Bites of Lists - Mapping and Filtering Sublists

Kurt Nørmark
Department of Computer Science
Aalborg University
Denmark
normark@cs.aau.dk

ABSTRACT

The idea of applying map and filter functions on consecutive sublists instead of on individual list elements is discussed and developed in this paper. A non-empty, consecutive sublist is called a bite. Both map and filter functions accept a function parameter - a bite function - which is responsible for returning a prefix bite of a list. We develop families of bite functions via a collection of higher-order bite generators. On top of the bite generators, a number of bite mapping and bite filtering functions are introduced. We illustrate the usefulness of bite mapping and filtering via examples drawn from a functional programming library that processes music, represented as Standard MIDI Files.

Categories and Subject Descriptors

D.1.1 [**Applicative (Functional) Programming**]: Lisp, Scheme; E.1 [**Data structures**]: Lists; H.5.5 [**Sound and Music Computing**]: Systems

1. INTRODUCTION

Mapping and filtering represent some of the classical higher-order functions on lists, together with similar functions such as reduction functions. Both the classical mapping and filtering functions deal with individual elements of a list. A mapping function applies a function to each individual element of the list, and it returns the list of these applications. A filter function selects those elements on which a predicate is fulfilled. The predicate of a filtering function is also applied on each individual element of the list.

The idea described in this paper is to apply mapping and filtering on sublists of a list. The Common Lisp function `maplist`, which applies a given function on successive tails of a list, is a simple example of a mapping function that belongs to this genre. In this context, a sublist of a list L= $(e_1\ e_2\ ...\ e_n)$ is a non-empty consecutive part of L, $(e_i\ ...\ e_j)$, where $i \leq j$, $i \geq 1$, and $j \leq n$. It is easy to see that for a list L of length n, there are $(n+1)(n/2)$ such sublists.

Although it is possible, and maybe even useful, to map and filter all possible sublists of a list, most of the work in this paper will deal with map and filter functions that process mutually disjoint sublists that partition the list. In this context, a disjoint partitioning of a list L = $(e_1\ e_2\ ...\ e_n)$ is formed by $L_1 = (e_{i_1}\ ...\ e_{j_1})$, $L_2 = (e_{i_2}\ ...\ e_{j_2})$, ..., $L_k = (e_{i_k}\ ...\ e_{j_k})$ where $k \leq n$, $i_1 = 1$, $j_k = n$, $j_m = i_{m+1}$ for $1 \leq m \leq k-1$. In order to simplify the vocabulary, each non-empty sublist in a disjoint partitioning will be called a *bite*. The bites $L_1\ ...\ L_k$ of a list L = $(e_1\ e_2\ ...\ e_n)$ are all non-empty, and when they are appended the result is L.

The development on *bites of lists* has been motivated by our previous work on MIDI music programming in Scheme [9]. A piece of music, represented as a MIDI file, consists of a list of discrete MIDI events. The MIDI list of a typical song consists of thousands of such events. When a list of MIDI events is captured from a MIDI instrument, the first job is typically to *impose structure* on the list. The structure will be a music-related division consisting of units, such as bars/measures, song lines, song verses, or chord sequences. The capturing of music related structures in a list of MIDI events was the starting point of the work described in this paper.

In addition to the music-related application area, we will also mention an example of another area in which mapping and filtering of sublists may be useful. Imagine a long list of time-stamped meteorological data objects that describes the weather conditions during a long period of time. Each object may contain information about temperature, air pressure, rain since last reading, and other similar data. In order to extract characteristics about weather conditions at a more coarse-grained level, it may be relevant to partition the list in sublists. These sublists may, for instance, correspond to regular periods of time (days, weeks, month, or years). Sublists with monotone progressions of the air pressure or temperature could also be of interest. Both kinds of sublists can be produced by the functions described in this paper. Systematic search for (maybe overlapping) sublists with certain more detailed properties is also an area which is supported by functions in this work.

In Section 3 we present the higher-order mapping and filtering functions, each of which rely on a bite function. When applied to a list, a bite function returns a prefix of the list. Prior to the discussion of the mapping and filtering functions, we will in Section 2 introduce a collection of *bite*

function generators. It turns out that these generated "biting functions" are the crucial part of the game. In Section 4 we discuss the bite generators relative to the motivating example of this work (introducing structure in a list of MIDI events). Section 5 contains a description of related work. Section 6 presents our conclusions.

The mapping and filtering of sublists has been developed in the context of the R5RS Scheme programming language [5]. Even though Scheme is dynamically typed, we will often describe the functions by means of statically typed signatures.

2. BITE GENERATORS

A bite function `b` is a function which takes a list of elements as parameter, and returns a non-empty prefix of that list. More precisely, a bite function returns a non-empty list prefix when applied on a non-empty list. Thus, the signature of a bite function `b` is List<E> → List<E> for some element type E. Of convenience, and for generalization purposes, a bite function return the empty list when applied on an empty list.

It is usually straightforward to program a *particular* bite function. In this section we will deal with families of similar bite functions, as generated by higher-order *bite generating functions.*

A particularly simple bite generator is `(bite-of-length n)`, which returns a function that takes a bite of length `n`:

```
((bite-of-length 3) '(a b c d e)) => (a b c)
```

If a function, returned by `(bite-of-length n)` is applied on a list with fewer than `n` elements it just returns that list.

Another simple bite generator `bite-while-element` is controlled by an element predicate passed as parameter to the function. When applied on a list, the generated bite function returns the longest prefix of the list whose elements, individually, satisfy the element predicate. Thus, for instance,

```
((bite-while-element even?) '(2 6 7 4)) => (2 6)
```

Elements which violate the element predicate are called *sentinels.* The requirement that successive bites of a list append-accumulate to the original list makes it necessary, one way or another, to include the sentinel elements in the list. Each bite includes at most one sentinel. Sentinels can either start a bite, terminate a bite, or occur alone as singleton bites. These variations are controlled by an optional keyword parameter[1], `sentinel`, of the bite generator. The default value of `sentinel` is `"last"`. However, the example given above assumes that `sentinel` is `"first"`.

---

[1]Keyword parameters are simulated in a way that corresponds to how LAML [8] handles XML attributes in Scheme functions. Following the conventions of LAML, an attribute name is a symbol and the attribute value must belong to another type (typically a string). In the context of the function `bite-while-element`, this explains why the sentinel role is a string (and not a symbol).
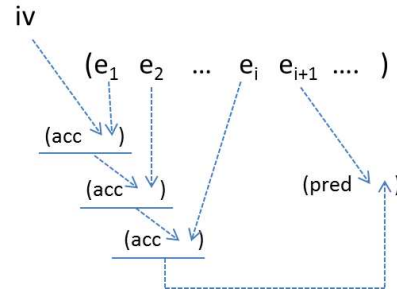

Figure 1: *In the list $(e_1 \dots e_i\ e_{i+1} \dots)$ the elements $e_1, \dots, e_i$ have been accumulated with use of the function `acc` and the initial value `iv`. The current element e_{i+1} is passed to the predicate `pred` together with the accumulated value.*

The following examples illustrate the role of the `sentinel` parameter:

```
((bite-while-element even? 'sentinel "first")
 '(1 2 6 7 4)) => (1 2 6)

((bite-while-element even? 'sentinel "alone")
 '(1 2 6 7 4)) => (1)

((bite-while-element even? 'sentinel "last")
 '(1 2 6 7 4))  => (1)

((bite-while-element even? 'sentinel "last")
 '(2 6 7 4)) => (2 6 7)
```

The remaining bite generators construct bites based on properties that do not (alone) pertain to individual elements. Functions generated by the expression

```
(bite-while-element-with-accumulation
    pred accumulator init-val)
```

accumulate the elements of the bite. The accumulated value and the 'the current list element' are handed to a predicate `pred`, which controls the extent of the bite. For the sake of the accumulation, a binary accumulator `acc` is needed together with an initial 'getting started value' `iv`. This is illustrated in Figure 1. The generated function returns the longest bite in which each element, together with the accumulation of the "previous elements", fulfill the predicate.

Here follows an example that accumulates elements in an integer list by simple `+` accumulation (using 0 as the initial value). In the example, the predicate states that we are interested in the longest prefix of the list that, successively, has a sum of 5 or below.

```
((bite-while-element-with-accumulation
    (lambda (e s)  (<= (+ e s) 5))
    (lambda (s e) (+ s e))
    0)
 '(1 2 1 1 -1 2 3 4))  => (1 2 1 1 -1)
```

As above, we assume in the general case that the element type of the list is E. The predicate (which is the first parameter shown above) has the signature E × S → bool. The accumulator (the second parameter) has the signature S × E → S. In the predicate, the S-valued parameter is the accumulation of all values before the E-valued 'current element'.

A function generated by `bite-while-element-with-accumulation` always 'consumes' the first element in the list without passing it to the predicate. As a consequence, the first element in the bite does not necessarily fulfill the predicate. Without this special case, bite functions generated by `bite-while-element-with-accumulation`, will be able to return empty bites. Recall from Section 1 that empty bites are illegal (unless taken from an empty list). Successive "biting" with bite a function generated by `bite-while-element-with-accumulation` is illustrated in Section 3.

A function generated by `(bite-while-compare er)` returns the longest bite where elements, pair-wise, fulfill a binary element relation defined by the function `er`. Here is an example where we identify the longest increasing prefix of a list:

```
((bite-while-compare <=)
 '(2 6 6 7 1 2))  => (2 6 6 7)
```

It comes out naturally that bite functions, generated by use of `bite-while-compare`, return non-empty bites when applied on non-empty input. It should be noticed that the effect of `bite-while-compare` can be achieved by a (rather clumsy) application of `bite-while-element-with-accumulation` with an accumulator that just returns the previous elements.

The last bite generation function that we will discuss in this section is `bite-while-monotone`, which can be seen as a convenient generalization of `bite-while-compare`. Based on an element comparator, which follows the C conventions of comparison functions[2], a function generated by `bite-while-monotone` returns the longest monotone bite of the list. More precise, the function returns the longest list prefix where successive pairs of elements have the same value when passed to the comparator function. Here is an example:

```
((bite-while-monotone (make-comparator < >))
 '(1 2 3 2 1 0 4 4 4 1 2 1))
  => (1 2 3)
```

The element comparator is constructed by `make-comparator`, which as input receives the *greater than* and the *less than* functions. Five "successive bitings" with the function in the example produces the bites (1 2 3), (2 1 0), (4 4 4), (1 2), and (1) respectively. Such "successive biting" can easily be realized with use of `map-bites` which will be introduced in the following section.

As explained above, all generated bite functions have the signature List<E> → List<E> for some element type E.

_____
[2]In the scope of this paper (`compare-to x y`) is -1 if `x` is less than `y`, 0 if `x` is equal to `y`, and 1 if `x` is greater than `y`.

In some situations it is useful to know where a given bite belongs relative to neighboring bites. For this reason, all generated bite functions accept a second integer parameter that informs the bite function about the *current bite number*, in contexts where bites are generated successively (as introduced in Section 1). In Scheme, this is handled by requiring that all generated bite functions have a rest parameter, like in `(lambda (lst . rest) ...)`, where the first element in `rest` will be bound to the current bite number. Examples are provided when we discuss the bite mapping and bite filtering functions in Section 3.

Finally, most bite generators accept an optional predicate, called a *noise predicate*. Elements that fulfill the noise predicate are passed unconditionally to the resulting bite. Noise elements are not counted (in the context of `bite-of-length`), are not taken into consideration by the predicate of `bite-while-element`, and are not accumulated by `bite-while-element-with-accumulation`. In Section 4 we will see practical examples that reveal the usefulness of noise predicates.

3. BITE MAPPING AND BITE FILTERING
We will now discuss a number of higher-order functions that successively applies a bite function to a list, and which processes the resulting bites in various ways. As already mentioned, some bite functions can be generated by one of the functions described in Section 2. More specialized bite functions will have to be explicitly programmed relative to the specific needs.

3.1 The map-bites function
The function `map-bites` is the natural counterpart to the classic `map` function in both Scheme and Common Lisp. `map-bites` applies a *bite transformation function* on each bite of a list (relative to repeated application of a given bite function):

```
(map-bites bite-function bite-transf lst)
```

If applied on a bite of type List<E>, the transformation function `bite-transf` is supposed to return another list of type List<F>. The lists produced by the bite transformation function are *spliced* (`append` accumulated) by `map-bites`. In that way (`map-bites bf id lst`), where `id` is the identity function, is equal to `lst` for any bite function `bf`.

Let us first use the Scheme function `list` as the bite transformation function, with the purpose of explicitly identifying the bites successively delivered by a given biting function. This particular transformation illustrate the typical need of somehow revealing/representing the individual bites in the results returned by the mapping or filtering functions.

```
(map-bites (bite-while-element number?) list
  '(1 -1 1 a 3 4 b 6 1 2 3)) =>
((1 -1 1 a) (3 4 b) (6 1 2 3))
```

The bites taken by `(bite-while-element number?)` places a sentinel value as the last element of the bite (because, as explained in Section 2, the default value of the optional

`sentinel` parameter is `"last"`). In the example, a sentinel element is an element which is not a number. If the intention is to get rid of sentinel elements after applying the bite function, it may be better to isolate them using the `"alone"` sentinel option:

```
(map-bites
  (bite-while-element number? 'sentinel "alone")
   list
   '(1 -1 1 a b c 3 4 b 6 1 2 3)) =>
((1 -1 1) (a) (b) (c) (3 4) (b) (6 1 2 3))
```

In this example it is straightforward to get rid of singular, non-numeric elements from this list by ordinary (element) filtering.

Let us also illustrate `map-bites` relative to other bite functions, as produced by the generators discussed in Section 2.

```
(define sum-at-most-5
  (bite-while-element-with-accumulation
   (lambda (e v)  (<= (+ e v) 5))
   (lambda (v e) (+ v e))
   0))

(map-bites sum-at-most-5 list
   '(1 2 1 1 -1 7 -1 3 1 4))  =>
((1 2 1 1 -1) (7) (-1 3 1) (4))

(define increasing
   (bite-while-compare <=))

(map-bites increasing list
   '(2 6 6 7 5 1 -3 1 8 9)) =>
((2 6 6 7) (5) (1) (-3 1 8 9))

(define monotone-ints
   (bite-while-monotone
     (make-comparator <= >=)))

(map-bites monotone-ints list
   '(2 6 6 7 5 1 -3 1 8 9))  =>
((2 6 6 7) (5 1 -3) (1 8 9))
```

3.2   The bite filtering functions
Bite filtering, as provided by the function `filter-bites`, has the following parameter profile:

```
(filter-bites bite-function bite-predicate lst)
```

After generation of each bite with use of `bite-function` the bite is passed to a `bite-predicate`, which decides if the bite should be part of the output list. The bites accepted by the bite predicate are spliced together, in the same way as in `map-bites`. The non-accepted bites are discarded. The following example, which works on bites of length 3 (whenever possible), filters the bites that start with a number:

```
(filter-bites
   (bite-of-length 3)
   (lambda (bite) (number? (car bite)))
   '(1 -1 1 a b c 3 4 b 6 1 2 3))  =>
(1 -1 1 3 4 b 6 1 2 3)
```

The following bites are taken out of the sample list: `(1 -1 1)`, `(a b c)`, `(3 4 b)`, `(6 1 2)`, and `(3)`. The predicate only discards `(a b c)`, because the first element of this bite is not a number. The remaining bites are spliced together and returned by `filter-bites`.

It is often convenient to apply a bite transformation function `bite-transf` just after filtering with `bite-predicate`:

```
(filter-map-bites bite-function bite-predicate
                  bite-transf lst)
```

This function first chunks the list `lst` with use of `bite-function`. Each resulting bite is passed to `bite-predicate`, and the accepted bites are transformed by `bite-transf` (to a value which must be a list). The lists returned by the bite transformations are finally spliced together.

In simple cases we can (just as illustrated for `map-bites` above) use the bite transformation function `list` for identification of the bites that have survived the filtering.

```
(filter-map-bites
  (bite-of-length 3)
  (lambda (bite) (number? (car bite)))
  list
  '(1 -1 1 a b c 3 4 b 6 1 2 3))  =>
((1 -1 1) (3 4 b) (6 1 2) (3))
```

The result of this filtering is similar to the previous example, but the bite structure is preserved in the output.

The implementations of `map-bites` and `filter-bites` are simple and straightforward. The function `map-bites` is implemented as a tail recursive function that collects the transformed bites in a list, which is reversed and `append`-accumulated as the very last step. The function `filter-bites` is implemented in a similar way.

In general, the bite functions are implemented by means of tail recursive functions which collect the bite elements in a parameter, which is reversed before a bite is returned. Because a bite is prefix of a list, say of length n, we need to allocate n new cons-cells for it. (This would not have been necessary if we worked on suffixes of a list, like the Common Lisp function `maplist`). It is crucial for our approach that the bites are materialized as separate lists, but it is also quite expensive to allocate (and deallocate) memory for these intermediate structures.

3.3   The step-and-map-bites function
We will now discuss `step-and-map-bites` which is a slightly more complex variant of `filter-map-bites`. The parameter lists of `step-and-map-bites` and `filter-map-bites` are

basically the same. `filter-bites` and `filter-map-bites` both discard a whole bite b, if b is not accepted by the bite predicate. Let us assume that the length of a bite b is n. It b is accepted by the bite predicate in `step-and-map-bites` it is transformed just like it is done by `filter-map-bites`, and we step n elements forward from the beginning of the current bite before we take the next bite of the list. If b is not accepted by `bite-predicate`, we do not discard n elements when using `step-and-map-bites`. Instead, the predicate gives back a step length s (typically 1), and the next bite taken into consideration starts s elements ahead. Elements stepped over are not discarded (as in filtering functions). Such elements are passed directly and untransformed to the output.

In the setup just described, the value returned by the bite predicate has two purposes: (1) The accepting purpose (a boolean view of the value) and (2) the stepping length purpose (an integer view of the value). Therefore the bite predicate of `step-and-map-bites` returns an integer. A positive integer p is considered as an accepting value, and p is the stepping length. The number p is typically (but not necessarily) the length of the most recent bite. A negative integer n is a non-accepting value, and $-n$ is the stepping length. In most practical cases n is -1.

The execution of the functions `map-bites`, `filter-bites`, and `filter-map-bites` are linear in the length of the input list, if the bite functions, together with the other function parameters, are linear in their processing of the prefixes of the list. In contrast, `step-and-map-bites` is able to regress the "biting process", hereby processing the same elements of the input list several times.

As another difference, `map-bites` and `filter-bites` process disjoint "bite" partitionings of a list, as described in Section 1. In contrast, and as discussed above, `step-and-map-bites` processes selected disjoint[3] bites that may be separated by list elements, which are unaffected by the mapping process. We may consider these "in between elements" as intermediate bites. Using this interpretation, the processing done by `step-and-map-bites` can be thought of as operating on disjoint bites (those selected together with the intermediate bites) that append-accumulate to the original list.

In the following example, the biting function takes bites of length 3 out of a list of integers. A bite is accepted if the sum of the elements of the bite is even. If the bite is not accepted, we take a single step forward, and recurses from there.

```
(step-and-map-bites
  (bite-of-length 3)
  (lambda (bite) (if (even? (apply + bite))
                     (length bite)
                     -1))
  list
```

[3]Disjointness is only assured if the stepping length of the integer-valued bite predicate returns a positive number which is not less than the length of the accepted bite. The last part of Section 3.3 shows an example where the stepping length is 1. This leads to processing of overlapping bites.

```
  '(0 1 2 1 2 3 4 0 -2 1 3 4 5))  =>
(0 (1 2 1) 2 3 (4 0 -2) (1 3 4) 5)
```

The first bite of length 3 is (0 1 2), and its element sum is odd. The stepping mechanism causes `step-and-map-bites` to output the element 0, and consider the next bite (1 2 1). The element sum of (1 2 1) is even, and it is accepted and transformed (with the function `list`). We therefore step 3 elements forward. The following (overlapping) bites (2 3 4) and (3 4 0) are not accepted, because their element sums are non-even. The elements 2 and 3 are consequently transferred the output list. The next bite of length 3, which is (4 0 -2), is accepted, etc.

If the "predicate" of `step-and-map-bites` (the second parameter) returns a positive integer smaller than the length of the bite, overlapping bites will be processed. The following variant of the expression shown above returns all possible consecutive triples of a list with even element sum:

```
(filter list?
  (step-and-map-bites
    (bite-of-length 3)
    (lambda (bite) (if (even? (apply + bite))
                       1    ; <-- The difference
                       -1))
    list
    '(0 1 2 1 2 3 4 0 -2 1 3 4 5)))  =>
((1 2 1) (1 2 3) (4 0 -2) (-2 1 3)
 (1 3 4) (3 4 5))
```

At the outer level of the expression we disregard all non-list elements with use of the ordinary element `filter` function.

4. MAPPING AND FILTERING BITES OF MIDI SEQUENCES

As already mentioned in Section 1, the development on *bites of lists* was motivated by our work on MIDI programming in Scheme. In this section we discuss a number of MIDI programming problems and solutions facilitated by bite mapping and filtering. First, however, we give some background on our approach to MIDI programming in Scheme.

MIDI is a protocol for exchange of music-related events. The MIDI protocol emphasizes sequences of discrete music events (such as `NoteOn` and `NoteOff` events) in contrast to an audio representation of the music. A piece of music can be represented as a *Standard MIDI file* in a compact binary representation. A Standard MIDI file[4] is basically a long sequence of MIDI events. We have developed a Scheme representation of a Standard MIDI file, which relies on LAML[5] for representation of MIDI sequences. We call it MIDI LAML [9].

In MIDI LAML we work on long lists of MIDI events (typically several thousands events). The main goal of our system

[4]A Standard MIDI file of format 0 is a sequence of MIDI events. Format 1 and format 2 midi files are structured in tracks and songs respectively.
[5]LAML [8] represents our approach and suite of tools to deal with XML documents in Scheme.
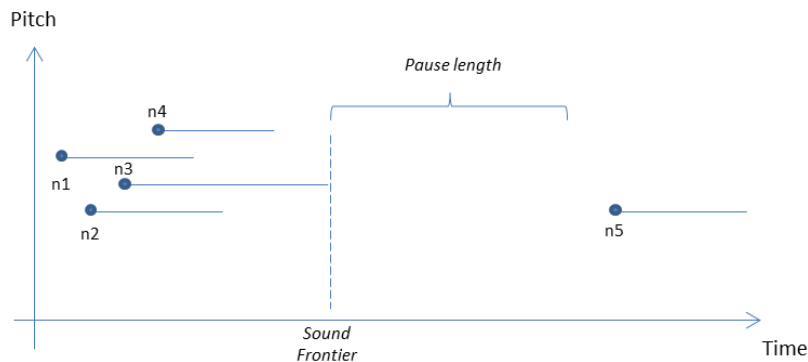
Figure 2: *A piano roll presentation of a few notes n1 .. n5 and their (horizontal) durations. The bite formed by the notes n1 .. n4 is followed by a pause, because n5 starts after the absolute time* Sound Frontier + Pause Length.

is to do useful work on music via functionally *programmed solutions* - in contrast to *interactive operations* in a MIDI sequencer environment. Instead of always working on individual MIDI events it is often attractive to work on selected, consecutive sublists of MIDI sequences. Such sublists are bites of MIDI events.

When a list of MIDI events is captured from a MIDI instrument, the first job is typically to *impose structure* on the list. The structure will consist of music-related divisions of the MIDI list. Many such structures can be captured by application of bite mapping. We will now describe how it can be done by use of the functions from Section 2 and 3. It is recommended that the reader consults the detailed program listings in the appendix while reading the subsections below.

4.1 Bars

For temporally strict music[6] the bar/measure structure can be captured by mapping a *next-bar bite function* over the music using `map-bites`. A relevant bite function can be generated by means of either `bite-while-element-with-accumulation` (for delta timed MIDI sequences) or `bite-while-element` (for absolutely timed MIDI sequences). A simple transformation of `map-bites` is to insert a bar division meta message into the stream of notes in order to emphasize the bar structure of the music. The function `map-bars`, as outlined in Appendix A.1, wraps these pieces together.

As a more interesting application, it is possible via the transformation function of `map-bites` to affect the characteristics of selected bars, for instance the tempo, the velocity (playing strength), or the left/right panning. In the last part of Appendix A.1 we show how to gradually slide the tempo of every fourth bar of a song with use of `map-bars`, which is programmed on top of `map-bites`.

4.2 Pauses

A song is often composed by parts separated by pauses. A *pause* is a period of time *pl* during which no `NoteOn` messages

---

[6]Music played by metronome, or music captured from a source which quantizes the start and duration of notes to common note lengths, is here called *temporally strict music*.

appear. In addition, we will require that all previously activated notes have ended before entering the pause of length *pl*.

It is obviously useful to identify pauses in a song, because it will provide a natural top-level structure in many kinds of music. We provide a function called `map-paused-sections` which maps some function `f` on sections of MIDI message that are separated by pauses. This function is implemented in terms of `map-bite`, which in turn uses a bite function generated by `bite-while-element-with-accumulation`. Please consult Appendix A.2 where the implementation of `map-paused-sections` is presented.

The use of the generated bite function is illustrated in Figure 2. Let us assume that we look for pauses of at least *pl* time ticks. The accumulator keeps track of a point in time called the *sound frontier sf* where all previous notes have been ended. The predicate examines if the next note starts at a time after *sf + pl*. If this is the case a pause has been identified, and this ends the current bite of the MIDI sequence.

The actual implementation of `map-paused-sections`, as it is shown in Appendix A.2, uses a variant of `map-bites` called `map-n-bites`, which passes the bite-number to the transformation function. With use of this variation, it is easy to insert numbered markers into the MIDI sequence.

4.3 Sustain intervals

When playing a piano, one of the pedals is used to hold the notes after they have been released on the keyboard. This is called *sustain*. In a MIDI representation, sustain is dealt with by particular `ControlChange` messages that represent the level of the pedal. We are interested in identifying the monotone sustain regions, such as R1, ..., R7 shown Figure 3. A single bite of the MIDI sequence can be identified with a function generated by `bite-while-monotone` from Section 2. When such a bite function is mapped over the entire sequence of MIDI messages with `map-bites`, we can conveniently process the regions shown in Figure 3.

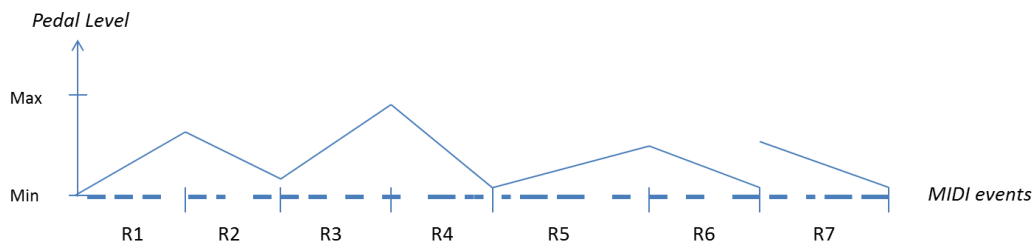We have programmed a function on top of these applica-

Figure 3: *Seven regions of MIDI events that represent 'pedal down" and 'pedal up' intervals. Region 6 and 7 are both pedal down intervals. The pedal moved quickly from a low value to a high value. It is not a requirement that monotone bites alternate between being increasing and decreasing.*

tions called `map-sustain-intervals`, which passes knowledge about the monotonicity to the transformation function of `map-bites`. (Please consult Appendix A.3 for details). This can, for instance, be used for fastening the downward pedal movements (in the regions R2, R4, R6, and R7 of Figure 3), such that sustained notes rings out steeper or earlier than in the original. Without use of a generated bite function, and without use of `map-bites`, it would be a fairly difficult task to program such a transformation (let alone the effort of realizing the change interactively in a conventional MIDI sequencer).

4.4 Chord identification

As the final example from the domain of MIDI music we will discuss how to identify the chords in a piece music. Chord recognition [12] is much more difficult to handle than the other music related examples we have discussed above. In the context of the work described in this paper, it is not the ambition to come up with a high-quality chord recognition algorithm. Rather, our goal is to find out which chords can be identified based on rather straightforward use of the functions described in Section 2 and 3.

A chord is, in a formulation due to Wikipedia, a set of three or more notes that is heard as if sounding simultaneously [17]. The notes in a chord are not necessarily initiated at the exact same time. At top-level, chords are identified by a function `map-chords`, which is shown in Appendix A.4. This is a higher-order function which applies a given function on every sublist identified as a chord. Internally, `map-chords` calls `step-and-map-bites` which is one of the functions we described in Section 3. We use `step-and-map-bites` together with a bite function generated with `bite-while--element-with-accumulation`. This bite function takes a bite of notes, where each note is temporally relatively close to the previous note. The predicate of `step-and-map-bites` (the second parameter) asserts if the relevant notes of the bite fulfill a chord formula. If it is not the case, the stepping mechanism of `step-and-map-bites` is activated. This implies that a new bite of temporally close notes is taken, and so on. Please take a look in Appendix A.4 for additional details.

4.5 Noise elements

In all but the first example in this section, it is useful to zoom in on certain MIDI events, and to be able to disre-

gard all other events. This is possible by use of the so-called *noise predicate* mentioned briefly in Section 2. The noise predicate can be applied on elements of the list, from which bites are taken. Elements that satisfy the noise predicate are disregarded while identifying a bite (in predicates, comparison, accumulation, etc.), but noise elements appear in the resulting bite.

In the function that locates pauses, `map-paused-sections`, which is shown in Appendix A.2, all non-`NoteOn` events (such as instrument selection, sustain and tempo changes) are considered as noise. In addition, `map-paused-sections` relies on a *relevance function* (third parameter) which at a detailed level points out the MIDI events which should be taken into account when looking for pauses. This may, for instance, be all notes in a particular channel above a certain note value (pitch value). The negation of the *relevance function* is used as noise function of the bite function "under the hood". This separation of concern turns out to be very useful: The filtering of relevant messages takes place in the generation of the bite function (`bite-while-element-with-accumulati-on`), totally separated from the logic that deals with the rules for pauses in the music.

In the function that identifies sustain intervals, all non-sustain messages (such as `NoteOn` messages) are noise elements. In the chord identification function only `NoteOn` events in a given channel are relevant. All other messages are considered as noise.

4.6 Discussion

As mentioned in the introduction to Section 4, the primary use of bite functions in a music related context is to *identify structures* in the music. Some structures, such as the use of *tracks*, are already manifest in the representation of some Standard MIDI files. The bar structuring can also appear explicitly in the MIDI LAML representation of Standard MIDI files.

Some structures are very difficult to identify automatically. Although we may attempt to capture song lines and song verses via use of bite functions, it is our experience that it is not always realistic to accomplish this with success. Therefore, the MIDI LAML environment contains a number of facilities for manual introduction of additional structures. This includes manual insertions of MIDI markers (meta events), and systematic transformation of events in

a dedicated channel to MIDI markers. See the paper about MIDI programming in Scheme [9] for additional details.

The programming technique explored in this paper relies, to large extent, on higher-order function that receives and generates functions. As a typical scenario, a bite mapping function receives both a bite function *bf* and a bite transformation function *btf*. *bf* may be generated by one of the bite generator functions from Section 2 on the basis of several other functions, such as a list element predicate, an accumulator, and a noise function.

It turns out to be a typical situation that the information established by one function, such as the bite function *bf*, also is needed by another function, such as the bite transformation function *btf*. As a concrete example, the direction of the pedal in Section 4.3 and Appendix A.3 is identified in the bite function, but it is also needed in bite transformation function. It would clutter everything if we attempted to pass this "additional information" as the function result together with the "main function result", for instance with use of multiple valued functions. In pure functional programming, passing the information via an output parameter is not feasible either. We choose to re-calculate the information in the bite transformation function - as the least evil way out of the problem. It is possible to abstract the duplicated parts to a common lambda expressions at an outer scope level, but this solution does not necessarily lower the complexity of the program. As an alternative, it could be tempting to let the bite function fuse the needed information into the bite, hereby transferring it to the bite transformation function. This solution requires that it is possible to associate extra information with the bites.

5. RELATED WORK

In this section we will discuss existing work which is related to our work on bites of lists.

The idea of capturing recursive patterns using "Functions with Functions as Arguments" first appeared in John McCarthy's seminal paper about recursive functions and symbolic expressions from 1960 [7]. In this paper, the `maplist` function (as mentioned in Section 1) appears together with a linear search function. It soon became clear that a large class of list-related problems can be solved by a few applications of `map` and `filter`, typically followed by some reduction. During fifty years, the use of mapping and filtering functions together with reduction functions have played a role in almost any textbook about Lisp, Scheme, and other functional languages.

Common Lisp supports functions on *sequences* [15]. A Common Lisp sequence is a generalization of lists and one-dimensional arrays. Many of the Common Lisp sequence functions are higher-order functions. Some functional arguments are passed as required parameters, others are passed as keyword parameters. The processing of successive bites, as proposed in this paper, can be handled by use of the `:start` and `:end` keyword parameters in many of the sequence functions. The `:start` and `:end` keyword parameters delimit a sublist which subsequently can be processed in various ways (removed, substituted). By use of these keyword parameters a Common Lisp programmer can do simple bite processing. The

Common Lisp sequence functions operate at the level of list elements. Only element testing and element transformation is provided for. In contrast, the bite mapping and filtering functions in this paper work on sublists as such. The higher level of abstraction in the bite-related functions may be convenient and powerful in some contexts, but it is also quite expensive. The cost comes primarily from copying prefixes of a list, in order to form the bites.

There exists a variant of Common Lisp sequences called *series* (see appendix A of [15]). In this work the main focus is on automatic transformation of series to efficient iterative looping constructs [16]. In our current work it would be interesting and useful to consider similar techniques for obtaining more efficient mapping and filtering of bites. A number of functions in the series package are oriented towards splitting of a list into one or more sublists (`split`, `split-if`, `subseries`, `chunk`). As such, it is plausible that some of the bite processing programs discussed in this paper can be converted to use functions from the series library.

R5RS Scheme [5] is quite minimalistic with respect to list supporting functions. The repertoire of list functions in the R6RS Scheme standard libraries [14] is more comprehensive. In relation to R5RS, additional list functions are supported by SRFIs, most notable the SRFI 1 List Library [13]. This library supports the `drop` and `take` functions. The expression `(take lst i)` returns the first `i` elements of `lst`, and it corresponds to the `((bite-of-length i) lst)`. The SRFI 1 function `take-while` correspond to the function `bite-while-element`, as described in Section 2.

Modern object-oriented programming languages handle a variety of different collection types via so-called *iterators*. An iterator is a mutable object that manages the traversal of a collection. The LINQ framework (well described in [1]) of C# [2] is a good example of a system which handles the processing of data collections via use of iterators. There exists a number of LINQ query operators that are related to sub-collections (`Take`, `TakeWhile`, `Skip`, `SkipWhile`, `GroupBy`). It remains to be seen to which degree the existing query operators can be used directly for the purposes that are discussed in this paper. If this is not the case, it should be noticed that it is easy to define new, specialized query operators (as extension methods in static classes) that carry out specialized operations of collections.

The concept of *bites*, as introduced in this paper, is known as *slices* in other contexts. Some programming languages have special notation for slicing. A good example is Python [6] which generalizes the classical subscripting notation `seq[i]` to slicing notation `seq[i:j]`. In this expression both `i` and `j` are optional. Therefore, the Python expression `seq[:j]`, which extracts the first `j` elements of a sequence object, corresponds to `((bite-of-length j) seq)` using the bite generator `bite-of-length` from Section 2 of this paper.

As much more advanced notation, known as *list comprehension*, is supported in many programming languages (such as in Haskell [4] or Python [6]). List comprehension, which is inspired by conventional mathematical set building notation, is a syntactic abstraction over applications of mapping and filtering functions. Therefore, list comprehension may
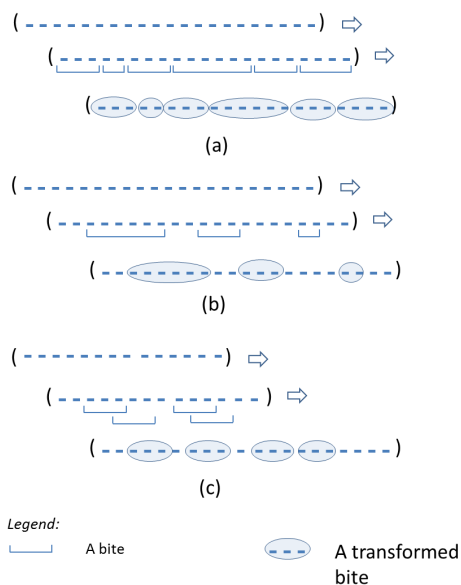
Figure 4: *Three different situations of bite taking and bite transformation.*

be used as an alternative to explicit mapping and filtering when we wish to process sublists of list.

During the last few years, the words "map" and "reduce" have been used to characterize a particular kind of parallel processing of very large collections of data. MapReduce [3], as used by Google, works on key/value pairs, and (in part) because of that, its relation to the original work on mapping, filtering and reduction is relatively weak. In the scope of this paper, it may be interesting to notice that the initial chunking of data (as a preparation for the parallel mapping in MapReduce) may be realized by taking bites of a list. The initial chunking is, however, not really a central part of MapReduce.

In an earlier paper about mapping and filtering in functional and object-oriented programming [11] we have described the idea of *general mapping*. General mapping is characterized by (1) element selection, (2) element ordering, (3) function selection (selection of function(s) to apply on the selected elements), (4) calculation (which transformation to apply), and (5) the result of the mapping. Relative to this understanding, the current paper contributes to the first aspect, namely a more elaborate way of selecting the part of list to be transformed in the mapping process.

6. CONCLUSIONS

The abstractions introduced in this paper are useful in situations where it is necessary to process selected sublists of a list, in contrast to individual elements of a list. As illustrated in Section 4 the generated bite functions are, together with the bite mapping functions, useful for discovering structures among the elements in a list. As a use case, we have demonstrated how a number of important music related structures can be captured in Standard MIDI Files.

Figure 4 illustrates three typical bite mapping scenarios, supported by our bite mapping functions. The most regular scenario, as supported by `map-bites`, is a *complete, disjoint chunking of a list* followed by *processing of the chunks*, as shown in Figure 4(a). With use of `step-and-map-bites` we can approach application areas in which we more exhaustively search for certain "sequences of consecutive elements" which together not necessarily span the entire list. This situation is sketched in Figure 4(b). As mentioned briefly in Section 3, it is also possible to extract and process overlapping sublists with use of `step-and-map-bites`. This situation is shown in Figure 4(c).

The organization of the sublisting facilities as higher-order functions has been the primary focus of this paper. We have striven for natural generalizations of the classical `map` and `filter` functions, which are well-known in most functional programming languages. Thus, the main emphasis in this paper has been to provide mapping and filtering of sublists via a few functions (such as `map-bites` and `filter-bites`) which take other functions are parameters. The bite functions introduced in Section 2 are of particular importance among these function parameters. We have explored how a number of useful bite functions can be produced by bite function generators, such as `bite-of-length`, and `bite--while-element`.

The functions discussed in this paper and their API documentation are available on the web [10].

7. ACKNOWLEDGEMENT

8. REFERENCES

[1] Joseph Albahari and Ben Albahari. *C# 4.0 in a nutshell.* O'Reilly, 2010.

[2] Microsoft Corporation. The C# language specification version 4.0, 2010.
http://www.microsoft.com/downloads/.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communication of the ACM*, 51:107–113, January 2008.

[4] Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *ACM Sigplan Notices*, 27(5), May 1992.

[5] Richard Kelsey, William Clinger, and Jonathan Rees. Revised[5] report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.

[6] Mark Lutz. *Programming Python.* O'Reilly and Associates, Inc, 2006.

[7] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3:184–195, April 1960.

[8] Kurt Nørmark. Web programming in Scheme with LAML. *Journal of Functional Programming*, 15(1):53–65, January 2005.

[9] Kurt Nørmark. MIDI programming in Scheme - supported by an Emacs environment. Proceedings of the European Lisp Workshop 2010 (ELW 2010), June 2010. http://www.cs.aau.dk/∼normark/laml/papers/-midi-laml-paper.pdf.

[10] Kurt Nørmark. Bites of lists - API documentation and source code. http://www.cs.aau.dk/∼normark/bites-of-lists/, March 2011.

[11] Kurt Nørmark, Bent Thomsen, and Lone Leth Thomsen. Mapping and visiting in functional and object-oriented programming. *Journal of Object Technology*, 7(7), September-October 2008.

[12] Ricardo Scholz and Geber Ramalho. COCHONUT: Recognizing complex chords from MIDI guitar sequences. In *ISMIR*, pages 27–32, 2008. `http://ismir2008.ismir.net/papers/ISMIR2008_200.pdf`.

[13] Olin Shivers. SRFI 1: List library. http://srfi.schemers.org/srfi-1/.

[14] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised^6 report on the algorithmic language Scheme. *Journal of Functional Programming*, 19(Supplement S1):1–301, 2009.

[15] Guy L. Steele. *Common Lisp, the language, 2nd Edition*. Digital Press, 1990.

[16] Richard C. Waters. Automatic transformation of series expressions into loops. *ACM Trans. Program. Lang. Syst.*, 13:52–98, January 1991.

[17] Wikipedia. Chord (music) — Wikipedia, the free encyclopedia, 2011. [Online; accessed 25-February-2011].

APPENDIX
A.   DETAILED MIDI PROGRAMS

In this appendix we present more detailed examples related to the MIDI application area. The examples are all introduced and discussed at an overall level in the subsections of Section 4.

A.1   Bars

The MIDI function library contains a function `map-bars`, which activates `map-bites` with an appropriate bite function and bite transformation function. Here is a sample application of `map-bars` on some temporally strict music referred to as `SOME-MIDI-EVENTS`:

```
(map-bars
  (lambda (messages n st et) (list (midi-marker-abs-time st "Bar" n) messages))
  480                                                          ; Pulses Per Quarter Note.
  '(4 4)                                                       ; Time signature is 4:4.
  SOME-MIDI-EVENTS
)
```

In addition to the `messages` in the bar, the function mapped over the bar (the lambda expression shown above) receives a bar number `n`, the start time `st`, and the end time `et` of the bar. In the body of the lambda expression, we see that the messages in the bar are being prefixed with a MIDI marker.

In `absTime` mode, the function `map-bars` is implemented in terms of `map-bites` using a bite function generated by `bite--while-element`. Here is an outline of the use of `map-bites` in `map-bars`:

```
(define (map-bars f ppqn time-signature . messages)
  ...
  (map-bites
    (lambda (lst . rest)                                       ; The bite function.
     (let* ((start-time-first-mes (midi 'absTime (first lst)))
            (bar-number (quotient start-time-first-mes ticks-per-bar))    ; Zero based.
            (bar-start-time (* bar-number ticks-per-bar))
            (bar-end-time (+ bar-start-time ticks-per-bar))
            )
       ((bite-while-element (lambda (mes) (< (midi 'absTime mes) bar-end-time)) 'sentinel "first") lst)))
    (lambda (bite)                                             ; The bite transformation
       (let* ((start-time-first-mes (midi 'absTime (first bite)))    ; function.
              (bar-number (quotient start-time-first-mes ticks-per-bar))
              (bar-start-time (* bar-number ticks-per-bar))
              (bar-end-time (+ bar-start-time ticks-per-bar))
            )
         (f bite (+ bar-number 1) bar-start-time (- bar-end-time 1)) ))  ; Activation of f on the bar.
    messages))
```

The `bar-number`, `bar-start-time` and `bar-end-time` are needed for taking a bar bite from the MIDI messages. As it appears, these values are recalculated in the bite transformation function, as a service to the function `f` being mapped to the bars of the music. These calculations can be lifted out of the `map-bites` application to a multi-valued function. But due to the unpacking of these values in both lambda expressions, the modified programs is not shorter, not simpler, and probably not more efficient than the version with the recalculations shown above. As discussed in Section 4.6, it seems to be typical that information calculated in the bite function also is useful in the bite transformation function.

As an example of a more elaborate use of `map-bars`, we will show how it is possible to slide the tempo of every fourth bar down to half speed, and back again to normal speed:

```
(map-bars
   (lambda (messages n st et)
     (if (and (> n 0) (= (remainder n 4) 0))                     ; Every fourth bar.
         (list
            (tempo-scale-1 20                                    ; Use tempo scaling.
               120
               (make-scale-function-by-xy-points                ; A scaling function
                  (from-percent-points '((0 100) (50 50) (100 100))))) ; use for tempo scaling.
               120                                               ; Base tempo is 120 BPM.
              messages
            )
            (midi-marker-abs-time st "Bar" n)                    ; Still inserting markers.
         )
         (list messages (midi-marker-abs-time st "Bar" n)))
   )
   480
   '(4 4)
   SOME-MIDI-EVENTS)
```

As it appears in the lambda expression shown above, bars with bar numbers divisible by 4 are tempo scaled by use of the
function `tempo-scale-1`. The details of the tempo scaling is not relevant for this paper.

A.2 Pauses
At the top level, pauses are captured in a similar way as we located the bars in Appendix A.1.

```
(map-paused-sections
   (lambda (n mes-lst)
      (list (midi-marker "Start of paused section" n "P") mes-lst))
   130                                                           ; Pause time ticks.
   (lambda (ast) (and (NoteOn? ast) (= (midi 'channel ast) 1))) ; The relevance function.
   SOME-MIDI-EVENTS)
```

The function `map-paused-sections` has been implemented with use of `map-n-bites` and a bite function generated by
`bite-while-element-with-accumulation`:

```
(define (map-paused-sections f pause-ticks relevance-predicate . messages)
 (map-n-bites
   (bite-while-element-with-accumulation
      (lambda (mes sound-frontier-time)                         ; The predicate.
         (not (and (> (midi 'absTime mes) sound-frontier-time)
                   (> (- (midi 'absTime mes) sound-frontier-time)
                         pause-ticks))))
      (lambda (sound-frontier-time NoteOnMes)                   ; The accumulator.
         (max sound-frontier-time
             (+ (midi 'absTime NoteOnMes) (midi 'duration NoteOnMes))))
      0                                                         ; The initial value.
      (lambda (x)                                               ; The noise function.
        (and (ast? x)
             (or (not (relevance-predicate x)) (not (NoteOn? x)))))
   )
   (lambda (midi-messages-bite n)                               ; The bite transformation
     (f n midi-messages-bite))                                  ; function.
    messages))
```

A.3 Sustain intervals

The following application of `map-sustain-intervals` illustrates how to obtain a faster release of the sustain pedal, without affecting the way the pedal is moved downwards.

```
(map-sustain-intervals
  1                                                  ; The channel affected.
  (lambda (messages n direction)                     ; The function mapped on
    (cond ((eq? direction 'increasing)               ; intervals of messages that
            messages)                                 ; are monotone in sustain
          ((eq? direction 'decreasing)               ; control messages.
            (scale-attribute-by-factor-1
              (lambda (ast) (ControlChange? ast 64 1))
              'value
              0.75
              messages))
          ((eq? direction 'constant)
            messages)
          (else (laml-error "Should not happen"))))
  SOME-MIDI-EVENTS)
```

Only in intervals with decreasing pedal movement, the `value` attributes of the appropriate `ControlChange` messages are scaled by the factor of 0.75.

The function `map-sustain-intervals` is implemented with use of `map-n-bites`. The bite function is generated by `bite--while-monotone`.

```
(define (map-sustain-intervals channel f . mes)
 (let ((cc-val-comp
         (make-comparator
            (lambda (cc1 cc2) (< (midi 'value cc1) (midi 'value cc2)))
            (lambda (cc1 cc2) (> (midi 'value cc1) (midi 'value cc2)))))
       (noise-fn (lambda (x) (not (ControlChange? x 64 channel))))
     )
   (map-n-bites
     (bite-while-monotone                            ; The bite function generated
        cc-val-comparator                             ; with bite-while-monotone.
        noise-fn)
     (lambda (mes bite-number)                       ; The bite transformation
        (f mes bite-number                            ; function.
           (cond ((increasing-list-with-noise? cc-val-comp noise-fn mes)
                    'increasing)
                 ((decreasing-list-with-noise? cc-val-comp noise-fn mes)
                    'decreasing)
                 (else 'constant))))
     mes)))
```

As it appears, the sustain interval map function `f` gets information about the monotonicity of the MIDI message interval. This information has already been established in the bite function, but it needs to be re-calculated in the bite transformation function (via the two calls of `increasing-list-with-noise`). Without this information, we would not have been able to accomplish the task of dimming only the release of the pedal.

A.4 Chords

Chord identification makes use of `step-and-map-bites` instead of `map-bites`. Hereby the chords are identified in a more elaborate searching process than we have seen in the other examples. At top level, we search for chords in channel 1 of a piece of music in this way:

```
(map-chords
  1                                                  ; Channel number.
  40                                                 ; Max chord note distance.
  chord-marker                                       ; Chord markup function.
  SOME-MIDI-EVENTS)
```

The function `chord-marker` inserts markers (MIDI meta events) around a chord. In addition to a "chordal bite", `chord-marker` receives the channel number, the bite number, the successful chord formula, and the chord name.

Here follows the function `map-chords` in order to illustrate the use of `step-and-map-n-bites` and the bite function generated by `bite-while-element-with-accumulation`.

```
(define (map-chords channel max-time-diff f . messages)
 (let ((normalized-note-val (lambda (noteon-mes) (remainder (midi 'note noteon-mes) 12)))
       (relevant-message? (lambda (x) (and (NoteOn? x) (= channel (midi 'channel x)))))
       )
  (step-and-map-n-bites
     (bite-while-element-with-accumulation
      (lambda (mes prev-time)                              ; Keep going while
         (if prev-time                                     ; notes are dense.
             (if (< (- (time-of-message mes) prev-time) max-time-diff)
                 #t
                 #f)
             #t))
      (lambda (time mes)                                   ; Accumulate time of
        (time-of-message mes))                             ; previous note.
      #f                                                   ; Initial accumulation value
      (negate relevant-message?)                           ; The noise function.
      )
     (lambda (bite)                                        ; The int returning
       (let ((chord-list                                   ; predicate ...
              (map (lambda (no) (normalized-note-val no))
                   (filter relevant-message? bite))))
         (if (chord-match? (normalize-chord-list chord-list))   ; ... that determines a
             (length bite)                                 ; chord match
             -1)))                                         ; or a stepping value.
     (lambda (bite n)                                      ; The function applied on a
       (let ((normalized-chord-list                        ; a chordal bite. Prepares
              (normalize-chord-list                        ; calling f with useful
               (map (lambda (no) (normalized-note-val no)) ; information.
                    (filter relevant-message? bite)))))
         (f bite channel n normalized-chord-list
            (chord-name-of-normalized-note-list normalized-chord-list))))

     messages)))
```

As it appears, we locate chords in a given `channel`, among notes with `max-time-diff` time ticks between them. Only `NoteOn` messages in the given `channel` are taken into account. The noise function, formed by generating the negation of `relevant-message?` shown in line 3 of the fragment above, is important for disregarding MIDI events, which are irrelevant to the chord recognition process.