Lecture 7 — The Collect Operation

Parallel and Sequential Data Structures and Algorithms, 15-210 (Qatar-Spring 2014)

Lectured by Kemal Oflazer — 2 February 2014

Material in this lecture:

- The collect operation.
- Set ADT

1 Review

Sections 1 and 2 of these notes are a repeat and expanded review of the material in Lecture Notes 7. We suggest that you read these sections to review the material.

Abstract Data Types. An ADT specifies a set of operations to be performed on a data type. It is an abstraction because it hides the detail of how a data type is implemented. You often don't want to know all the details of how a data type is implemented just to use one. This turns out to be one of the most powerful tools in computer science because it allows us solve problems that would otherwise take too much time.

Abstraction is not unique to computer science it is everywhere. For example, you might not want to spend your life baking cookies (implementing priority queues) but you do want to be able to enjoy one (use one when solving the shortest path problem on graphs).

There are many ways to specify a data type. Some languages are terrible at it. But ML is pretty good, because it allows you to specify the meaning of a data type by using types and does the hard work of checking that an implementation matches that specification (at the level of types).

We may specify a vehicle ADT in SML as follows.

Example 1.1.
```
(* The signature for a basic vehicle.
   The vehicle can be started and stopped.
   When it stops, it returns the average speed travelled
   for the last trip and the cost per mile.
 *)
signature Vehicle
sig
  type t
  val start: t -> t
  val stop: t -> t*real*real
end
```

†Lecture notes by Umut Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan, with additional edits by Kemal Oflazer

Question 1.2. *Can you think of a implementations of the* `Vehicle` *signature.*

A vehicle signature can be implemented by a bike, by a car, by a truck, by a train, by an airplane. We can further differentiate between each kind, a Ford automobile, a Toyota automobile, a Specialized bike, a Bianci bike, a Ducati motorbike, etc.

Question 1.3. *Can you think of enriching the signature?*

The signature is very basic at this time. A vehicle usually has a more complex interface. For example, it can change direction, it can carry things, so we can add functionality for all these things.

Question 1.4. *Can you specify such a rich interface by using just types?*

For specifying such rich behavior types often don't suffice. For example you may want to know about the turning radius of your vehicle, how many pounds it can carry, etc.

Question 1.5. *Have you noticed the language that we used for specifying abstract data types?*

To specify ADT's, we will often use mathematics, typically set notation. You may want to go back and look at your notes from lecture 6 to see an example.

Question 1.6. *What do you care about when trying to decide on the kind of vehicle that you would like to purchase?*

Typically you would consider many things such as the functionality, how big it should be, how much luggage room it should have. But there is one thing that nearly all of us would also take into account: the performance and the cost. If you have to travel 30 miles to work or school, you may not want to purchase a bike. If you live one-mile away from work, you probably would not want to buy and maintain a Ferrari.

In fact, when purchasing a vehicle you are often not interested in the detail- of how it is implemented per se. How many of you would care about about the construction of the turbo charger in a Ford Focus or the Shimano Derailleur in the Specialized bike that you are considering to purchase?

In this class, we will think about data types in a similar way. In addition to the specification of their meaning by using a set notation, we will also give them a *cost specifications*.

Question 1.7. *In vehicles, how is cost expressed? How would you want to express it when specifying the cost of an abstract data type.*

In designing and using abstract data types, we often specify, work, span, and sometimes space. We use the familiar big-Oh, big-Omega, the Theta notations when specifying the cost.

Sequences with Scan and Reduce. In addition to specifying data types you have also talked about sequence ADT and two key operations: scan and reduce.

Question 1.8. *Can you recall the type specification of scan and reduce?*

Their type signatures follow.

```
scan: ('a * 'a -> 'a) -> 'a -> 'a seq -> ('a seq * a)
reduce: ('a * 'a -> 'a) -> 'a -> 'a seq -> a
```

The key point to remember here is that for `scan`, we assume an associative operation, the first argument.

In addition, we have also talked about their semantics. Scan returns you all the prefix sums of the sequence and reduce returns you the whole prefix sum. The term sum here means of course sum with respect to the associative operation specified.

To understand these operations, it is very important to understand the notion of associativity.

Definition 1.9. A binary operation $\oplus$ defined for a set S is associative if for any $a, b, c \in S$, $a \oplus (b \oplus c) = (a \oplus b) \oplus c$.

Question 1.10. *Can you think of a non-associative operation over integers?*

Perhaps the most basic associative binary operation is minus, e.g., $10-(5-1) \neq (10-5)-1$.

Question 1.11. *Why did we assume the operation to be associative when defining scan?*

Associativity allows to compute a sum such as $s_0 \oplus s_1 \oplus s_2 \ldots s_n$ in any order. By that we don't mean that you can exchange the places of the s_i's but that you can apply $\oplus$ operations in any order that you like.

The implementation of scan carefully takes advantage of this.

Exercise 1. *Convince yourself that you understand how the implementation of scan takes advantage of the associativity of the binary operation.*

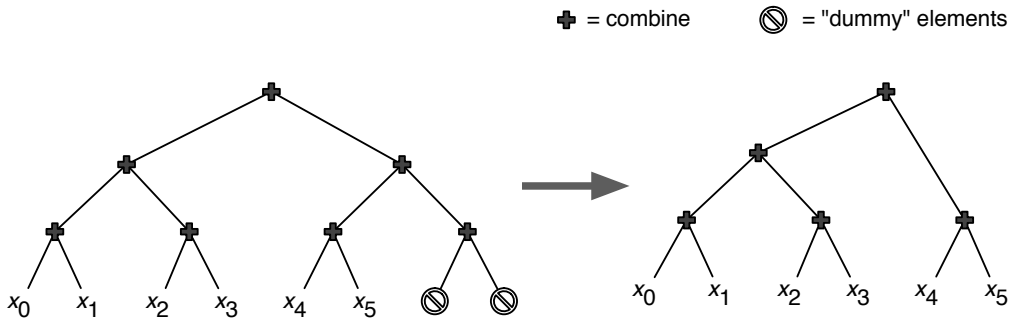
When specifying reduce, we didn't want to assume associativity because it can be too strong.

Question 1.12. *How can we specify the semantics of* `reduce` *without assuming associativity?*

If we don't assume associativity, then we have no choice but specify the order in which the binary operations are performed.

Question 1.13. *Do you remember how* `reduce` *was specified so that the order of operations would be clear?*

We specified reduce by extending the sequence to the nearest power of 2 by adding dummy elements at the end and then fitting a perfect reduce tree (where every node except for the leaves) has exactly two children on the sequence and then eliminating the empty leaves and nodes with one or more dummy elements.



Question 1.14. *What if the operations that we are reducing with is actually associative?*

If we are reducing with an associative operation then as far as the output of the reduce is concerned we don't have to think about the order.

2 Analyzing the Costs of Higher-Order Functions

In the `reduce` examples thus far, we often assumed that the cost of the binary operation is always $O(1)$. This is not always the case. In fact in the last lecture, we talked about how we can use `reduce` to implement many divide-and-conquer algorithms on sequences. For many interesting applications there, the cost of the binary operation would not be constant.

In general, when working with higher order functions, we have to pay attention to the cost of the functions passed as argument.

Question 2.1. *Can you specify the cost for the* `map` *function on sequences?*

For `map` it is easy to find its costs base on the cost of the function applied:

$$\begin{aligned} W(\texttt{map}\ f\ S) &= 1 + \sum_{s \in S} W(f(s)) \\ S(\texttt{map}\ f\ S) &= 1 + \max_{s \in S} S(f(s)) \end{aligned}$$

Tabulate is similar.

Question 2.2. *Can we specify the cost of* `reduce` *similarly?*

The cost of reduce often crucially depends on the order in which we apply the binary operation. As an example, consider the append operation on strings.

Question 2.3. *Is the append operation on strings associative?*

We don't need it to be so for using with `reduce` but append in an associative operation.

Question 2.4. *Consider $a \oplus b \oplus c$ where $\oplus$ is append and a, b, c are strings. What is the cost of this operation in terms of the lengths of a, b, c?*

The work actually depends on the order in which we perform these operations. The cost of $(a \oplus b) \oplus c$ is $(n_a + n_b) + ((n_a + n_b) + n_c)$. The cost of $a \oplus (b \oplus c)$ is $(n_a + (n_b + n_c)) + (n_b + n_c)$.

Let's consider another operation on sequences: that of merging two sorted sequences.

Question 2.5. *Is merging associative?*

Like `append`, `merge` is an associative operation.

We can use `merge` to sort a sequence. To this end, we specialize it to use the $<$ operation, written $\texttt{merge}_<$.

Assuming a constant work comparison function, two sequences S_1 and S_2 with lengths n_1 and n_2 can be merged with the following costs:

$$\begin{aligned} W(\texttt{merge}_<(S_1, S_2)) &= O(n_1 + n_2) \\ S(\texttt{merge}_<(S_1, S_2)) &= O(\log(n_1 + n_2)) \end{aligned}$$

Question 2.6. *Let's consider the reduction order we get by sequentially merging the elements of a sequence in one after the other. On input $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle$, the sequence of* $\texttt{merge}_<$ *calls looks like the following:*

$$\texttt{merge}_<(\ldots\ \texttt{merge}_<(\texttt{merge}_<(\texttt{merge}_<(I, \langle x_0 \rangle), \langle x_1 \rangle), \langle x_2 \rangle), \ldots)$$

i.e. we first merge I and $\langle x_0 \rangle$, then merge in $\langle x_1 \rangle$, then $\langle x_2 \rangle$, etc.
What is the work of this sequence of merge operations?

With this order $\texttt{merge}_<$ is called when its left argument is a sequence of varying size between 1 and $n-1$, while its right argument is always a singleton sequence. The final merge combines $(n-1)$-element with 1-element sequences, the second to last merge combines $(n-2)$-element with

1-element sequences, so on so forth. Therefore, the total work for an input sequence S of length n is

$$\sum_{i=1}^{n-1} c \cdot (1+i) \in O(n^2)$$

since merge on sequences of lengths n_1 and n_2 has $O(n_1 + n_2)$ work.

Note that this reduction order is the order that the `iter` function uses, and hence is equivalent to:

```
fun iterSort(S) =
  iter merge< (empty()) (map singleton S)
```

Furthermore, using this reduction order, the algorithm is effectively working from the front to the rear, "inserting" each element into a sorted prefix where it is placed at the correct location to maintain the sorted order. This corresponds to the well-known insertion sort.

Question 2.7. *Is there parallelism in* `iterSort`*?*

Recall that merge on sequences of lengths n_1 and n_2 has $O(\log(n_1+n_2))$ span. Since the reduction order has no parallelism except within each merge, the span is

$$S(\texttt{iterSort}\ x) \ \leq \ \sum_{i=1}^{n-1} c \cdot \log(1+i) \in O(n \log n)$$

Question 2.8. *Can you see what algorithm* `iterSort` *implements?*

It implements the insertion sort algorithm.

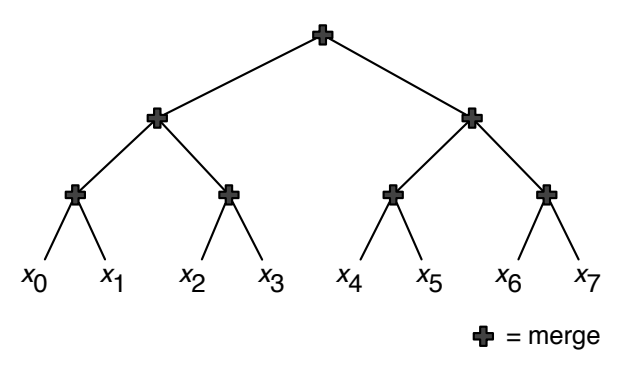
Question 2.9. *Can you think of a way to improve our bound by using a different reduction order with the merge operation?*

In `iterSort`, the reduction tree is unbalanced. We can improve the cost by using a balanced tree instead.

For ease of exposition, let's suppose that the length of our sequence is a power of 2, i.e., $|x| = 2^k$. Now we lay on top the input sequence a perfect binary tree[1] with 2^k leaves and merge according to the tree structure.

[1] This is simply a binary tree in which every node either has exactly 2 children or is a leaf, and all leaves are at the same depth.

Example 2.10. *As an example, the merge sequence for* $|x| = 2^3$ *is shown below.*



What would the cost be if we use a perfect tree?

At the bottom level where the leaves are, there are $n = |x|$ nodes with constant cost each.

Stepping up one level, there are $n/2$ nodes, each corresponding to a `merge` call, each costing $c(1+1)$. In general, at level i (with $i = 0$ at the root), we have 2^i nodes where each node is a merge with input two sequences of length $n/2^{i+1}$.

Therefore, the work of such a balanced tree of $\texttt{merge}_<$'s is the familiar sum

$$\begin{aligned} &\leq \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^{i+1}} + \frac{n}{2^{i+1}}\right) \\ &= \sum_{i=0}^{\log n} 2^i \cdot c\left(\frac{n}{2^i}\right) \end{aligned}$$

This sum, as you have seen before, evaluates to $O(n \log n)$.

Merge Sort. In fact, this algorithm is essentially the merge sort algorithm. We can use our reduction technique for implementing divide-and-conquer algorithms to implement merge sort with a `reduce`.

In particular, we can write a version of merge sort, which we refer to as reduceSort, as follows:

```
combine = merge_<
base = singleton
emptyVal = empty()
fun reduceSort(S) = reduce combine emptyVal (map base S)
```

where $\texttt{merge}_<$ is a merge function that uses an (abstract) comparison operator $<$.

Exercise 2. *Implement* `iterSort` *with* `reduce`.

Summary 2.11. *A brief summary of a few points.*

- *When applying a binary operation, if the operation is associative, the order of applications doesn't matter to the final result. The* `scan` *operation takes advantage of this in its implementation.*
- *When applying a binary operation, then the order of applications matters when calculating the cost (work and span), regardless of whether the operation is associative or not.*
- *In* `scan` *we assume that the operation is associative.*
- *In* `reduce` *we don't assume that the operation is assaciative. We avoid this assumption by specifying carefully the order in which the operation is applied.*
- *We have been able to implement merge sort and insertion sort with reduce.*

The cost of reduce in in general. In general, how would we go about defining the cost of `reduce` with higher order functions. Given a reduction tree, we'll first define $\mathcal{R}(\texttt{reduce}\ f\ \mathbb{I}\ S)$ as

$$\mathcal{R}(\texttt{reduce}\ f\ \mathbb{I}\ S) = \left\{\text{all function applications } f(a,b) \text{ in the reduction tree}\right\}.$$

Following this definition, we can state the cost of `reduce` as follows:

$$W(\texttt{reduce}\ f\ \mathbb{I}\ S) = O\left(n + \sum_{f(a,b)\in\mathcal{R}(f\ \mathbb{I}\ S)} W(f(a,b))\right)$$
$$S(\texttt{reduce}\ f\ \mathbb{I}\ S) = O\left(\log n \max_{f(a,b)\in\mathcal{R}(f\ \mathbb{I}\ S)} S(f(a,b))\right)$$

The work bound is simply the total work performed, which we obtain by summing across all combine operations. The span bound is more interesting. The $\log n$ term expresses the fact that the tree is at most $O(\log n)$ deep. Since each node in the tree has span at most $\max_{f(a,b)} S(f(a,b)$, any root-to-leaf path, including the "critical path," has at most $O(\log n \max_{f(a,b)} S(f(a,b))$ span.

This can be used, for example, to prove the following lemma:

Lemma 2.12. *For any combine function $f : \alpha \times \alpha \to \alpha$ and a monotone size measure $s : \alpha \to \mathbb{R}_+$, if for any x, y,*

1. $s(f(x,y)) \le s(x) + s(y)$ *and*
2. $W(f(x,y)) \le c_f\,(s(x)+s(y))$ *for some universal constant c_f depending on the function f,*

then

$$W(\texttt{reduce}\ f\ \mathbb{I}\ S) = O\left(\log|S| \sum_{x\in S}(1+s(x))\right).$$

Applying this lemma to the merge sort example, we have

$$W(\texttt{reduce}\ \texttt{merge}_<\ \langle\,\rangle\ \ \langle\langle a\rangle : a \in A\rangle) = O(|A|\log|A|)$$

3 Collect

Thus far we considered two very important functions on sequences, `scan` and `reduce`.

We now look a third function: `collect`.

Specification of Collect. Let's start with something that you may have heard of.

Question 3.1. *Do you know of key-value stores?*

The term key-value store often refers to a storage systems (which may in on disk or in-memory) that stores pairs of the form "key x value."

Question 3.2. *Can you think of a way of representing a key-value store using a data type that we know?*

We can use a sequence to represent such a store.

Example 3.3. *For example, we may have a sequence of key-value pairs consisting of our students from last semester and the classes they take.*

$$\begin{aligned}\texttt{Data} = \langle &(\text{“jack sprat”}, \text{“15-210”}),\\ &(\text{“jack sprat”}, \text{“15-213”}),\\ &(\text{“mary contrary”}, \text{“15-210”}),\\ &(\text{“mary contrary”}, \text{“15-213”}),\\ &(\text{“mary contrary”}, \text{“15-251”}),\\ &(\text{“peter piper”}, \text{“15-150”}),\\ &(\text{“peter piper”}, \text{“15-251”}),\\ &\ldots\rangle\end{aligned}$$

Note that key-value pairs are intentionally asymmetric: they map a key to a value. This is fine because that is how we often like them to be.

But sometimes, we often want to put together all the values for a given key.

We refer to this operations as a *collect*.

Example 3.4. *We can determine the classes taken by each student.*

$$\begin{aligned}\texttt{classes} = \langle &(\text{“jack sprat”}, \langle \text{“15-210”}, \text{“15-213”}, \ldots\rangle)\\ &(\text{“mary contrary”}, \langle \text{“15-210”}, \text{“15-213”}, \text{“15-251”}, \ldots\rangle)\\ &(\text{“peter piper”}, \langle \text{“15-210”}, \text{“15-251”}, \ldots\rangle)\\ &\ldots\rangle\end{aligned}$$

Collecting values together based on a key is very common in processing databases. In relational

database languages such as SQL it is referred to as "Group by". More generally it has many applications and furthermore it is naturally parallel.

We will use the function `collect` for this purpose, and it is part of the sequence library. Its interface is:

$$\texttt{collect} : (\alpha \times \alpha \to \mathit{order}) \to (\alpha \times \beta)\ \mathit{seq} \to (\alpha \times \beta\ \mathit{seq})\ \mathit{seq}$$

1. $\alpha \times \alpha \to \mathit{order}$ is a function for comparing keys of type α
2. $(\alpha \times \beta)\ \mathit{seq}$ is a sequence of key-value pairs
3. $(\alpha \times \beta\ \mathit{seq})\ \mathit{seq}$ is the resulting sequence of each unique α value paired with a sequence of all β values it appears with.

The `collect` function collects all values that share the same key together into a sequence, ordering the values in the same order as their appearance in the original sequence.

Example 3.5. *Given sequence of pairs each consisting of a student's name and a course they are taking, we want to collect all entries by course number so we have a list of everyone taking each course. This would give us the roster for each class, which can be viewed as another sequence of key-value pairs. For example,*

$$\begin{aligned}\texttt{rosters} = \langle &(\text{“15-150”}, \langle \text{“peter piper”}, \ldots \rangle)\\ &(\text{“15-210”}, \langle \text{“jack sprat”, “mary contrary”}, \ldots \rangle)\\ &(\text{“15-213”}, \langle \text{“jack sprat”}, \ldots \rangle)\\ &(\text{“15-251”}, \langle \text{“mary contrary”, “peter piper”} \rangle)\\ &\ldots \rangle\end{aligned}$$

Question 3.6. *Can you see how to create a roster?*

Example 3.7. *If we wanted to collect the entries of* `Data` *given above by course number to create a roster, we could do the following:*

$$\texttt{collectStrings} = \texttt{collect String.compare}$$

$$\texttt{rosters} = \texttt{collectStrings}(\langle (c,n) : (n,c) \in \texttt{Data} \rangle)$$

This would give something like:

$$\begin{aligned}\texttt{rosters} = \langle &(\text{“15-150”}, \langle \text{“peter piper”}, \ldots \rangle) \\ &(\text{“15-210”}, \langle \text{“jack sprat”}, \text{“mary contrary”}, \ldots \rangle) \\ &(\text{“15-213”}, \langle \text{“jack sprat”}, \ldots \rangle) \\ &(\text{“15-251”}, \langle \text{“mary contrary”}, \text{“peter piper”} \rangle) \\ &\ldots \rangle\end{aligned}$$

We use a map ($\langle (c,n) : (n,c) \in \texttt{Data} \rangle$) *to put the course number in the first position in the tuple since that is the position used to collect on.*

Cost of Collect. Collect can be implemented by sorting the keys based on the given comparison function, and then partitioning the resulting sequence. In particular, the sort will move the pairs so that all equal keys are adjacent.

A partition function can then identify the positions where the keys change values, and pull out all pairs between each change. Doing this partitioning can be done relatively easily by filtering out the indices where the value changes.

The dominant cost of `collect` is therefore the cost of the sort.

Assuming the comparison has complexity bounded above by W_c work and S_c span, then the costs of `collect` are $O(W_c n \log n)$ work and $O(S_c \log^2 n)$ span.

Exercise 3. *Complete the details of this implementation. Better yet, implement collect on your own.*

Question 3.8. *Can you think of another way to implement collect?*

It is also possible to implement a version of collect that runs in linear work using hashing. But hashing would require that a hash function is also supplied and would not return the keys in sorted order. Later we discuss tables which also have a `collect` function. However tables are specialized to the key type and therefore neither a comparison nor a hash function need to be passed as arguments.

4 Using Collect in Map Reduce

Some of you have probably heard of the map-reduce paradigm first developed by Google for programming certain data intensive parallel tasks. It is now widely used within Google as well as by many others to process large data sets on large clusters of machines—sometimes up to tens of thousands of machines in large data centers. The map-reduce paradigm is often used to analyze various statistical data over very large collections of documents, or over large log files that track the activity at web sites. Outside Google the most widely used implementation is the Apache Hadoop implementation, which has a free license (you can install it at home). The paradigm is different from the `mapReduce` function you might have seen in 15-150 which just involved a map then a reduce. The map-reduce paradigm actually involves a map followed by a collect followed by a bunch of reduces, and therefore might be better called the map-collect-reduces.

The map-reduce paradigm processes a collection of documents based on a map function $f_{\mathtt{m}}$ and a reduce function $f_{\mathtt{r}}$ supplied by the user. The $f_{\mathtt{m}}$ function must take a document as input and generate a sequence of key-value pairs as output. This function is mapped over all the documents. All key-value pairs across all documents are then collected based on the key. Finally the $f_{\mathtt{r}}$ function is applied to each of the keys along with its sequence of associated values to reduce to a single value.

In ML the types for map function $f_{\mathtt{m}}$ and reduce function $f_{\mathtt{r}}$ are the following:

$$\begin{aligned} f_{\mathtt{m}} &: (\mathit{document} \to (\mathit{key} \times \alpha)\ \mathit{seq}) \\ f_{\mathtt{r}} &: (\mathit{key} \times (\alpha\ \mathit{seq}) \to \beta) \end{aligned}$$

In most implementations of map-reduce the document is a string (just the contents of a file) and the key is also a string. Typically the α and β types are limited to certain types. Also, in most implementations both the $f_{\mathtt{m}}$ and $f_{\mathtt{r}}$ functions are sequential functions. Parallelism comes about since the $f_{\mathtt{m}}$ function is mapped over the documents in parallel, and the $f_{\mathtt{r}}$ function is mapped over the keys with associated values in parallel.

In ML map reduce can be implemented as follows

```
1  fun mapCollectReduce fm fr docs =
2    let
3        pairs = flatten ⟨ fm(s) : s ∈ docs ⟩
4        groups = collect String.compare pairs
5    in
6      ⟨ fr(g) : g ∈ groups ⟩
7    end
```

The function `flatten` simply flattens a nested sequence into a flat sequence, e.g.:

```
   flatten ⟨⟨a, b, c⟩, ⟨d, e⟩⟩
⇒  ⟨a, b, c, d, e⟩
```

As an example application of the paradigm, suppose we have a collection of documents, and we want to know how often every word appears across all documents. This can be done with the following $f_{\mathtt{m}}$ and $f_{\mathtt{r}}$ functions.

fun $f_{\mathtt{m}}(\mathtt{doc}) = \langle (w, 1) : \mathtt{tokens\ doc} \rangle$

fun $f_{\mathtt{r}}(w, s) = (w, \mathtt{reduce\ +\ 0\ } s)$

Here `tokens` is a function that takes a string and breaks it into tokens by removing whitespace and returning a sequence of strings between whitespace.

Now we can apply `mapCollectReduce` to generate a `countWords` function, and apply this to an example case.

val `countWords` = `mapCollectReduce` $f_{\mathtt{m}}$ $f_{\mathtt{r}}$

`countWords` $\langle$*"this is a document"*,
"this is is another document",
"a last document"$\rangle$

$\Rightarrow \langle$(*"a"*, 2), (*"another"*, 1), (*"document"*, 3), (*"is"*, 3), (*"last"*, 1), (*"this"*, 2)$\rangle$