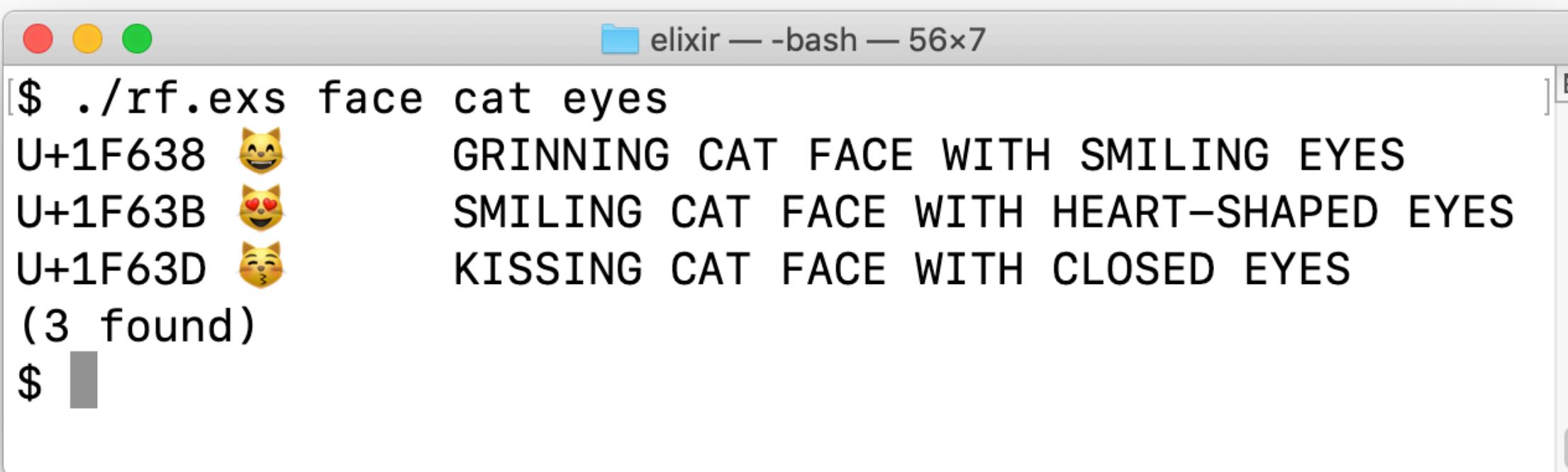
Show newest headlines
on side bar S, excluding
headlines shown in the
main content area M.



Georg Cantor in 1870 (age 25)

USE CASE #2: UNICODE DATABASE (FLAT FILE SCAN)

Show character if all words in the query Q appear in name field N.



A terminal window titled "elixir — -bash — 56x7" displays the output of a command. The command is "./rf.exs face cat eyes". The output shows three results, each consisting of a Unicode code point followed by a emoji and its corresponding name:

Code Point	Emoji	Name
U+1F638	😺	GRINNING CAT FACE WITH SMILING EYES
U+1F63B	😻	SMILING CAT FACE WITH HEART-SHAPED EYES
U+1F63D	😽	KISSING CAT FACE WITH CLOSED EYES

(3 found)

USE CASE #2: UNICODE DATABASE (FLAT FILE SCAN)

That's
a subset test!
 $Q \subseteq N$



Show character if all
words in the query Q
appear in name field N.

query = ["FACE", "CAT", "EYES"]

...
1F637;FACE WITH MEDICAL MASK
1F638;GRINNING CAT FACE WITH SMILING EYES
1F639;CAT FACE WITH TEARS OF JOY
1F63A;SMILING CAT FACE WITH OPEN MOUTH
1F63B;SMILING CAT FACE WITH HEART-SHAPED EYES
1F63C;CAT FACE WITH WRY SMILE
1F63D;KISSING CAT FACE WITH CLOSED EYES
1F63E;POUTING CAT FACE
1F63F;CRYING CAT FACE
1F640;WEARY CAT FACE
1F641;SLIGHTLY FROWNING FACE
1F642;SLIGHTLY SMILING FACE
1F643;UPSIDE-DOWN FACE
1F644;FACE WITH ROLLING EYES
...

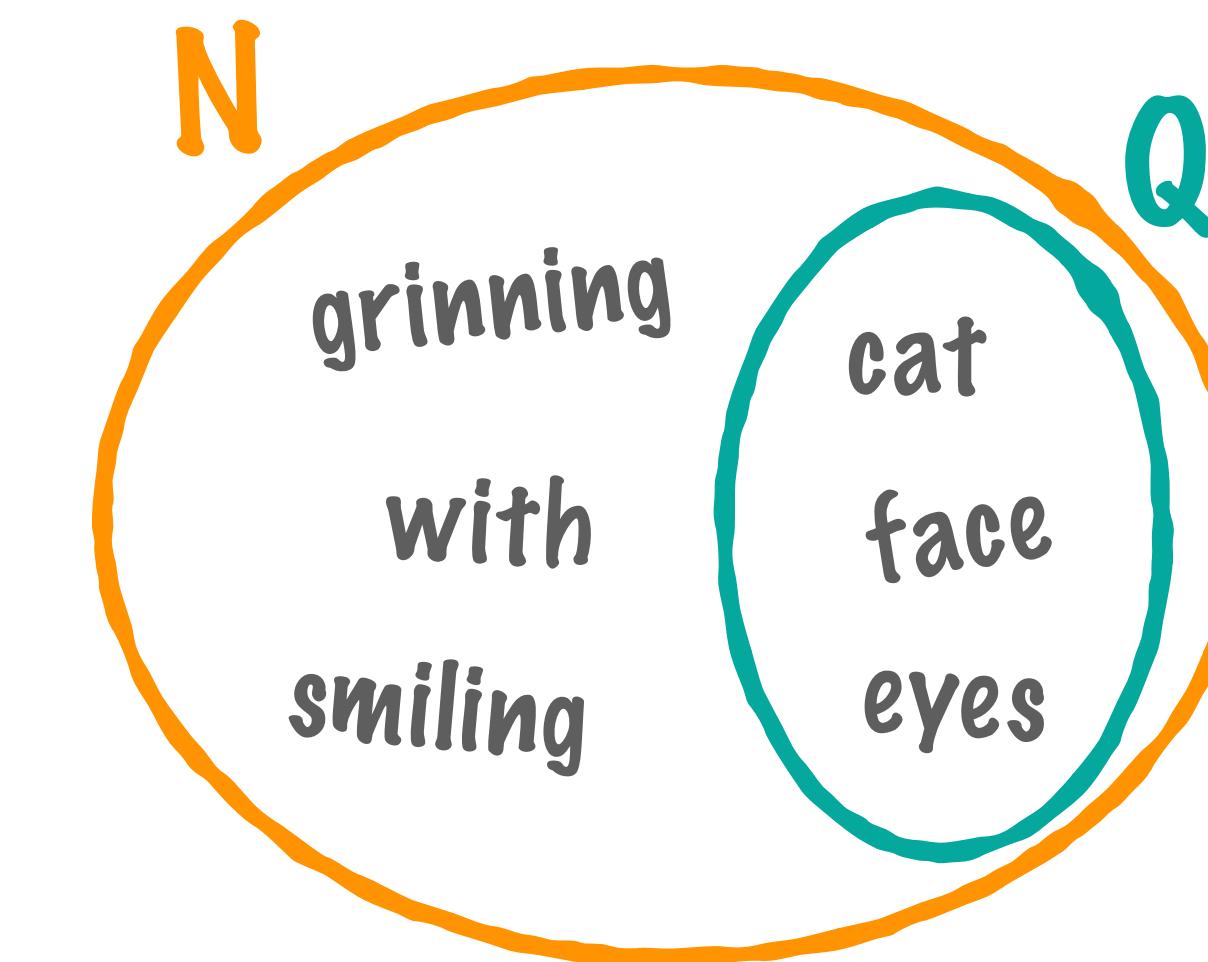
USE CASE #2: UNICODE DATABASE (FLAT FILE SCAN)

That's
a subset test!

$$Q \subseteq N$$

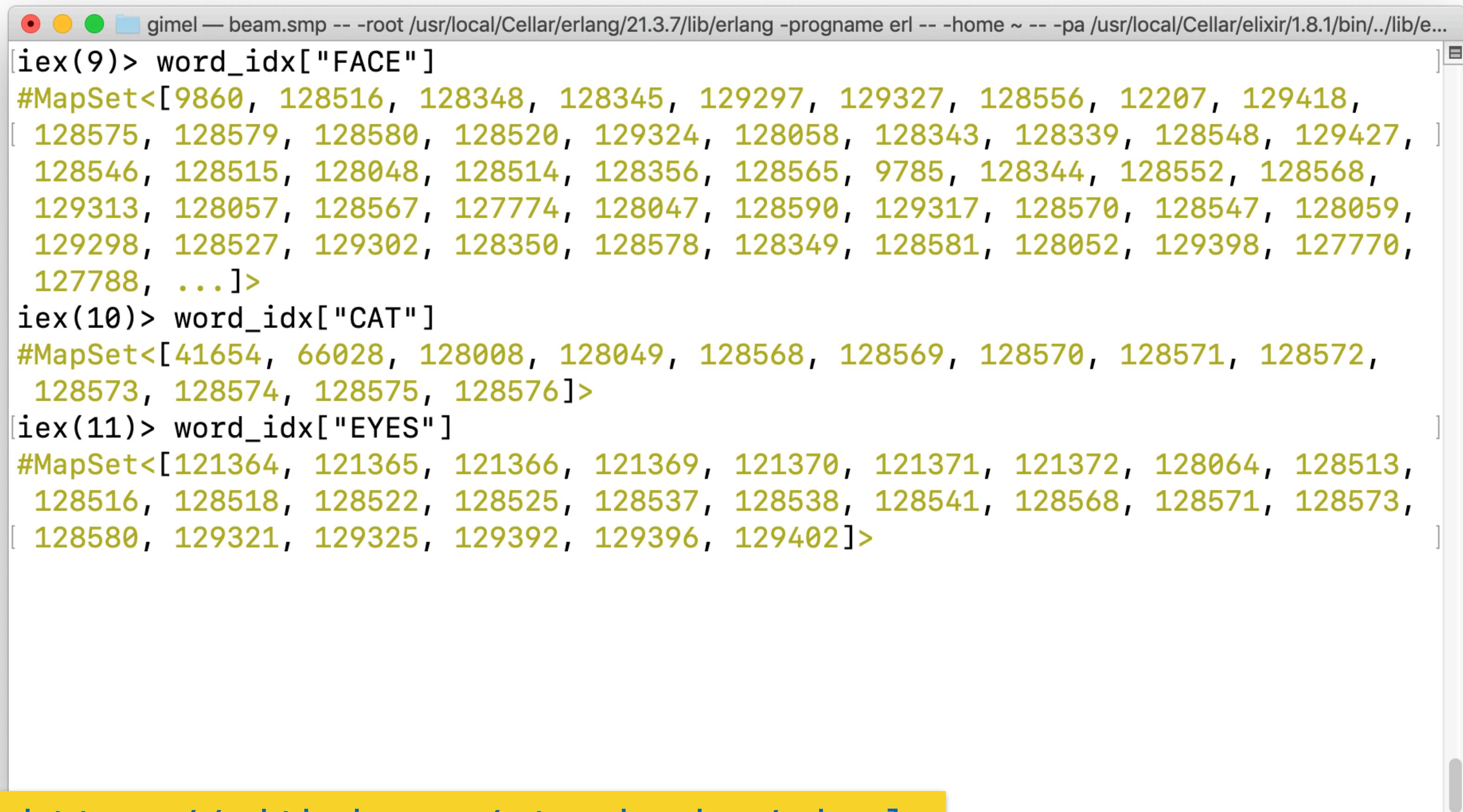


Show character if all words in the query Q appear in name field N.



USE CASE #3: UNICODE DATABASE (INVERTED INDEX)

Given a mapping from each word (eg. "FACE") to a set of code points with that word in their names (eg. 9860, 128516, etc.)...



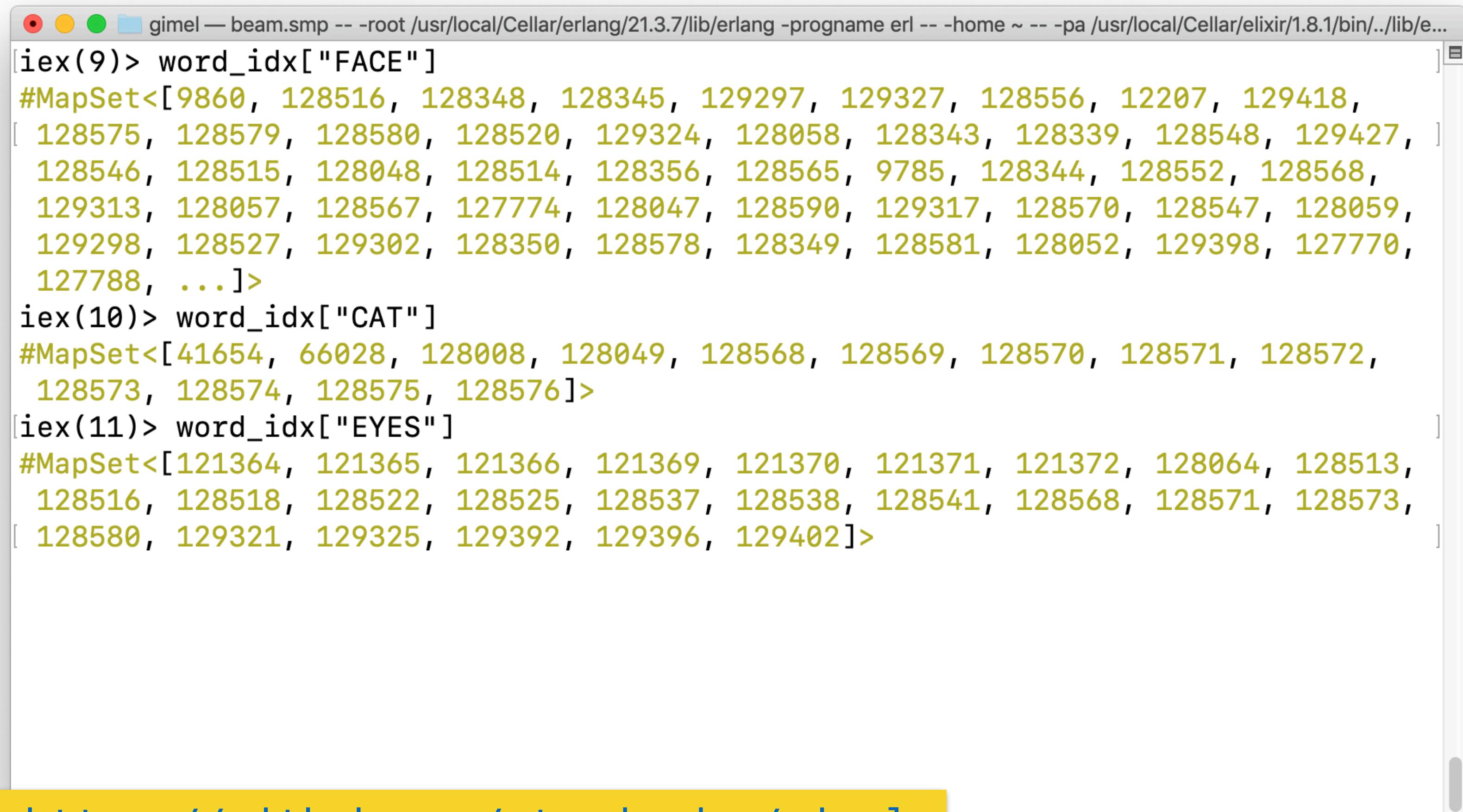
The screenshot shows a terminal window titled "gimel — beam.smp". The window displays three Elixir iex sessions:

- iex(9) > word_idx["FACE"]
#MapSet<[9860, 128516, 128348, 128345, 129297, 129327, 128556, 12207, 129418, 128575, 128579, 128580, 128520, 129324, 128058, 128343, 128339, 128548, 129427, 128546, 128515, 128048, 128514, 128356, 128565, 9785, 128344, 128552, 128568, 129313, 128057, 128567, 127774, 128047, 128590, 129317, 128570, 128547, 128059, 129298, 128527, 129302, 128350, 128578, 128349, 128581, 128052, 129398, 127770, 127788, ...]>
- iex(10) > word_idx["CAT"]
#MapSet<[41654, 66028, 128008, 128049, 128568, 128569, 128570, 128571, 128572, 128573, 128574, 128575, 128576]>
- iex(11) > word_idx["EYES"]
#MapSet<[121364, 121365, 121366, 121369, 121370, 121371, 121372, 128064, 128513, 128516, 128518, 128522, 128525, 128537, 128538, 128541, 128568, 128571, 128573, 128580, 129321, 129325, 129392, 129396, 129402]>

source: <https://github.com/standupdev/gimel>

USE CASE #3: UNICODE DATABASE (INVERTED INDEX)

To find emoji with the words "CAT FACE EYES" you need to compute...



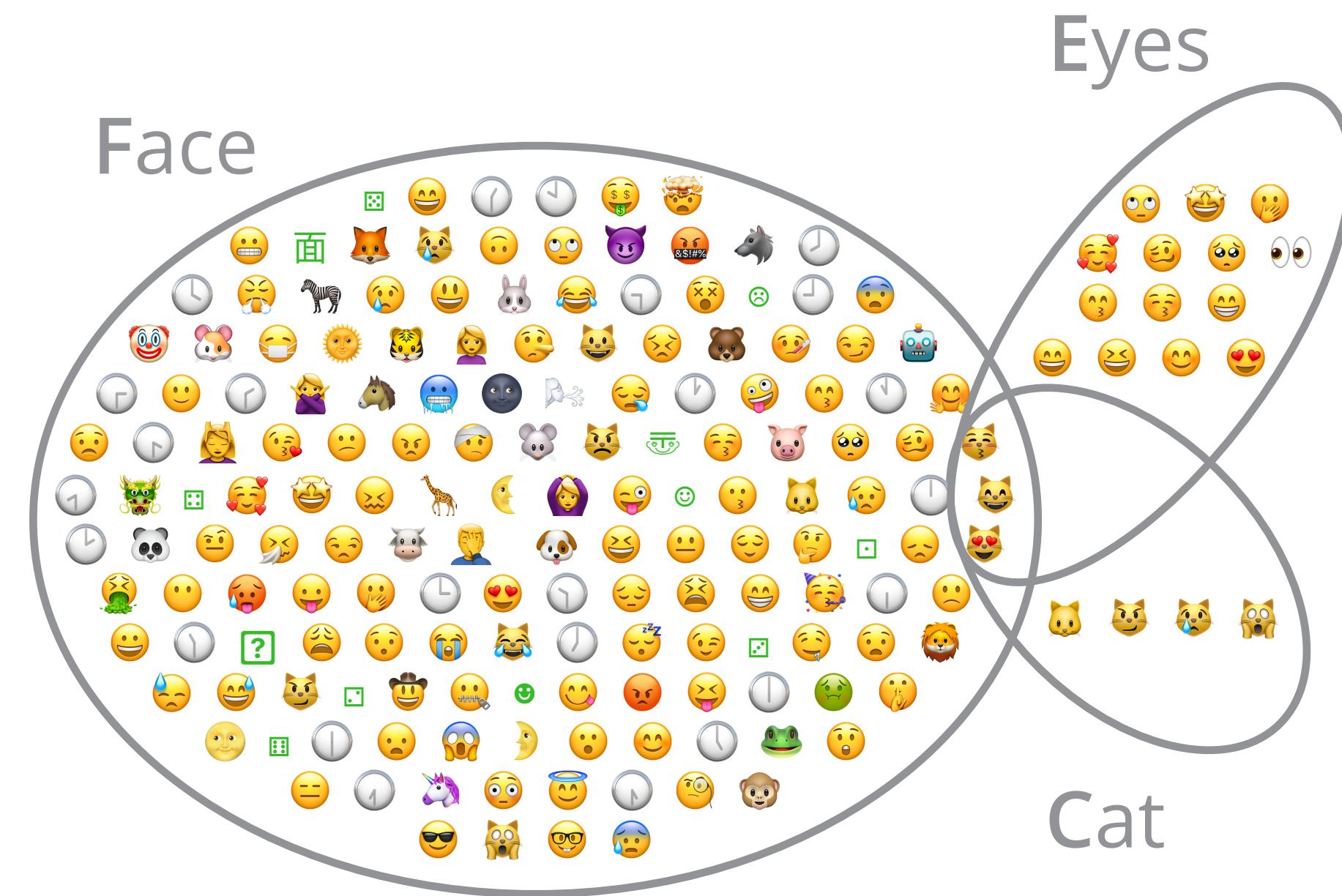
The screenshot shows a terminal window titled "gimel — beam.smp". The code demonstrates an inverted index for emoji words:

```
iex(9)> word_idx["FACE"]
#MapSet<[9860, 128516, 128348, 128345, 129297, 129327, 128556, 12207, 129418,
[ 128575, 128579, 128580, 128520, 129324, 128058, 128343, 128339, 128548, 129427,
128546, 128515, 128048, 128514, 128356, 128565, 9785, 128344, 128552, 128568,
129313, 128057, 128567, 127774, 128047, 128590, 129317, 128570, 128547, 128059,
129298, 128527, 129302, 128350, 128578, 128349, 128581, 128052, 129398, 127770,
127788, ...]>
iex(10)> word_idx["CAT"]
#MapSet<[41654, 66028, 128008, 128049, 128568, 128569, 128570, 128571, 128572,
128573, 128574, 128575, 128576]>
iex(11)> word_idx["EYES"]
#MapSet<[121364, 121365, 121366, 121369, 121370, 121371, 121372, 128064, 128513,
128516, 128518, 128522, 128525, 128537, 128538, 128541, 128568, 128571, 128573,
128580, 129321, 129325, 129392, 129396, 129402]>
```

source: <https://github.com/standupdev/gimel>

USE CASE #3: UNICODE DATABASE (INVERTED INDEX)

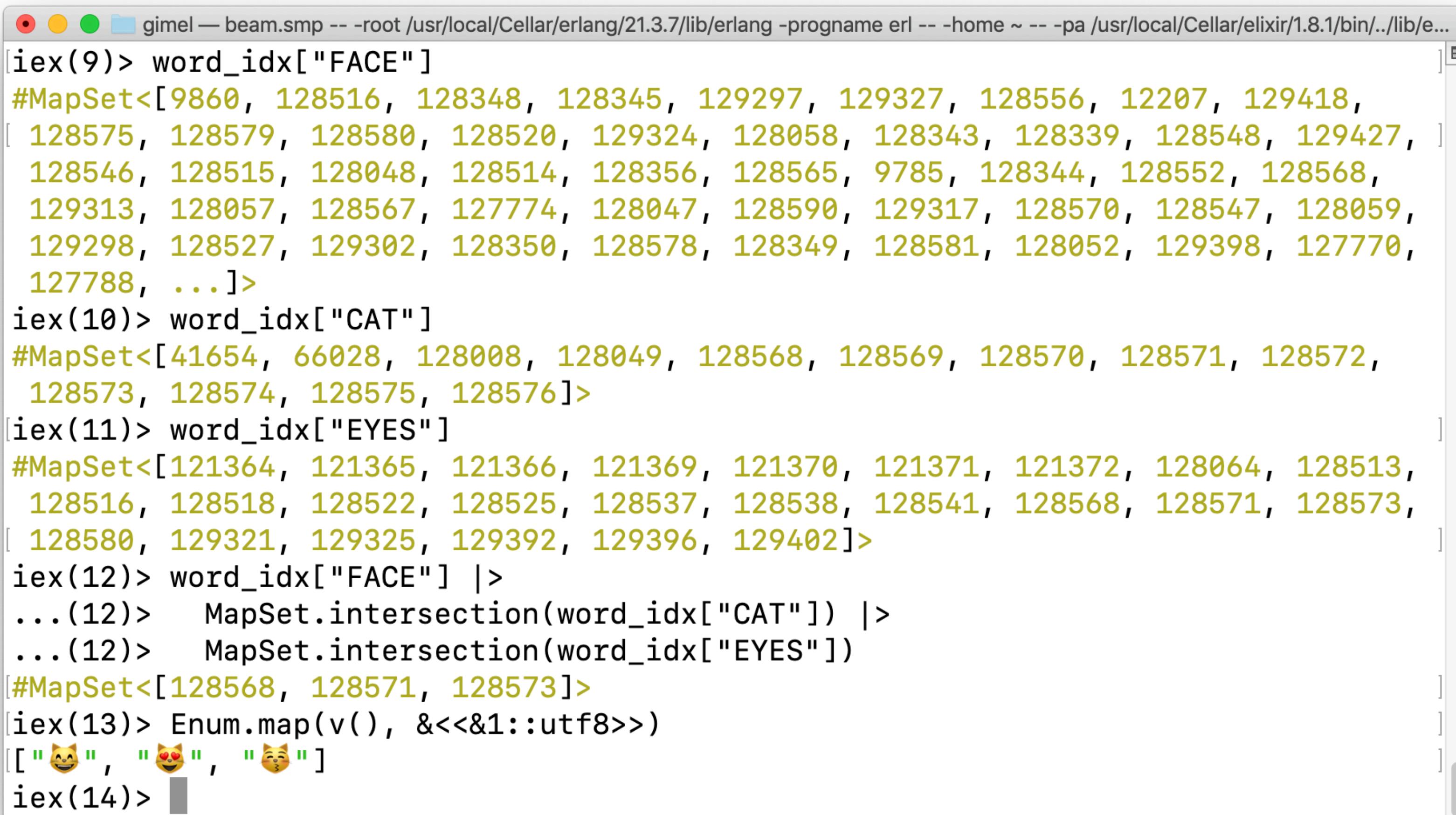
That's
intersection of
intersection!
 $(F \cap E) \cap C$



Simplified diagram. There are more characters in: $F \cap C$, $F \cap E$, $C \cap E$

USE CASE #3: UNICODE DATABASE (INVERTED INDEX)

Fortunately, Elixir **MapSet** provides **intersection/2**:



```
gimel — beam.smp -- -root /usr/local/Cellar/erlang/21.3.7/lib/erlang -proname erl -- -home ~ -- -pa /usr/local/Cellar/elixir/1.8.1/bin/../lib/e...
[iex(9)> word_idx["FACE"]
#MapSet<[9860, 128516, 128348, 128345, 129297, 129327, 128556, 12207, 129418,
[ 128575, 128579, 128580, 128520, 129324, 128058, 128343, 128339, 128548, 129427,
128546, 128515, 128048, 128514, 128356, 128565, 9785, 128344, 128552, 128568,
129313, 128057, 128567, 127774, 128047, 128590, 129317, 128570, 128547, 128059,
129298, 128527, 129302, 128350, 128578, 128349, 128581, 128052, 129398, 127770,
127788, ...]>
iex(10)> word_idx["CAT"]
#MapSet<[41654, 66028, 128008, 128049, 128568, 128569, 128570, 128571, 128572,
128573, 128574, 128575, 128576]>
[iex(11)> word_idx["EYES"]
#MapSet<[121364, 121365, 121366, 121369, 121370, 121371, 121372, 128064, 128513,
128516, 128518, 128522, 128525, 128537, 128538, 128541, 128568, 128571, 128573,
128580, 129321, 129325, 129392, 129396, 129402]>
iex(12)> word_idx["FACE"] |>
...(12)> MapSet.intersection(word_idx["CAT"]) |>
...(12)> MapSet.intersection(word_idx["EYES"])
#[#MapSet<[128568, 128571, 128573]>
[iex(13)> Enum.map(v(), &<<&1::utf8>>)
[ "😺", "😺", "😺"]
iex(14)>
```

source: <https://github.com/standupdev/gimel>

USE CASE #3: UNICODE DATABASE (INVERTED INDEX)

Fortunately, Elixir **MapSet** provides **intersection/2**:

```
iex(12)> word_idx["FACE"] |>
...> MapSet.intersection(word_idx["CAT"]) |>
...> MapSet.intersection(word_idx["EYES"])
[#MapSet<[128568, 128571, 128573]>
iex(13)> Enum.map(v(), &<<&1::utf8>>)
[ "😺", "😻", "😽" ]
```

source: <https://github.com/standupdev/gimel>

THE MAPSET API

How it compares

MAPSET: ELEMENT API

Any basic set API supports these operations:

new()	Creates new empty MapSet .
new(enum)	Creates new MapSet with elements from enumerable.
new(enum, transform)	Same as above, applying transform to each element.

member?(set, element)	Is element included MapSet ? Same as: $e \in M$
put(set, element)	Inserts element . No-op if element already in set .
delete(set, element)	Removes element from set .
size(set)	Returns number of elements in set .
to_list(set)	Builds new list from elements in set .

MAPSET: ELEMENT API

Any basic set API supports these operations:

new()	Creates new empty MapSet .
new(enum)	Creates new MapSet with elements from enumerable.
new(enum, transform)	Same as above, applying transform to each element.

member?(set, element)	Is element included MapSet ? Same as: $e \in M$
put(set, element)	Inserts element . No-op if element already in set .
delete(set, element)	Removes element from set .
size(set)	Returns number of elements in set .
to_list(set)	Builds new list from elements in set .

These are all the operations JS ES6 gives you...

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus

IBM Research Laboratory, San Jose



General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

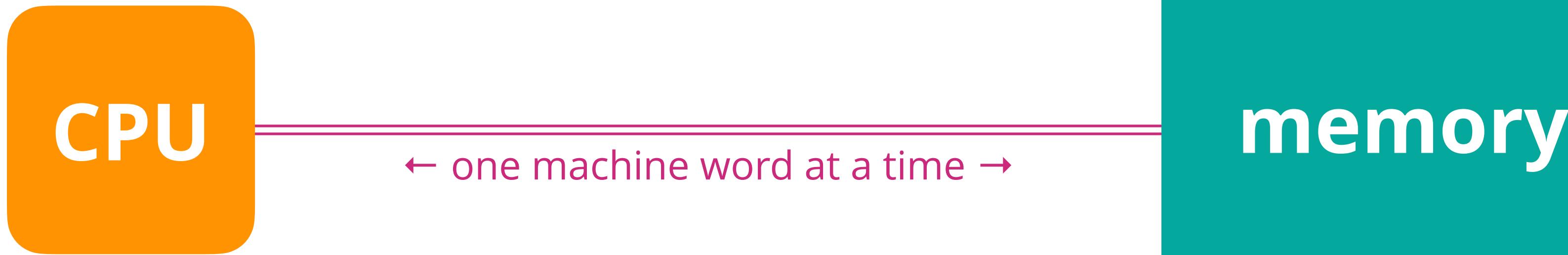
© 1978 ACM 0001-0782/78/0800-0613 \$00.75

613

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

THE VON NEUMANN BOTTLENECK



MAPSET: SET API

Operations between whole sets:
declarative code, no error-prone looping.

intersection(set1, set2)	Intersection between sets.	$A \cap B$
union(set1, set2)	Union of two sets.	$A \cup B$
difference(set1, set2)	Difference of set1 — set2 .	$A \setminus B$

subset?(set1, set2)	Are all elements of set1 in set2 ?	$A \subseteq B$
equal?(set1, set2)	Do set1 and set2 have <i>all equal</i> elements?	$A = B$
disjoint?(set1, set2)	Do set1 and set2 have <i>only distinct</i> elements?	$A \cap B = \emptyset$

MAPSET: SET API

Operations between whole sets:
declarative code, no error-prone looping.

intersection(set1, set2)	Intersection between sets.	$A \cap B$
union(set1, set2)	Union of two sets.	$A \cup B$
difference(set1, set2)	Difference of set1 — set2 .	$A \setminus B$

subset?(set1, set2)	Are all elements of set1 in set2 ?	$A \subseteq B$
equal?(set1, set2)	Do set1 and set2 have <i>all equal</i> elements?	$A = B$
disjoint?(set1, set2)	Do set1 and set2 have <i>only distinct</i> elements?	$A \cap B = \emptyset$

Beyond the von Neumann bottleneck!

NATURAL SET

A didactic set type

SOURCE OF THE IDEA: THE GO PROGRAMMING LANGUAGE

The Go Programming Language

Alan A. A. Donovan & Brian W. Kernighan

6.5. Example: Bit Vector Type

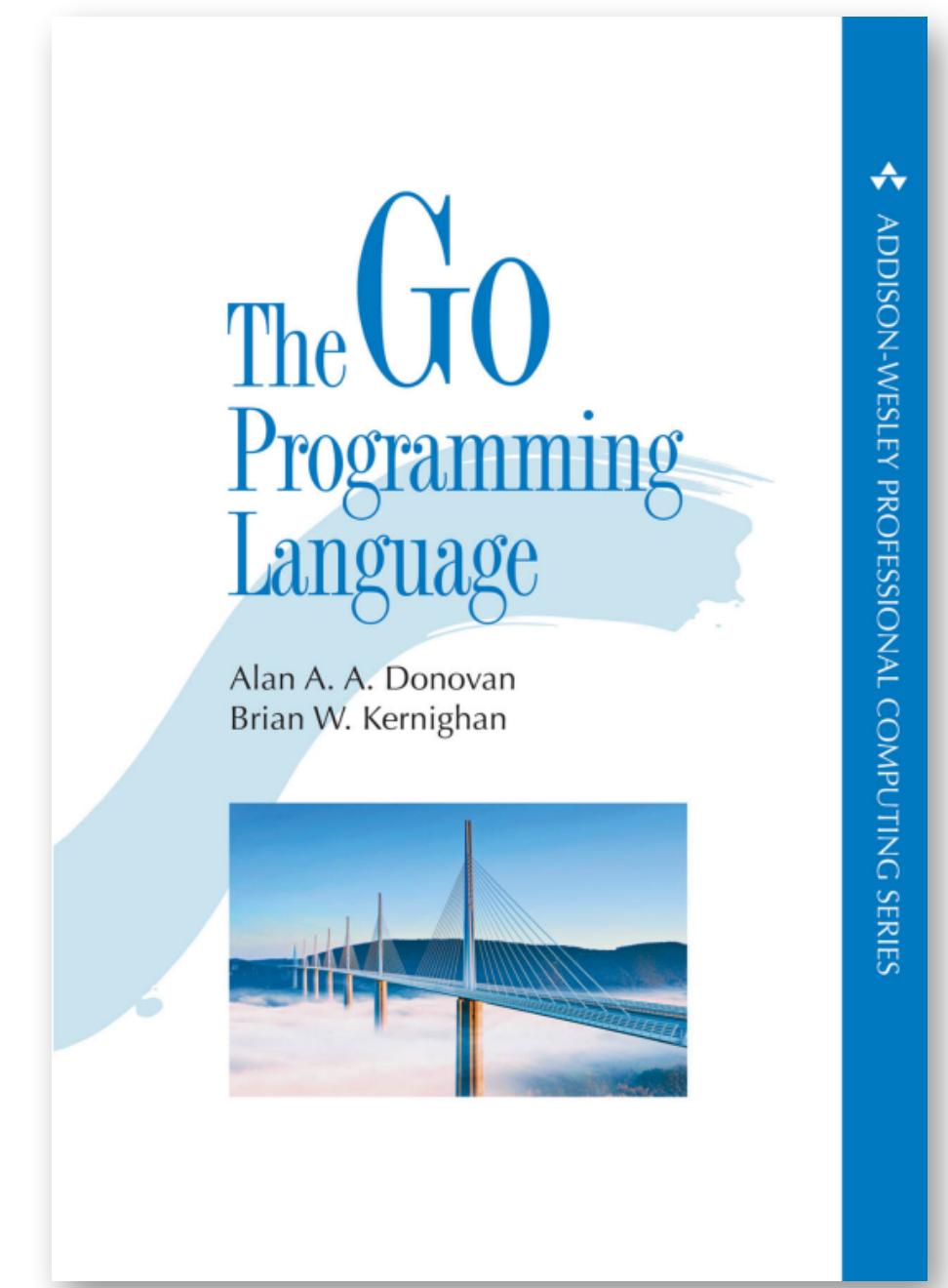
Sets in Go are usually implemented as a `map[T]bool`, where `T` is the element type. A set represented by a map is very flexible but, for certain problems, a specialized representation may outperform it. For example, in domains such as dataflow analysis where set elements are small non-negative integers, sets have many elements, and set operations like union and intersection are common, a *bit vector* is ideal.

A bit vector uses a slice of unsigned integer values or “words,” each bit of which represents a possible element of the set. The set contains i if the i -th bit is set. The following program demonstrates a simple bit vector type with three methods:

```
gopl.io/ch6/intset

// An IntSet is a set of small non-negative integers.
// Its zero value represents the empty set.
type IntSet struct {
    words []uint64
}

// Has reports whether the set contains the non-negative value x.
func (s *IntSet) Has(x int) bool {
    word, bit := x/64, uint(x%64)
    return word < len(s.words) && s.words[word]&(1<<bit) != 0
}
```



THE NATURAL SET IMPLEMENTATION

Show me the code!

THE CODE

natural_set hex package

128 LOC (docstrings excluded)

567 LOC total (with docstrings + test module)

```
1 defmodule NaturalSet do
2   use Bitwise, only_operators: true
3
4   defstruct bits: 0
5
6   def new, do: %NaturalSet{}
7
8   def new(enumerable) do
9     Enum.reduce(enumerable, %NaturalSet{}, &NaturalSet.put(&2, &1))
10
11 end
12
13 def new(enumerable, transform) when is_function(transform, 1) do
14   enumerable
15   |> Stream.map(transform)
16   |> new
17 end
18
19 def put(%NaturalSet{bits: bits}, element) when is_integer(element) and element >= 0 do
20   %NaturalSet{bits: 1 <<< element ||| bits}
21 end
22
23 def member?(%NaturalSet{bits: bits}, element) when is_integer(element) and element >= 0 do
24   (bits >> element &&& 1) == 1
25 end
26
27 def delete(natural_set, element) when is_integer(element) and element >= 0 do
28   if member?(natural_set, element) do
29     new_bits = (1 <<< element) ^~ natural_set.bits
30     %NaturalSet{bits: new_bits}
31   else
32     natural_set # return unchanged
33   end
34 end
35
36 def equal?(%NaturalSet{bits: bits1}, %NaturalSet{bits: bits2}) do
37   bits1 == bits2
38 end
39
40 def intersection(%NaturalSet{bits: bits1}, %NaturalSet{bits: bits2}) do
41   %NaturalSet{bits: bits1 &&& bits2}
42 end
43
```

```
45   def union(%NaturalSet{bits: bits1}, %NaturalSet{bits: bits2}) do
46     %NaturalSet{bits: bits1 ||| bits2}
47   end
48
49   def difference(%NaturalSet{bits: bits1}, %NaturalSet{bits: bits2}) do
50     %NaturalSet{bits: bits1 &&& bits2 ^~^ bits1}
51   end
52
53   def subset?(natural_set1, natural_set2) do
54     difference(natural_set1, natural_set2).bits == 0
55   end
56
57   def disjoint?(natural_set1, natural_set2) do
58     intersection(natural_set1, natural_set2).bits == 0
59   end
60
61   def length(%NaturalSet{bits: bits}) do
62     count_ones(bits, 0)
63   end
64
65   defp count_ones(0, count), do: count
66
67   defp count_ones(bits, count) do
68     count = count + (bits &&& 1)
69     count_ones(bits >>> 1, count)
70   end
71
72   def stream(%NaturalSet{bits: bits}) do
73     Stream.unfold({bits, 0}, &next_one/1)
74   end
75
76   defp next_one({0, _index}), do: nil
77
78   defp next_one({bits, index}) when (bits &&& 1) == 1 do
79     # LSB is one: return {next_element, new_accumulator_tuple}
80     {index, {bits >> 1, index + 1}}
81   end
82
83   defp next_one({bits, index}) do
84     # LSB is 0: shift bits; increment index; try again
85     next_one({bits >> 1, index + 1})
86   end
87
```

```
88   def to_list(natural_set) do
89     natural_set |> stream |> Enum.to_list
90   end
91
92   defimpl Enumerable do
93     def count(natural_set) do
94       {:ok, NaturalSet.length(natural_set)}
95     end
96
97     def member?(natural_set, val) do
98       {:ok, NaturalSet.member?(natural_set, val)}
99     end
100
101    def slice(_set) do
102      {:error, __MODULE__}
103    end
104
105    def reduce(natural_set, acc, fun) do
106      Enumerable.List.reduce(NaturalSet.to_list(natural_set), acc, fun)
107    end
108
109    defimpl Collectable do
110      def into(original) do
111        collector_fun = fn
112          set, {<:cont, elem} => NaturalSet.put(set, elem)
113          set, :done => set
114          _set, :halt => :ok
115        end
116
117        {original, collector_fun}
118      end
119    end
120
121    defimpl Inspect do
122      import Inspect.Algebra
123
124      def inspect(natural_set, opts) do
125        concat(["#NaturalSet<", Inspect.List.inspect(NaturalSet.to_list(natural_set), opts), ">"])
126      end
127    end
128  end
129
```

https://hex.pm/packages/natural_set

MAKING A NATURAL SET

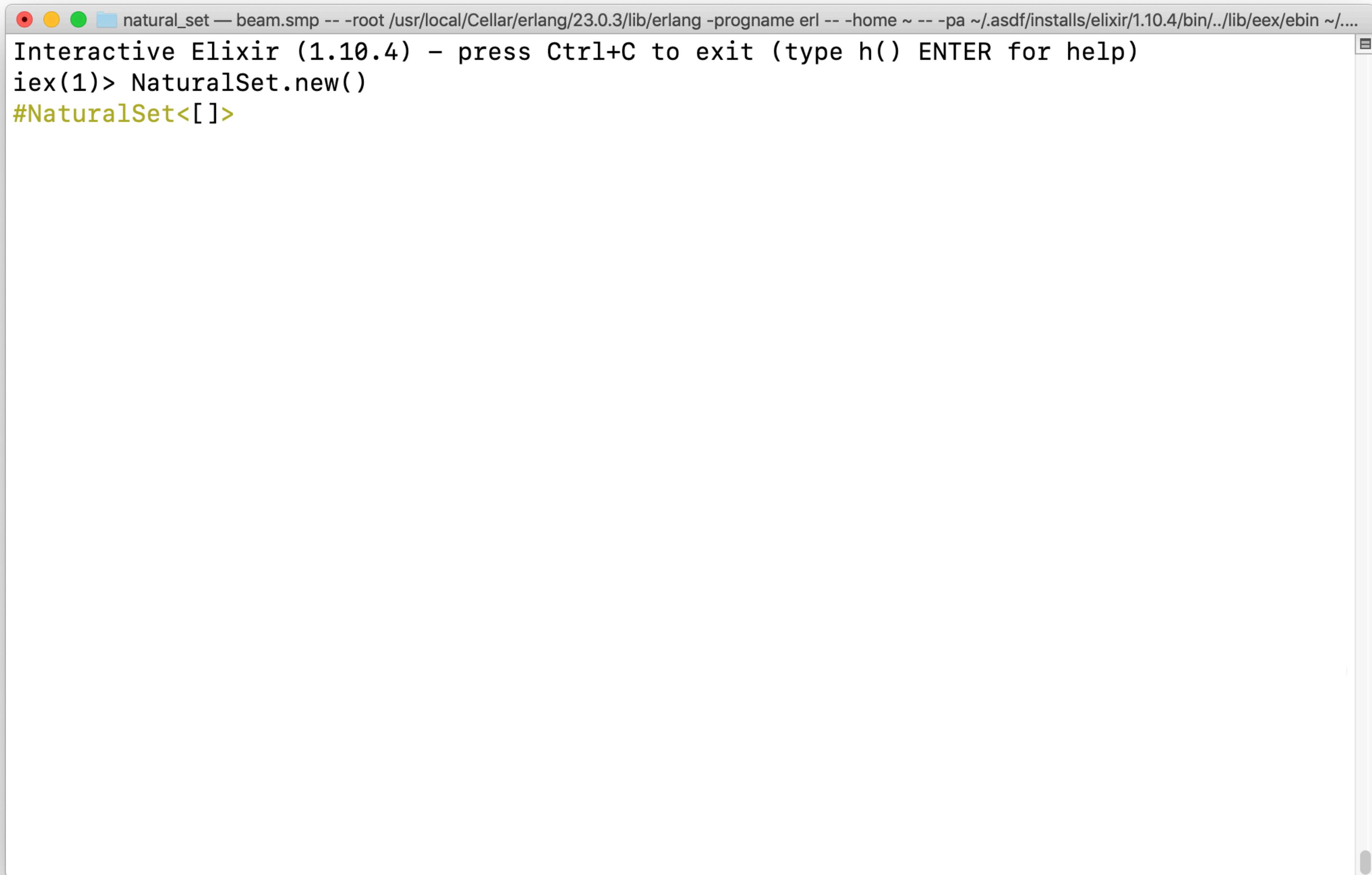
```
1 defmodule NaturalSet do
2
3   use Bitwise, only_operators: true
4
5   defstruct bits: 0
6
7   def new, do: %NaturalSet{}
8
9   def new(enumerable) do
10    Enum.reduce(enumerable, %NaturalSet{}, &NaturalSet.put(&2, &1))
11  end
12
13  def new(enumerable, transform) when is_function(transform, 1) do
14    enumerable
15    |> Stream.map(transform)
16    |> new
17  end
```

https://hex.pm/packages/natural_set

USING ONE INTEGER AS A BIT VECTOR

Bits all the way down

DEMO: NATURAL SETS AS BITS



The screenshot shows a terminal window titled "natural_set — beam.smp -- -root /usr/local/Cellar/erlang/23.0.3/lib/erlang -progname erl -- -home ~ -- -pa ~/asdf/install/elixir/1.10.4/bin/../lib/eex/ebin ~/...." running on an Apple Mac OS X system. The window title bar includes the standard OS X window controls (red, yellow, green, blue). The terminal output is as follows:

```
Interactive Elixir (1.10.4) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> NaturalSet.new()
#NaturalSet<[]>
```

DEMO: NATURAL SETS AS BITS

```
● ○ ● natural_set — beam.smp -- -root /usr/local/Cellar/erlang/23.0.3/lib/erlang -prognome erl -- -home ~ -- -pa ~/asdf/install/elixir/1.10.4/bin/../lib/eex/ebin ~/....  
Interactive Elixir (1.10.4) - press Ctrl+C to exit (type h() ENTER for help)  
iex(1)> NaturalSet.new()  
#NaturalSet<[]>  
iex(2)> NaturalSet.new().bits  
0
```

DEMO: NATURAL SETS AS BITS

```
● ○ ● natural_set — beam.smp -- -root /usr/local/Cellar/erlang/23.0.3/lib/erlang -prognome erl -- -home ~ -- -pa ~/asdf/install/elixir/1.10.4/bin/../lib/eex/ebin ~/....  
Interactive Elixir (1.10.4) - press Ctrl+C to exit (type h() ENTER for help)  
iex(1)> NaturalSet.new()  
#NaturalSet<[]>  
iex(2)> NaturalSet.new().bits  
0  
iex(3)> NaturalSet.new([0]).bits  
1
```

DEMO: NATURAL SETS AS BITS

```
● ○ ● natural_set — beam.smp -- -root /usr/local/Cellar/erlang/23.0.3/lib/erlang -prognome erl -- -home ~ -- -pa ~/asdf/install/elixir/1.10.4/bin/../lib/eex/ebin ~/....  
Interactive Elixir (1.10.4) - press Ctrl+C to exit (type h() ENTER for help)  
iex(1)> NaturalSet.new()  
#NaturalSet<[]>  
iex(2)> NaturalSet.new().bits  
0  
iex(3)> NaturalSet.new([0]).bits  
1  
iex(4)> NaturalSet.new([1]).bits  
2
```

DEMO: NATURAL SETS AS BITS

```
● ○ ● natural_set — beam.smp -- -root /usr/local/Cellar/erlang/23.0.3/lib/erlang -progname erl -- -home ~ -- -pa ~/asdf/install/elixir/1.10.4/bin/../lib/eex/ebin ~/....  
Interactive Elixir (1.10.4) - press Ctrl+C to exit (type h() ENTER for help)  
iex(1)> NaturalSet.new()  
#NaturalSet<[]>  
iex(2)> NaturalSet.new().bits  
0  
iex(3)> NaturalSet.new([0]).bits  
1  
iex(4)> NaturalSet.new([1]).bits  
2  
iex(5)> NaturalSet.new([1]).bits |> Integer.to_string(2)  
"10"
```

DEMO: NATURAL SETS AS BITS

```
● ○ ● natural_set — beam.smp -- -root /usr/local/Cellar/erlang/23.0.3/lib/erlang -progname erl -- -home ~ -- -pa ~/asdf/install/elixir/1.10.4/bin/../lib/eex/ebin ~/....  
Interactive Elixir (1.10.4) - press Ctrl+C to exit (type h() ENTER for help)  
iex(1)> NaturalSet.new()  
#NaturalSet<[]>  
iex(2)> NaturalSet.new().bits  
0  
iex(3)> NaturalSet.new([0]).bits  
1  
iex(4)> NaturalSet.new([1]).bits  
2  
iex(5)> NaturalSet.new([1]).bits |> Integer.to_string(2)  
"10"  
iex(6)> NaturalSet.new([0, 1]).bits |> Integer.to_string(2)  
"11"
```

DEMO: NATURAL SETS AS BITS

```
● ○ ● natural_set — beam.smp -- -root /usr/local/Cellar/erlang/23.0.3/lib/erlang -progname erl -- -home ~ -- -pa ~/asdf/install/elixir/1.10.4/bin/../lib/eex/ebin ~/....  
Interactive Elixir (1.10.4) - press Ctrl+C to exit (type h() ENTER for help)  
iex(1)> NaturalSet.new()  
#NaturalSet<[]>  
iex(2)> NaturalSet.new().bits  
0  
iex(3)> NaturalSet.new([0]).bits  
1  
iex(4)> NaturalSet.new([1]).bits  
2  
iex(5)> NaturalSet.new([1]).bits |> Integer.to_string(2)  
"10"  
iex(6)> NaturalSet.new([0, 1]).bits |> Integer.to_string(2)  
"11"  
iex(7)> NaturalSet.new(1..5).bits |> Integer.to_string(2)  
"111110"
```

DEMO: NATURAL SETS AS BITS

```
● ○ ● natural_set — beam.smp -- -root /usr/local/Cellar/erlang/23.0.3/lib/erlang -progname erl -- -home ~ -- -pa ~/asdf/install/elixir/1.10.4/bin/../lib/eex/ebin ~/....  
Interactive Elixir (1.10.4) - press Ctrl+C to exit (type h() ENTER for help)  
iex(1)> NaturalSet.new()  
#NaturalSet<[]>  
iex(2)> NaturalSet.new().bits  
0  
iex(3)> NaturalSet.new([0]).bits  
1  
iex(4)> NaturalSet.new([1]).bits  
2  
iex(5)> NaturalSet.new([1]).bits |> Integer.to_string(2)  
"10"  
iex(6)> NaturalSet.new([0, 1]).bits |> Integer.to_string(2)  
"11"  
iex(7)> NaturalSet.new(1..5).bits |> Integer.to_string(2)  
"111110"  
iex(8)> NaturalSet.new([1, 3, 5]).bits |> Integer.to_string(2)  
"101010"
```

DEMO: NATURAL SETS AS BITS

```
● ○ ● natural_set — beam.smp -- -root /usr/local/Cellar/erlang/23.0.3/lib/erlang -progname erl -- -home ~ -- -pa ~/asdf/install/elixir/1.10.4/bin/../lib/eex/ebin ~/....  
Interactive Elixir (1.10.4) - press Ctrl+C to exit (type h() ENTER for help)  
iex(1)> NaturalSet.new()  
#NaturalSet<[]>  
iex(2)> NaturalSet.new().bits  
0  
iex(3)> NaturalSet.new([0]).bits  
1  
iex(4)> NaturalSet.new([1]).bits  
2  
iex(5)> NaturalSet.new([1]).bits |> Integer.to_string(2)  
"10"  
iex(6)> NaturalSet.new([0, 1]).bits |> Integer.to_string(2)  
"11"  
iex(7)> NaturalSet.new(1..5).bits |> Integer.to_string(2)  
"111110"  
iex(8)> NaturalSet.new([1, 3, 5]).bits |> Integer.to_string(2)  
"101010"  
iex(9)> NaturalSet.new([0, 2, 4]).bits |> Integer.to_string(2)  
"10101"
```

DEMO: NATURAL SETS AS BITS

DEMO: NATURAL SETS AS BITS

DEMO: NATURAL SETS AS BITS

SET OPERATIONS

Bit vector reconstruction

ELEMENT BY ELEMENT OPERATIONS

```
19  def put(%NaturalSet{bits: bits}, element) when is_integer(element) and element >= 0 do
20    %NaturalSet{bits: 1 <<< element ||| bits}
21  end
22
23  def member?(%NaturalSet{bits: bits}, element) when is_integer(element) and element >= 0 do
24    (bits >>> element &&& 1) == 1
25  end
26
27  def delete(natural_set, element) when is_integer(element) and element >= 0 do
28    if member?(natural_set, element) do
29      new_bits = (1 <<< element) ^^^ natural_set.bits
30      %NaturalSet{bits: new_bits}
31    else
32      natural_set # return unchanged
33    end
34  end
```

FLIPPING BITS

That's what computers are made for

ZOOM-IN: HOW TO PUT AN ELEMENT

```
def put(%NaturalSet{bits: bits}, element) when is_integer(element) and element >= 0  
    %NaturalSet{bits: 1 <<< element ||| bits}  
end
```

ZOOM-IN: HOW TO PUT AN ELEMENT

```
def put(%NaturalSet{bits: bits}, element) when is_integer(element) and element >= 0 do
  %NaturalSet{bits: 1 <<< element ||| bits}
end
```

The screenshot shows an iex session window. The title bar says "natural_set_demo — beam.smp -- -root /usr/local/Cellar/erlang/23.0.3/lib/erl...". The session starts with "iex(1)> use Bitwise", followed by "Bitwise". Then "iex(2)> ns = NaturalSet.new([0, 4, 5])" creates a new NaturalSet with elements [0, 4, 5]. The output is "#NaturalSet<[0, 4, 5]>". Next, "iex(3)> ns.bits" prints the value 49. Finally, "iex(4)> v |> inspect(base: :binary)" prints the binary representation "0b110001".

```
iex(1)> use Bitwise
Bitwise
iex(2)> ns = NaturalSet.new([0, 4, 5])
#NaturalSet<[0, 4, 5]>
iex(3)> ns.bits
49
iex(4)> v |> inspect(base: :binary)
"0b110001"
```

Given **ns** with elements **[0, 4, 5]**,
then **ns.bits** is **49**, a.k.a. **0b110001**.

ZOOM-IN: HOW TO PUT AN ELEMENT

```
def put(%NaturalSet{bits: bits}, element) when is_integer(element) and element >= 0 do
  %NaturalSet{bits: 1 <<< element ||| bits}
end
```

The screenshot shows an Erlang iex session. The session starts with the command `use Bitwise`. Then, it creates a `NaturalSet` with elements [0, 4, 5] and inspects its `bits` attribute, which is shown as the binary string "0b110001". Next, it defines a variable `element = 2` and performs a left shift operation `1 <<< element`, resulting in the binary string "0b100".

```
iex(1)> use Bitwise
Bitwise
iex(2)> ns = NaturalSet.new([0, 4, 5])
#NaturalSet<[0, 4, 5]>
iex(3)> ns.bits
49
iex(4)> v |> inspect(base: :binary)
"0b110001"
iex(5)> element = 2
2
iex(6)> 1 <<< element
4
iex(7)> v |> inspect(base: :binary)
"0b100"
```

Given **ns** with elements **[0, 4, 5]**,
then **ns.bits** is **49**, a.k.a. **0b110001**.

To put element **2**:

- shift **1** left by **2**:
result is **4**, a.k.a. **0b100**

ZOOM-IN: HOW TO PUT AN ELEMENT

```
def put(%NaturalSet{bits: bits}, element) when is_integer(element) and element >= 0 do
  %NaturalSet{bits: 1 <<< element ||| bits}
end
```

The screenshot shows an Erlang iex session with the following transcript:

```
iex(1)> use Bitwise
Bitwise
iex(2)> ns = NaturalSet.new([0, 4, 5])
#NaturalSet<[0, 4, 5]>
iex(3)> ns.bits
49
iex(4)> v |> inspect(base: :binary)
"0b110001"
iex(5)> element = 2
2
iex(6)> 1 <<< element
4
iex(7)> v |> inspect(base: :binary)
"0b100"
iex(8)> 0b100 ||| ns.bits
53
iex(9)> v |> inspect(base: :binary)
"0b110101"
```

Given **ns** with elements [0, 4, 5],
then **ns.bits** is 49, a.k.a. **0b110001**.

To put element 2:

- shift 1 left by 2:
result is 4, a.k.a. **0b100**
- bitwise OR **0b100** with **ns.bits**:
result is 53, a.k.a. **0b110101**

ZOOM-IN: HOW TO PUT AN ELEMENT

```
def put(%NaturalSet{bits: bits}, element) when is_integer(element) and element >= 0 do
  %NaturalSet{bits: 1 <<< element ||| bits}
end
```

The screenshot shows an Erlang iex session. The session starts with the command `use Bitwise`. Then, it creates a `NaturalSet` with elements [0, 4, 5] and inspects its bits, which are shown as the binary string "0b110001". Next, it shifts the bit 1 left by 2 positions to get the binary string "0b100". It then performs a bitwise OR operation between "0b100" and "0b110001" to get the result "0b110101". Finally, it creates a new `NaturalSet` with elements [0, 2, 4, 5].

```
iex(1)> use Bitwise
Bitwise
iex(2)> ns = NaturalSet.new([0, 4, 5])
#NaturalSet<[0, 4, 5]>
iex(3)> ns.bits
49
iex(4)> v |> inspect(base: :binary)
"0b110001"
iex(5)> element = 2
2
iex(6)> 1 <<< element
4
iex(7)> v |> inspect(base: :binary)
"0b100"
iex(8)> 0b100 ||| ns.bits
53
iex(9)> v |> inspect(base: :binary)
"0b110101"
iex(10)> %NaturalSet{bits: 0b110101}
#NaturalSet<[0, 2, 4, 5]>
iex(11)>
```

Given **ns** with elements **[0, 4, 5]**,
then **ns.bits** is **49**, a.k.a. **0b110001**.

To put element **2**:

- shift **1** left by **2**:
result is **4**, a.k.a. **0b100**
- bitwise OR **0b100** with **ns.bits**:
result is **53**, a.k.a. **0b110101**
- build new set with those bits:
result is **#NaturalSet<[0, 2, 4, 5]>**

ELEMENT BY ELEMENT OPERATIONS

```
19  def put(%NaturalSet{bits: bits}, element) when is_integer(element) and element >= 0 do
20    %NaturalSet{bits: 1 <<< element ||| bits}
21  end
22
23  def member?(%NaturalSet{bits: bits}, element) when is_integer(element) and element >= 0 do
24    (bits >>> element &&& 1) == 1
25  end
26
27  def delete(natural_set, element) when is_integer(element) and element >= 0 do
28    if member?(natural_set, element) do
29      new_bits = (1 <<< element) ^^^ natural_set.bits
30      %NaturalSet{bits: new_bits}
31    else
32      natural_set # return unchanged
33    end
34  end
```

- ||| bitwise OR
- &&& bitwise AND
- ^^^ bitwise XOR
- <<< shift left
- >>> shift right

OPERATIONS ON ENTIRE SETS

```
36  def equal?(%NaturalSet{bits: bits1}, %NaturalSet{bits: bits2}) do
37    bits1 == bits2
38  end
39
40  def intersection(%NaturalSet{bits: bits1}, %NaturalSet{bits: bits2}) do
41    %NaturalSet{bits: bits1 &&& bits2}
42  end
43
44  def union(%NaturalSet{bits: bits1}, %NaturalSet{bits: bits2}) do
45    %NaturalSet{bits: bits1 ||| bits2}
46  end
47
48  def difference(%NaturalSet{bits: bits1}, %NaturalSet{bits: bits2}) do
49    %NaturalSet{bits: bits1 &&& bits2 ^^^ bits1}
50  end
51
52  def subset?(natural_set1, natural_set2) do
53    difference(natural_set1, natural_set2).bits == 0
54  end
55
56  def disjoint?(natural_set1, natural_set2) do
57    intersection(natural_set1, natural_set2).bits == 0
58  end
```

LENGTH: HOW MANY ELEMENTS ARE PRESENT?

The corresponding function in **MapSet** is **size/1**.

However, counting the elements in **NaturalSet** takes O(n) time.

Therefore, by convention, this function must be named **length/1**.

```
60  def length(%NaturalSet{bits: bits}) do
61      count_ones(bits, 0)
62  end
63
64  defp count_ones(0, count), do: count
65
66  defp count_ones(bits, count) do
67      count = count + (bits &&& 1)
68      count_ones(bits >>> 1, count)
69  end
```

PROTOCOLS

Support for polymorphic functions

PROTOCOLS IN ELIXIR 1.10

Predefined protocols

- Collectable
- Enumerable
- Inspect
- Inspect.Algebra
- Inspect.Opts
- List.Chars
- String.Chars

Support for building protocols

- Protocol

The screenshot shows a web browser window titled "Kernel — Elixir v1.10.4". The URL is <https://hexdocs.pm/elixir/Kernel.html>. The page content is as follows:

Kernel

`Kernel` is Elixir's default environment.

It mainly consists of:

- basic language primitives, such as arithmetic operators and others
- macros for control-flow and defining new functions
- guard checks for augmenting pattern matching

You can invoke `Kernel` functions and macros anywhere without prefix since they have all been automatically imported.

```
iex> is_number(13)
true
```

If you don't want to import a function or macro from the `Kernel` module, you can qualify it with its full name and arity:

```
import Kernel, except: [if: 2, unless: 2]
```

See `Kernel.SpecialForms.import/2` for more information.

Elixir also has special forms that are always imported via `Kernel.SpecialForms`.

The standard library

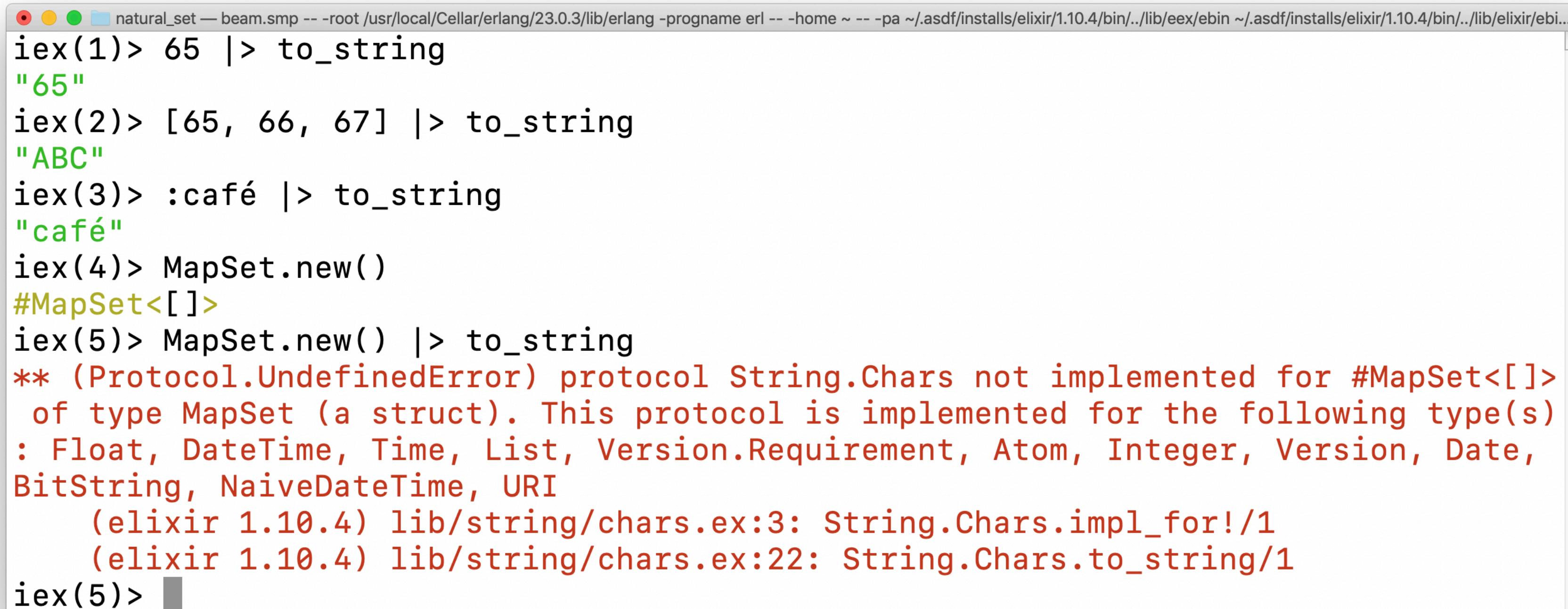
`Kernel` provides the basic capabilities the Elixir standard library needs. You can explore the standard library for advanced functionality.

standard library (this list is not a complete reference)

AN ESSENTIAL PROTOCOL: STRING.CHARS

Protocol **String.Chars** is used by **Kernel.to_string**, **IO.puts** and string interpolation.

The Elixir standard library does not implement **String.Chars** for **MapSet**.



```
iex(1)> 65 |> to_string
"65"
iex(2)> [65, 66, 67] |> to_string
"ABC"
iex(3)> :café |> to_string
"caf "
iex(4)> MapSet.new()
#MapSet<[]>
iex(5)> MapSet.new() |> to_string
** (Protocol.UndefinedError) protocol String.Chars not implemented for #MapSet<[]>
  of type MapSet (a struct). This protocol is implemented for the following type(s)
  : Float, DateTime, Time, List, Version.Requirement, Atom, Integer, Version, Date,
  BitString, NaiveDateTime, URI
    (elixir 1.10.4) lib/string/chars.ex:3: String.Chars.impl_for!/1
    (elixir 1.10.4) lib/string/chars.ex:22: String.Chars.to_string/1
iex(5)>
```

STRING.CHARS PROTOCOL DEFINITION

A protocol is defined by **defprotocol**.

Inside **defprotocol** there are function signatures with no body.

In this example: **to_string(term)**

```
1 import Kernel, except: [to_string: 1]
2
3 defprotocol String.Chars do
4   @moduledoc ~S"""
5   The `String.Chars` protocol is responsible for
6   converting a structure to a binary (only if applicable).
7   """
20
21   @spec to_string(t) :: String.t()
22   def to_string(term)
23 end
```

ANY MODULE CAN IMPLEMENT STRING.CHARS FOR MAPSET

To implement a protocol for a type in a different module, use **defimpl**, for:

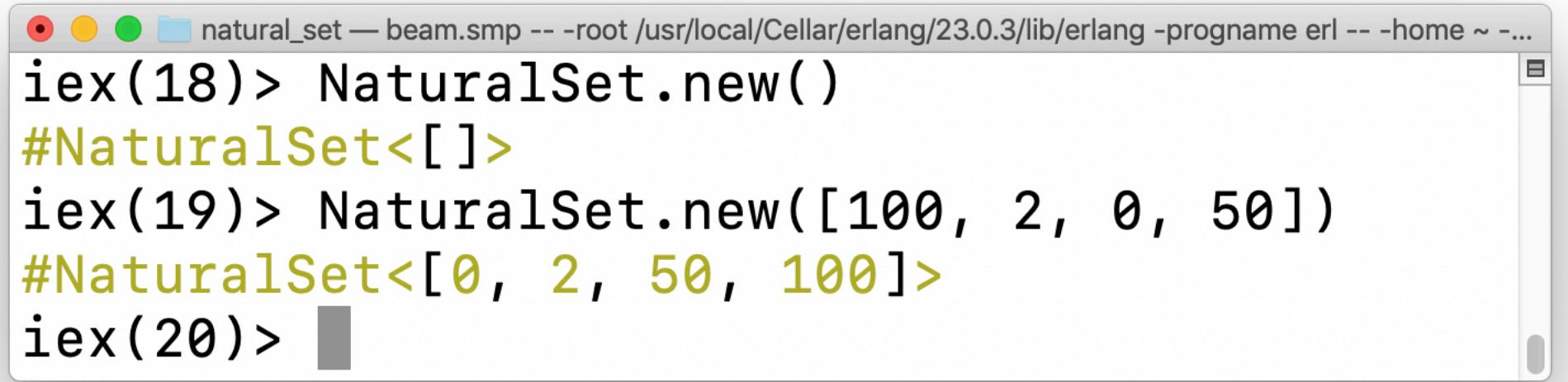
```
1 defmodule ProtocolDemo do
2   @moduledoc """
3     Demonstration: implementing `String.Chars` for `MapSet`.
4
5     iex> MapSet.new([4, 2, 1, 3]) |> to_string
6     "{1 2 3 4}"
7     """
8
9     defimpl String.Chars, for: MapSet do
10       def to_string(map_set) do
11         s = map_set |> Enum.join(" ")
12         "#{#{s}}"
13       end
14     end
15   end
```

NATURAL SET PROTOCOLS

Inspect, Enumerable, and Collectable

INSPECT PROTOCOL USAGE

Inspect supports **Kernel.inspect**, used by iex and doctests.



A screenshot of an iex session window. The title bar says "natural_set — beam.smp -- -root /usr/local/Cellar/erlang/23.0.3/lib/erlang -progrname erl -- -home ~ -...". The iex prompt shows three commands:

```
iex(18)> NaturalSet.new()
#NaturalSet<[]>
iex(19)> NaturalSet.new([100, 2, 0, 50])
#NaturalSet<[0, 2, 50, 100]>
iex(20)>
```

Trying to insert a duplicate is a no-op:

```
iex> natural_set = NaturalSet.new()
#NaturalSet<[]>
iex> natural_set = NaturalSet.put(natural_set, 3)
#NaturalSet<[3]>
iex> natural_set |> NaturalSet.put(2) |> NaturalSet.put(2)
#NaturalSet<[2, 3]>
```

INSPECT PROTOCOL IMPLEMENTATION

To support **Inspect**, implement an **inspect/2** function.

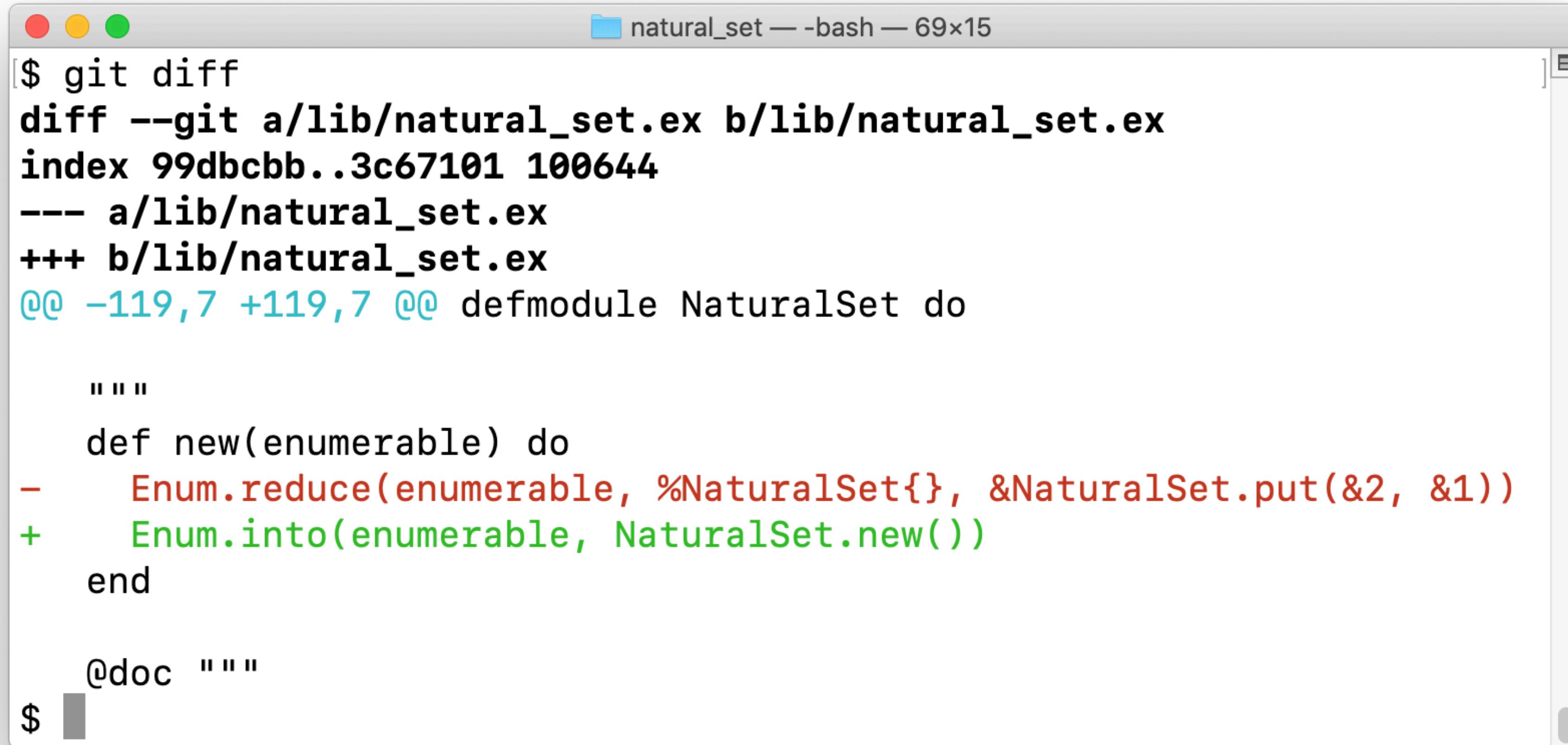
```
defimpl Inspect do
  import Inspect.Algebra

  def inspect(natural_set, opts) do
    concat(["#NaturalSet<", Inspect.List.inspect(NaturalSet.to_list(natural_set), opts), ">"])
  end
end
```

COLLECTABLE PROTOCOL USAGE

Collectable supports the **Enum.into/2** function.

For example, here's the **NaturalSet.new/1** function simplified:



```
$ git diff
diff --git a/lib/natural_set.ex b/lib/natural_set.ex
index 99dbcbb..3c67101 100644
--- a/lib/natural_set.ex
+++ b/lib/natural_set.ex
@@ -119,7 +119,7 @@ defmodule NaturalSet do

    """
    def new(enumerable) do
-      Enum.reduce(enumerable, %NaturalSet{}, &NaturalSet.put(&2, &1))
+      Enum.into(enumerable, NaturalSet.new())
    end

    @doc """
$
```

COLLECTABLE PROTOCOL IMPLEMENTATION

To implement **Collectable**, write an **into/1** function.

I copied this from the **MapSet** implementation.

Only line 112 was changed to call **NaturalSet.put/2**.

```
109  defimpl Collectable do
110    def into(original) do
111      collector_fun = fn
112        set, {:cont, elem} -> NaturalSet.put(set, elem)
113        set, :done -> set
114        _set, :halt -> :ok
115      end
116
117      {original, collector_fun}
118    end
119  end
```

ENUMERABLE PROTOCOL IMPLEMENTATION

Enumerable supports many functions in **Enums** and **Streams**.

Implementation has **count/1**, **member?/2**, **slice/1** and **reduce/3**.

slice/1 would require **size/1**, so this implementation returns an error. This is a convention.

```
91  defimpl Enumerable do
92    def count(natural_set) do
93      {:ok, NaturalSet.length(natural_set)}
94    end
95
96    def member?(natural_set, val) do
97      {:ok, NaturalSet.member?(natural_set, val)}
98    end
99
100   def slice(_set) do
101     {:error, __MODULE__}
102   end
103
104   def reduce(natural_set, acc, fun) do
105     Enumerable.List.reduce(NaturalSet.to_list(natural_set), acc, fun)
106   end
107 end
```

STREAMS 101

Composable and lazy enumerables

STREAMS 101: THE GOOD OLD FIBONACCI EXAMPLE

stream/0 lazily yields Fibonacci numbers *forever*.*

* the size on an Elixir integer is limited only by memory

```
1 defmodule Fibonacci do
2   @doc ~S"""
3     Yields the Fibonacci sequence, limited only by available memory
4
5   ## Example
6
7   iex> Fibonacci.stream() |> Enum.take(10)
8   [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
9   """
10  def stream() do
11    Stream.unfold({0, 1}, fn {a, b} -> {a, {b, a + b}} end)
12  end
```

STREAMS 101: THE GOOD OLD FIBONACCI EXAMPLE

Stream.unfold/2 takes: **accumulator** and **function/1**.

In this example:

- initial **accumulator** is **{0, 1}**: first pair of the sequence, passed to **function/1**.
- function/1** must return: **{number_to_emit, next_accumulator}**
- next_accumulator** is **{next_a, next_b}**

```
1 defmodule Fibonacci do
2   @doc ~S"""
3     Yields the Fibonacci sequence, limited only by available memory
4
5   ## Example
6
7   iex> Fibonacci.stream() |> Enum.take(10)
8   [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
9   """
10  def stream() do
11    Stream.unfold({0, 1}, fn {a, b} -> {a, {b, a + b}} end)
12  end
```

STREAMS 101: A FIBONACCI EXAMPLE THAT STOPS

stream_max/1 lazily yields numbers from the Fibonacci series until the next number **a** is larger than the **max** argument.

Stream.unfold/2 stops when the inner function yields **nil**.

```
30      @doc ~S"""
31      Yields numbers from the Fibonacci sequence up to `max`.
32
33      ## Example
34
35      iex> Fibonacci.stream_max(100) |> Enum.to_list
36      [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
37      """
38
39      def stream_max(max) do
40          Stream.unfold({0, 1}, fn
41              {a, _} when a > max -> nil
42              {a, b} -> {a, {b, a + b}}
43          end)
44      end
```

STREAMING ELEMENTS

Making NaturalSet streamable

STREAM: LAZILY YIELD THE ELEMENTS, ONE BY ONE

Here, **Stream.unfold/2** takes **accumulator** and **next_one/1**:

- **accumulator** is **{bits, index}**, where **index** is the *value* of a (possible) element.
- **next_one/1** returns: **nil** or **{element_to_emit, {next_bits, next_index}}**

```
71  def stream(%NaturalSet{bits: bits}) do
72    Stream.unfold({bits, 0}, &next_one/1)
73  end
74
75  defp next_one({0, _index}), do: nil
76
77  defp next_one({bits, index}) when (bits &&& 1) == 1 do
78    # LSB is one: return {next_element, new_accumulator_tuple}
79    {index, {bits >>> 1, index + 1}}
80  end
81
82  defp next_one({bits, index}) do
83    # LSB is 0: shift bits; increment index; try again
84    next_one({bits >>> 1, index + 1})
85  end
```

TAKE AWAYS

5 ideas to remember

TAKE AWAYS

- If you've never used **MapSet**, I bet you've written a lot of redundant code.

TAKE AWAYS

- If you've never used **MapSet**, I bet you've written a lot of redundant code.
- **MapSet** has a rich API, including powerful operations with whole sets.

TAKE AWAYS

- If you've never used **MapSet**, I bet you've written a lot of redundant code.
- **MapSet** has a rich API, including powerful operations with whole sets.
- Implementing protocols allow custom types to interoperate with core parts of the Elixir standard library: **Kernel**, **Enums**, **Streams**...

TAKE AWAYS

- If you've never used **MapSet**, I bet you've written a lot of redundant code.
- **MapSet** has a rich API, including powerful operations with whole sets.
- Implementing protocols allow custom types to interoperate with core parts of the Elixir standard library: **Kernel**, **Enums**, **Streams**...
- Streaming can be implemented with the help of **Streams.unfold/2** (and other helpers in the **Streams** module).

TAKE AWAYS

- If you've never used **MapSet**, I bet you've written a lot of redundant code.
- **MapSet** has a rich API, including powerful operations with whole sets.
- Implementing protocols allow custom types to interoperate with core parts of the Elixir standard library: **Kernel**, **Enums**, **Streams**...
- Streaming can be implemented with the help of **Streams.unfold/2** (and other helpers in the **Streams** module).
- To learn more, study the code for **MapSet** and **NaturalSet**.

https://hex.pm/packages/natural_set

THANK YOU!

Luciano Ramalho

@ramalhoorg | @standupdev

luciano.ramalho@thoughtworks.com

ThoughtWorks®