Motivation

This design enables several use cases:

- Converting between data structures without intermediate `List` allocations (e.g. `|> Dict.keys |> CustomDataStructure.fromList`)
- No longer needing every data structure to have all the `walk` variants (or else sacrifice some amount of convenience or expressive power)
- Being able to iterate over small strings without having to allocate a `List U8` (or needing things like `Str.walkUtf8BytesUntil` etc.)
- Fusion optimizations to eliminate intermediate allocations and traversals when not changing between data structures
  - e.g. `|> List.map fn1 |> List.map fn2 |> List.keepIf fn3` would optimize into one pass over the input list, instead of allocating and then traversing the intermediate lists
- `for` and `while` loops that can run forward, backward, and on custom (non-builtin) data structures, as well as on non-data-structures like number ranges

Loops

There's a separate WIP document about exactly how loops could work, but without going into much detail, the relevant part of that idea is that iterators can enable things like this:

```
# walk
for elem in List.toIter list do

# walk backwards
for elem in List.toIterRev list do

# walk with index
```

```
# (`for` just gives you the index if you ask for
it)
for index, elem in List.toIter list do

# walk backwards with index
for index, elem in List.toIterRev list do
```

These could also work on ranges, e.g. `Num.range` could return an iterator instead of a `List`, which would avoid a heap allocation.

```
for num in Num.range { start: At 0, end: At 9 }
```

The types of `List.toIter` and `List.toIterRev` would be the same:

```
List.toIter : List elem -> Iter elem
List.toIterRev : List elem -> Iter elem
```

The only difference would be that one would return an Iter that walks the list in reverse.

The `Iter` module

The `Iter` module would expose familiar operations on iterators, e.g.

```
Iter.map : Iter a, (a -> b) -> Iter b
Iter.keepIf : Iter a, (a -> Bool) -> Iter a
Iter.concat : Iter a, Iter a -> Iter a
Iter.andThen : Iter a, (a -> Iter b) -> Iter b
Iter.enumerate : Iter a -> Iter (U64, a)
Iter.mapBoth : Iter a, Iter b, (a, b -> c) -> Iter
c
```

```
Iter.walk : Iter a, state, (state, a -> state) ->
state
```

The fundamental operation is:

```
Iter.next : Iter a -> Result (a, Iter a) [NoMore]
```

This is what `for` can call repeatedly to step through the iterator.

There would also be this:

```
Iter.lenIfKnown : Iter a -> Result U64 [LenUnknown]
```

Iterators over things like lists instantly know how long they are, but others, such as an iterator returned by a parser (e.g. "match as many of these in a row as you can find") might not know in advance what their length will be until they've actually run.

Note about `enumerate`: there's no consensus among languages as to which should go first (the index or the element) in similar situations. For example, JavaScript, C#, and Ruby put the element first, whereas Python, PHP, Kotlin, Go, Swift, and Rust put it second. It seems that more languages put it first, so that's what I went with here.

Individual data structures like `List` and `Dict` could offer both `List.toIter` and `Dict.toIter`, but also `List.fromIter` and `Dict.fromIter`.

I think a good convention would be:

- Every data structure (not just builtins, but custom userspace ones too) exposes `iter`, `iterRev` (if it makes sense), and `fromIter`
    - This makes it possible to use the data structure in loops

    - If every data structure does this, it enables converting between all data structures without intermediate allocations
- Data structures should also offer `toList` and `fromList` as well
    - Although it's not necessary if you can already convert to/from iterators, it would be such a common case that it would always be appreciated
    - In some cases, it would actually be a performance improvement. For example, when converting a large Str to a List U8, you don't want to use an intermediary "iterate over each byte individually" when you could instead be like "here is the contiguous chunk of bytes we already have in memory."
- When making an "iterable view" over something, use iterators over lists if it would avoid unnecessary allocations. For example:
    - Num.range should return an Iter
    - Str.toUtf8 should return an Iter, and Str.walkUtf8Bytes should be removed
        - Str.toUtf8List should be added because wanting specifically a List U8 would still be so common, e.g. for writing to a file or to the network, and it would be more efficient
        - Parsing would use the Iter and writing to a file would use the List

Creating custom iterators

The familiar `Iter.walk` can already be used to convert an iterator into something else. The much less familiar `Iter.custom` would be used to create an iterator from scratch.

The type of `Iter.custom` might look surprising, but the benefits of this design will be explained later. (Also, keep in mind that `Iter.custom` will only be called once per custom data structure, so the fact that it looks strange will only be encountered in practice by custom data structure authors.)

```
Iter.custom :
    # the collection being iterated, or e.g. a
range
    source,
    # the length of the collection, if known up
front
    [Known U64, Unknown],
    # get the next element in the collection
    (source -> Result (elem, Iter elem) [NoMore]),
    -> Iter elem
```

Here's how `Iter.custom` can be used to implement `List.toIter`:

```
List.toIter : List elem -> Iter elem
List.toIter = \list ->
    Iter.custom list (Known (List.len list))
getNext

# Making this a separate function to avoid
accidentally
# closing over the input `list` in the other
function.
# It's important to not close over that, as we'll
see!
getNext = \list ->
    when List.splitFirst list is
        Ok (first, rest) -> Ok (first, List.toIter
rest)
        Err ListWasEmpty -> Err NoMore
```

Implementing `List.fromIter` in terms of `Iter.walk` (which is the same as `List.walk` but with `Iter elem` instead of `List elem`) is more straightforward:

```
List.fromIter : Iter elem -> List elem
List.fromIter = \iter ->
    list =
        Iter.lenIfKnown iter
        |> Result.withDefault 0
        |> List.withCapacity

    Iter.walk iter list List.append
```

The `Iter` module operations (`Iter.map`, `Iter.keepIf`, etc.) would all implemented in terms of what's in `Iter.custom`. They can enable things like this:

```
Dict.toIterKeys dict
|> Iter.keepIf shouldKeepKey
|> List.fromIter
```

This transforms the dictionary's keys into a smaller list of them without allocating and populating an intermediate list along the way like using `Dict.keys |> List.keepIf` would have. This could also have been done using `Dict.walk`, but it's considerably less nice to read:

```
Dict.walk dict
    (List.withCapacity (Dict.len dict))
    \list key _ ->
        if shouldKeepKey key then
            List.append list key
        else
```

```
        list
```

Fusion

This enables an optimization known as stream fusion. Consider this pipeline:

```
list
|> List.map transform1
|> List.map transform2
|> List.keepIf shouldKeep
```

Today, this allocates lots of intermediate lists. However, we can optimize away those intermediate lists if we implement each of these List operations in terms of Iter (behind the scenes) and then apply inlining and stream fusion. For example:

```
List.map = \list, fn ->
    list
    |> List.toIter
    |> Iter.map fn
    |> List.fromIter
```

This might seem like an unnecessarily complicated way to implement `List.map` (although it could be convenient for authors of custom collections), but if inlining and optimizations cause it to end up being compiled to effectively the same implementation we have today, there wouldn't be any loss of runtime performance.

The benefit comes when we start inlining things. Here it is after inlining:

```
List.map = \list, fn ->
    list
```

```
        |> Iter.custom lenIfKnown next
        |> Iter.map fn
        |> Iter.walk (…) List.append
```

Now if we inline that implementation into this…

```
list
|> List.map transform1
|> List.map transform2
|> List.keepIf shouldKeep
```

…we get: (important pairs of calls bolded for emphasis)

```
list
|> Iter.custom lenIfKnown next
|> Iter.map transform1
|> Iter.walk (…) List.append
|> Iter.custom lenIfKnown next
|> Iter.map transform2
|> Iter.walk (…) List.append
|> Iter.custom lenIfKnown next
|> Iter.keepIf shouldKeep
|> Iter.walk (…) List.append
```

These pairs can cancel out; each of them is walking through an entire iterator only to immediately turn around and use its output to create another iterator of the same type. We can delete both of these pairs and keep the iterator (of the same element type) that we already had, since we know all the functions involved are pure. All that could possibly be noticeably missing would be calls to `dbg` or `crash`.

After deleting those intermediate calls, we end up with:

```
list
|> Iter.custom lenIfKnown next
|> Iter.map transform1
|> Iter.map transform2
|> Iter.keepIf shouldKeep
|> Iter.walk (…) List.append
```

…which is the inlined version of what we'd have written if we had explicitly converted to an iterator in the first place:

```
list
|> List.toIter
|> Iter.map transform1
|> Iter.map transform2
|> Iter.keepIf shouldKeep
|> List.fromIter
```

…except that we instead got to write this nicer version, while having the intermediate lists optimized away behind the scenes:

```
list
|> List.map transform1
|> List.map transform2
|> List.keepIf shouldKeep
```

The key to this is the strange-looking `Iter.custom` signature. Here it is again, this time along with its actual implementation:

```
Iter.custom :
    # the sequence being iterated, or e.g. a range
    source,
```

```
        # the number of elements in the sequence, if
    known
        # (it may be infinite or otherwise unknown)
        [Known U64, Unknown],
        # get the next element in the collection
        (source -> Result (elem, Iter elem) [NoMore])
        -> Iter elem
    Iter.custom = \source, lenIfKnown, next -> @Iter {
        lenIfKnown,
        # create a thunk which captures source and next
        # (we can't store them directly without
        # adding a second type parameter to Iter)
        next: \{} ->
            when next source is
                # translate Result into [One …, Done,
    Skip …]
                # (this difference will come up later)
                Ok (elem, iter) -> One elem iter
                Err NoMore -> Done
    }

    Iter elem := {
        lenIfKnown : [Known U64, Unknown],
        next : ({} -> [One elem (Iter elem), Skip U64,
    Done])
    }
```

The type of `Iter.custom` could be simpler if it closed over the source collection instead of taking it as a separate argument. For example, it could have been this:

```
    Iter.custom :
```

```
        [Known U64, Unknown],
        ({} -> Result (elem, Iter elem) [NoMore]),
        -> Iter elem
```

…and then it could have been called like this: (this example uses Unknown for the length because it makes it easier to see the problem, although the problem would still be there if it had been `(Known (List.len list))` instead)

```
List.toIter : List elem -> Iter elem
List.toIter = \list ->
    Iter.custom Unknown \{} ->
        when List.splitFirst list is
            Ok (first, rest) ->
                Ok (first, List.toIter rest)
            Err ListWasEmpty -> Err NoMore
```

Here's the problem: in order to perform the fusion, we need to detect this pattern of calls:

```
Iter.custom (Iter.walk …) …)
```

In the proposed design, where the first argument to `Iter.custom` is the collection, this is very straightforward to detect! In this design, on the other hand, the thunk hides where the collection actually is, making it absurdly difficult to detect the pattern of nested calls we want to eliminate. You'd have to scour the body of the thunk passed to `Iter.custom` and hope that the argument appeared in some recognizable location.

So this type signature, although more complicated for the (very rare) collection author who is actually calling Iter.custom in order to implement an iterator for a custom collection, facilitates the stream fusion optimization in a way that would probably be infeasible with the simpler type.

Big-Picture Tradeoffs

There are, of course, some downsides to this overall design:

1. It relies on inlining. If there is no inlining, this optimization absolutely will not happen. That also means if you pull something out into a named variable, whether or not the optimization will apply will depend on whether that variable got inlined into the call site.
2. It makes implementations more complex under the hood, which will affect compile times by some amount. (Probably not by much, perhaps not even noticeable, but there would definitely be an overall increase in compiler work done.)
3. Some optimizations might become harder (or unfeasible) compared to not doing this, e.g. making List.map do everything in-place if it's passed a unique list which has elements of the same size as the desired list—we'd have to be careful on the timing of an optimization like that, because if we tried to apply it *after* having done this list fusion, we'd already have inlined the List.map into an Iter.map and would no longer be able to recognize that the optimization was available.
4. It relies on it being okay to eliminate pure functions that should be no-ops semantically. This means that if there's a dbg, expect, or crash in there, we arguably should not eliminate it because those are observable side effects. However, I think we should eliminate them anyway because our general rule should be that pure functions are always fair game to be optimized away (you should never rely on dbg for program correctness, an expect that never fails will never be heard from anyway, and the compiler should consider avoiding a crash a positive optimization!) and once we're doing that sort of thing, the *absence* of an expected dbg, expect, or crash can actually confirm for you whether something is actually getting optimized away in practice.
5. There are various ways someone can supply an incorrect function to `Iter.custom` (e.g. crashing or returning an iterator over something

closed over that isn't the passed-in source collection) in which eliminating `(Iter.custom (Iter.walk …) …)` pairs would change program behavior. This seems fine; in practice it seems like you'd have to go out of your way to break the rules (which of course the docs for `Iter.custom` should explain, including how they will be used for optimization), plus you're handing a function to an opaque API which is making no guarantees about when and how it might call that function.

The stream fusion paper talks about how to implement these `Iter` operations. For example, here's `Iter.map`:

```
Iter.map : Iter a, (a -> b) -> Iter b
Iter.map = \@Iter { source, next, lenIfKnown }, fn
->
    Iter.custom source lenIfKnown \newSource ->
        when next source is
            Done -> Done
            Skip n -> Skip n
            One elem iter -> One (fn elem) iter
```

The `Skip` variant is important to make certain operations (such as `keepIf`) efficient.

These iteration functions aren't the most straightforward-looking Roc functions, but they don't need to be. They'll be implemented as builtins.

Some examples of `Dict.toIter`, `Dict.toIterKeys`, and `Dict.toIterValues`:

```
Dict.toIter : Dict k v -> Iter (k, v)
Dict.toIter = \dict ->
    # Dict.toList is zero-cost because Dict stores
```

```
    # an actual List (k, v) to remember insertion
order
    Dict.toList dict
    |> List.toIter

Dict.toIterKeys : Dict k v -> Iter k
Dict.toIterKeys = \dict ->
    Dict.toIter dict
    |> Iter.map \(k, _) -> k

Dict.toIterValues : Dict k v -> Iter v
Dict.toIterValues = \dict ->
    Dict.toIter dict
    |> Iter.map \(_, v) -> v
```

These are actually the most efficient possible implementations of these operations!

A naive call to `Dict.keys` (which perhaps should be renamed to `Dict.keyList`, if not outright eliminated, so that `Dict.keys` could return the `Iter`) would actually create a whole List, whereas using Dict.toList (which is zero-cost) as the starting point for the iterator—even when subsequently mapping away either the keys or values—is the fastest way to do it. This design means you don't have to know about that! It's enough for the implementor of key iteration and value iteration to know it, and to provide efficient implementations of those key and/or value iterators based on that knowledge.

In summary, here is what `Iter` means:

- Collections will generally expose not only toList and fromList, but also `iter` and `fromIter`. Possibly also `iterRev`.
- It's no longer necessary for collections that expose iterators to expose all the `walk` variants (or any `walk` variants, since `Iter` can offer those).

- Doing transformations while converting between one collection type and another no longer requires intermediate allocations
- Doing several operations on a single collection (e.g. multiple maps, map + keepIf, etc.) will now typically no longer result in intermediate allocations (unless inlining doesn't apply or something)
    - This can result in some dbgs, expects, and/or crashes not running as expected, which is by design
- `for` loops can Just Work on lots of collections, as well as ranges.
- Num.range can return Iter instead of List and still have a nice API without allocating.
- Str.toUtf8 can return an Iter U8 instead of List U8 and avoid an allocation when working on small strings. There's no longer a need for Str.walkUtf8
- Most APIs that currently involve List still do. For example, file I/O and HTTP requests still use List U8 because the underlying primitives (e.g. OS syscalls) generally want a contiguous array of bytes, not stepping through one byte at a time.