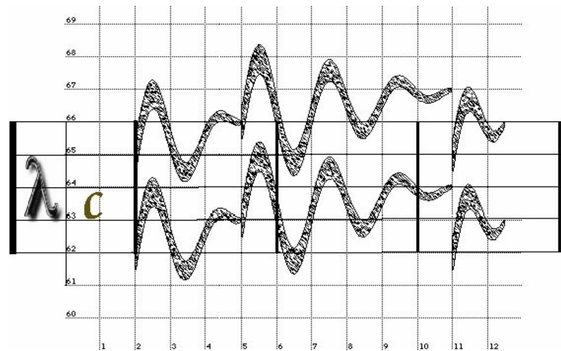
NoteFun: explorations in Lisp

Michiel Borkent

January 13, 2005



Internship report INF
Student number: 9908986
e-mail: michielborkent@gmail.com
Supervised by:
Peter Desain of the Radboud University Nijmegen
and Anton Nijholt of University of Twente Enschede
september 2004-januari 2005

You think you know when you learn, are more sure when you can write, even more when you can teach, but certain when you can program. - Alan Perlis

Chapter 1

Preface

Welcome to my internship report! My internship lasted from september 2004 till mid-januari 2005. It took place at the Radboud University in Nijmegen, The Netherlands. My supervisor was Peter Desain of the MMM group, part of the NICI[1]. In this internship I learned a lot and gained a lot of experience about programming, designing and music. To be more concrete, I had a thorough introduction into Common Lisp[2]. Not only I learned how to program in Lisp but also how to extend the programming language by writing macro's. I had to read about the midi protocol and think about how to use it in order to actually get sounds out of my program. Many other things like reading books and watching documentaries helped me to gain more understanding in the fields of informatics and music. The atmosphere at the MMM group was a very inspiring one.

Thanks go out to Peter Desain, Barbara Raven, Makiko Sadakata, Anton Nijholt, Jan Kuper, Yvonne Schoute and many others who helped me in some way or another to get to the finish of my internship.

We have decided to name this project NoteFun. The ambiguity of this name gives rise to many explanations which I leave as an exercise to the reader.

Nijmegen, January 13, 2005,

Michiel Borkent

[1] See http://www.nici.ru.nl/mmm for more information.
[2] When I mention Lisp in this report, I always mean Common Lisp.

Abstract

In this report we will explore functions of different times, *GTFs*, as described in Desain and Honing (1992) and Honing (1995). These functions apply to the start, duration and actual time of a note. In NoteFun we will use these functions to continuously control different aspects of notes e.g., volume and pitch. First we will lead you through a hopefully easy-to-understand tutorial. Then we will elaborate about how functional languages and Lisp perform compared to other programming languages. Lastly we will tell more about specific implementation issues of the NoteFun environment.

Contents

1 **Preface** **3**

Contents **5**

2 **Introduction** **7**
Description of the project 7

3 **Tutorial** **10**
3.1 Discrete musical events 10
3.2 Extension towards continuous control 13
Simple GTFs 13
Complex GTFs: creating GTFs out of other GTFs 17
Intermezzo: guide to a wider application of GTFs 22
Complex GTFs continued 23
GTFs created from mathematical functions 29
Lifting tables 32
Creation of GTFs by writing your own Lisp code 33
Case study: ADSR 36

4 **Overview of NoteFun's functions** **42**
4.1 Construction of musical objects 42
4.2 Construction of GTFs 43
4.3 Mediate musical objects 44
Show musical objects 44
Play musical objects 45

5 **Other programming languages compared to Lisp** **46**
5.1 Introduction 46
5.2 Why Lisp? 46
Ad 1 47
Parentheses scary? 47

Familiarity . . . 48
Hybrid style: good or bad? . . . 48
Specific or general purpose? . . . 48
Side effects and reasoning about programs . . . 49
Survival of the fittest? . . . 49
Ad 2 . . . 50
Dynamical typing and regularity in representation . . . 50
Ad 3 . . . 50
Code esthetics . . . 50
Debugging . . . 51
Interactiveness . . . 51
5.3 Conclusions . . . 52

6 Design and Implementation **53**
6.1 Notes on functional programming style . . . 53
6.2 Extending the language by use of macro's . . . 54
6.3 One layered versus two layered GTFs . . . 58
6.4 Midi output . . . 61
Translation of pitch related GTFs . . . 61
Translation of other aspect related GTFs . . . 63

7 Final words **65**

Bibliography **66**

Chapter 2

Introduction

Description of the project

In this project we have developed a microworld called NoteFun, that deals with continuous control of aspects of notes in music, as an extension to a discrete approach to musical events. These aspects, e.g., pitch and volume can be described using *functions of time*. Examples of such functions are a sine wave applied as a notes vibrato, or a linear function applied as a glissando between two pitches.

Interesting questions arise when we manipulate durations of notes. What do we want to happen to the sine wave, attached to the note, when we want to stretch the note to make it, e.g., twice as long? In most natural cases we want the sine function not to be stretched with the new duration of the note. We want to maintain its original form and concatenate more of the similar sine waves till it fits the new duration. See Figure 2.1 and Figure 2.2 for a demonstration of this. In some cases we might want to stretch the sine function for some reason. Figure 2.3 exemplifies this. In all figures used in this report the horizontal axis represents time, the vertical axis represents pitch and the thickness of the line representing the GTF indicates the volume.

To make similar behaviors of functions possible, we need an environment in which we can specify this behavior exactly. In order to do this we will introduce the notion of different times. This idea stems from and is elaborated in Desain and Honing (1992) and Honing (1995).

These functions of different times will be referred to as *GTFs*: generalized time functions. After introducing the GTFs we will introduce a number of tools with which one can build from simple GTFs more complex ones. These GTFs can be used for different aspects of notes, e.g., volume, brightness, pitch, panning, etc. We will also see the application of a GTF

to a group of notes.

During the project we have also developed a mechanism which translates our representation of musical events to midi control data. This data can be interpreted and made audible by a MIDI supporting synthesizer.

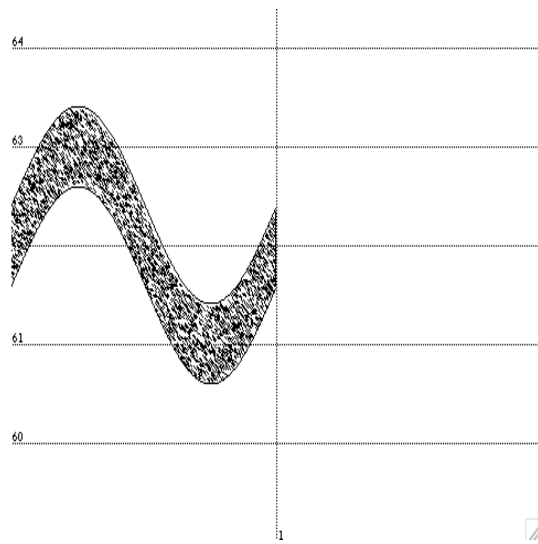


Figure 2.1: Note with a sine function attached, causing a periodic deviation from its pitch.

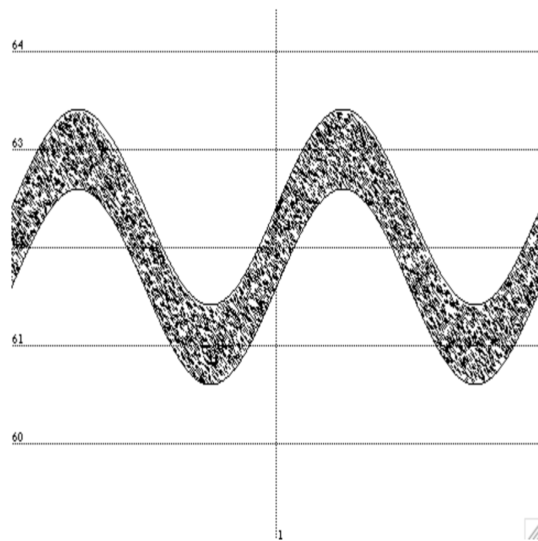


Figure 2.2: Stretched note with the same periodic deviation from its pitch.

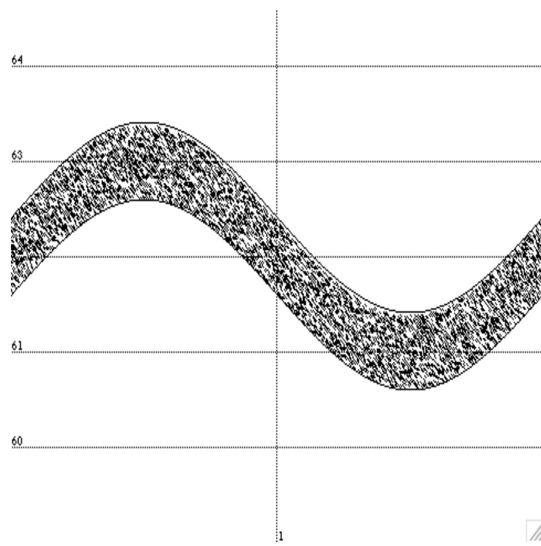


Figure 2.3: Stretched note with a stretched periodic deviation.

Chapter 3

Tutorial

3.1 Discrete musical events

This tutorial provides information about the usage of NoteFun. It is not about Lisp programming but it does assume you have some knowledge about Lisp notation. We have tried to reduce the amount of programming-related issues although all the things you know about Lisp programming will probably make you a better composer in NoteFun.

In electronical composition one needs to specify elementary musical events which together constitute a musical piece. The most elementary musical events in NoteFun are a `note` and a `pause`.
For example, `(note :pitch 58 :duration .5)` and `(pause 1.5)` are valid events in NoteFun. The default pitch and duration of a note are 60 and 1 respectively. This means that you can write `(note)` instead of the longer `(note :duration 1 :pitch 60)`. As you can see, the order of the attributes which you specify does not matter. This is because each attribute goes along with a keyword, indicated by a colon as the first character. In fact, `note` has a lot more attributes which we will ignore for now, as we can use their default values for the moment. The number which indicates the pitch corresponds to semitones, whereas the middle-C is indicated by 60.

In composition, like the word indicates itself, we constitute a work out of more basic objects than the work itself. In our case this means that we would want to make groups of notes that are somehow related. In NoteFun you can make a group of notes in two ways. One way is to make a sequence of notes and pauses. For example, we could think of the first line of *Are you sleeping*[1] as a time-wise sequence of the notes c,d,e and again c. In Notefun we could compose this as

[1] In dutch this song is known as *Vader Jacob*

Listing 3.1: example of a sequence

```
(seq (note)
     (note :pitch 62)
     (note :pitch 64)
     (note))
```

`seq` can take any number of notes and groups of notes. For example, we could repeat the first line of *Are you sleeping* twice, just like in the song itself:

Listing 3.2: Example of a bigger sequence

```
(seq (seq (note :pitch 60)
          (note :pitch 62)
          (note :pitch 64)
          (note :pitch 60))
     (seq (note :pitch 60)
          (note :pitch 62)
          (note :pitch 64)
          (note :pitch 60)))
```

You already can see right now that composition of the complete version of *Are you sleeping* would result in a long and unreadable list of 'seqs' and 'notes'. This problem can be overcome by making a reference to an expression. For example, we could name the first line of our song `line1`. This is done by the following:

Listing 3.3: Example of making a reference

```
(setq line1 (seq (note)
                 (note :pitch 62)
                 (note :pitch 64)
                 (note)))
```

Now we can represent the double repetition of line1 as:

```
(seq line1 line1)
```

which of course looks a lot cleaner. The second way of grouping notes is ordering them parallel in time. This can be done with `sim` which stands for *simultaneous*. For example a major-C triad can be written as

```
(sim (note) (note :pitch 64) (note :pitch 67))
```

and the second inversion of the major-G triad as

```
(sim (note :pitch 59) (note :pitch 62) (note :pitch 67))
```

To abbreviate these chords we write:

```
(setq c-chord (sim (note) (note :pitch 64) (note :pitch 67)))
(setq g-chord (sim (note :pitch 59) (note :pitch 62) (note :pitch 67)))
```

Note that with our little bit of references we can already write a simple yet nice arrangement for the beginning of *Are you sleeping*:

```
(setq line1-arrang (seq c-chord g-chord c-chord c-chord))
(setq ays (sim (seq line1 line1) (seq line1-arrang line1-arrang)))
(play ays)
```

With `play` we send our composition to a connected midi-synthesizer which then interprets it into sounds. If `play` is somewhat problematic in your set-up, you might want to use `save` instead, which lets you save the musical data as a standard midi file which you can later use in whatever manner you like.

As you can hear, the melody and the chords get a bit in each others way. It would be nice if we could transpose `line1-arrang` one octave lower. No problem. We do it with the following construction:

```
(setq line1-arrang-transposed (trans -12 line1-arrang))
(setq ays (sim (seq line1 line1)
               (seq line1-arrang-transposed
                    line1-arrang-transposed)))
(play ays)
```

`trans` takes a number[2] and a musical object which is decreased or increased in pitch, according to the number. The number represents a number of semitones as you might have guessed. -12 then indicates "one octave lower".

The standard *program number* (which indicates an instrument in the General Midi scheme) of a note is 80. It would be nice if we could use one instrument for melody and a different one for the rest of the arrangement. Let us change `ays` in that we have piano for the melody and strings for the chords. First let us explain how to change the instrument for a single note and after that how to change it for a group of notes. Actually the first can also be done by passing a keyword with a value to note:

```
(note :program 49)
```

The above note sounds as a middle C of one second long, played by a string ensemble. It would be a tedious job to specify the non-default instrument per note. That is why we will now introduce `for-all-notes` that lets us express that all notes in the chords must be played by the string ensemble:

```
(setq line1-arrang (for-all-notes :program 49 line1-arrang))
```

As you might know, two different instruments cannot be played at the same midi channel. That is why we have to explicitly specify that the chords have to be played at an other channel than the melody, which will be played at the default channel (number 0).

```
(setq line1-arrang (for-all-notes :channel 1 line1-arrang))
```

Please note that when binding a variable to an expression in which the same variable is used can be tricky. Suppose we had written this:

[2] To be more general, `trans` takes not only a number but also a *GTF*. We will explain this later on.

```
(setq line1-arrang (trans -12 line1-arrang))
```

After executing this for the fist time `line1-arrang` will be transposed one octave down. After you execute the same code again, `line1-arrang` will be transposed one more octave down. This might be obvious but it is recommended to use these kinds of constructs with special attention. Finally we are ready to play the melody and the chords together, both using their own instruments. Note that we have to evaluate the following expression again:

```
(setq ays (sim (seq line1 line1) (seq line1-arrang line1-arrang)))
```

before we can use `ays` as the new composition with multiple instruments, because `ays` has to be build up with the new `line1-arrang`. One more remark on `for-all-notes` is that you can pass multiple keyword/argument pairs at once. We could have done the last two operations on `line1-arrang` in this shorter way:

```
(setq line1-arrang (for-all-notes :channel 1 :program 49 line1-arrang))
```

A very interesting operation which will bring us to the continuous extension of this still discrete world of notes is `stretch`. You might have had the feeling that `ays` is still a bit too slowly played. Wouldn't it be nice to shorten the whole musical object at once, instead of specifying a shorter duration for each note and building a new `ays`? Then try this:

```
(play (stretch .5 ays)).
```

It hopefully sounds a lot better.

3.2 Extension towards continuous control

Simple GTFs

Now we will show how to use functions of `start`, `duration` and `time` to control a notes internal workings. A nice example to begin with is a glissando from one note to another. Let us have two notes with a pause in between, which we will later replace with the glissando.

```
(draw (seq (note) (pause 1) (note :pitch 62)))
```

`draw` draws a piano roll representation of a musical object on the screen, as you can see in, e.g., Figure 3.1. The center of the notes height indicates the pitch and the height itself indicates that amplitude. As you can see, the pitch and the amplitude of both notes remain constant throughout the notes duration. Now we will put a note in between of these notes which represents a glissando from the first to the second note. A glissando's pitch glides linearly from the pitch of the first note to the pitch of the second note. You see that we need a construct that enables us to change an aspect

of a note throughout its duration. This is what we call a *GTF*, a generalized time function. Now we will give the code with a glissando instead of a pause (see Figure 3.2):

```
(draw (seq (note) (note :pitch (glissando 60 62)) (note :pitch 62)))
```

You can read `(note :pitch (glissando 60 62))` as: a note that has a line from 60 to 62 spread over its duration as a pitch. This means that, if you will change the notes duration, the glissando will adapt to the new duration and it will still be a glissando from 60 to 62. To demonstrate this, we will shrink the whole musical object (see Figure 3.3).

```
(draw (stretch .5 (seq
                     (note)
                     (note :pitch (glissando 60 62))
                     (note :pitch 62))))
```

.

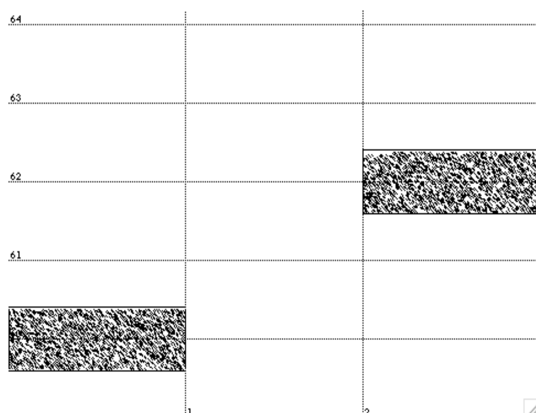


Figure 3.1: Two notes separated by a pause

Apparently, the glissando behaves well under transformation, since this is how we intend a transformed glissando to behave. A different transformational behavior is shown in the vibrato example. Suppose we want to add a nice vibrato to the first note. We can do that as follows (also see Figure 3.4):

```
(draw (seq (note :pitch (vibrato :around 60 :period .2 :amplitude .4))
           (note :pitch (glissando 60 62))
           (note :pitch 62)))
```

It might be self-explanatory that the keyword `around` indicates around which pitch the periodic deviation moves. The `amplitude` indicates how much the deviation from the notes pitch will be. [3] When we shrink this object,

[3] `vibrato` has got more keywords which you can check out in the source code.

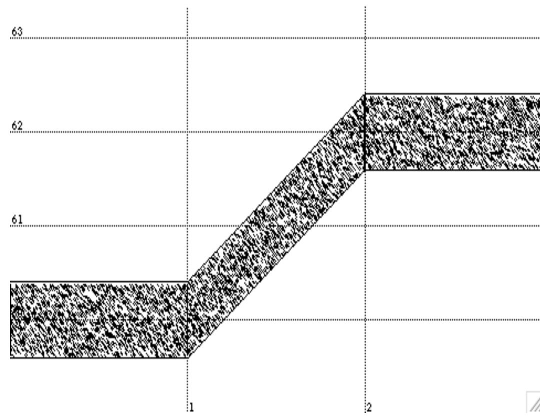


Figure 3.2: Two notes connected by a glissando

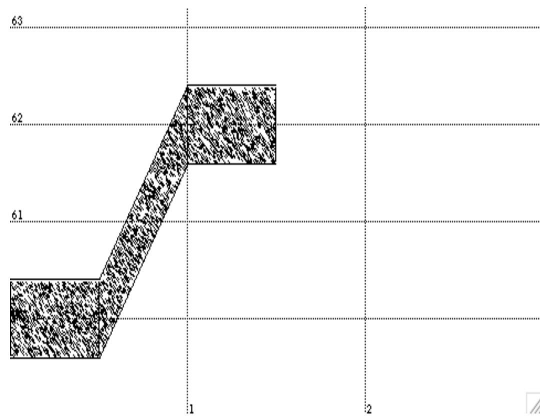


Figure 3.3: Musical object from figure 3.2 but shrinked

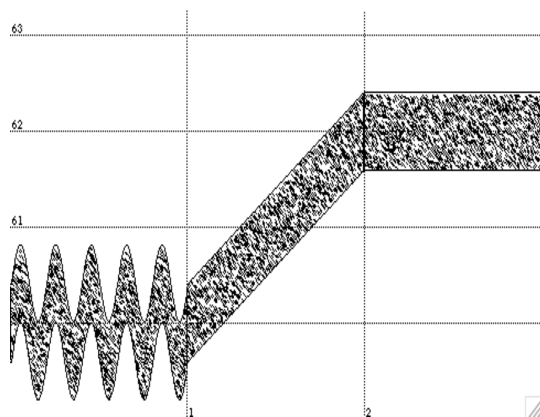


Figure 3.4: The first note has a vibrato.

we want the vibrato of the first note *not* to shrink. Instead we want the amount of periods to decrease. If the vibrato would shrink the note would sound totally different and not the way we intend to. Now let us shrink the object again (see Figure 3.5):

```
(draw (stretch .5 (seq (note :pitch (vibrato :around 60
                                             :period .2
                                             :amplitude .4))
                       (note :pitch (glissando 60 62))
                       (note :pitch 62))))
```

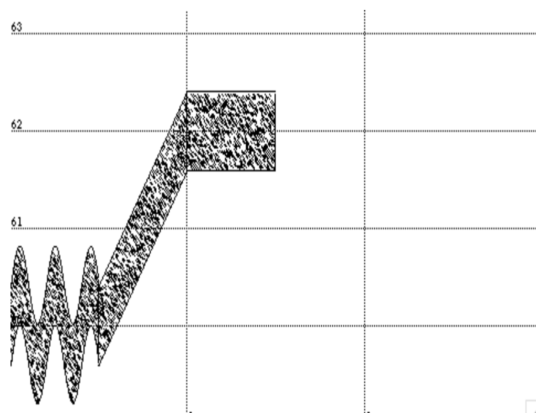


Figure 3.5: Musical object from figure 3.4 but shrinked

As you can see, `vibrato` wonderfully behaves under this transformation like we intend to.

To conclude this subsection we will give a 3D plot in which you can see the behavior of a vibrato and a glissando within all possible durations from the interval $[0, 3]$. In Figure 3.6 you see how the vibrato behaves in time for all durations. The NoteFun code for the vibrato is `(vibrato :period 1)`. In Figure 3.7 you see that the glissando goes from 60 to 62 in a linear way according to the durations. The NoteFun code for the glissando is `(glissando 60 62)`.

In the next chapter we will introduce four methods of creating new GTFs.

- Creation of GTFs out of other GTFs
- Creation of GTFs out of mathematical functions (lifting)
- Creation of GTFs out of numeric tables
- Creation of GTFs by writing your own Lisp code

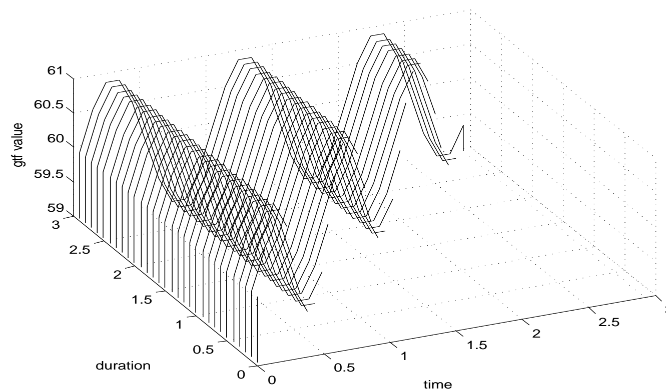


Figure 3.6: Vibrato with various durations ranging from 0 to 3 seconds plotted in time.

Complex GTFs: creating GTFs out of other GTFs

Now we will introduce tools with which you can make complex GTFs from simpler ones. One is called `concat-gtf` and like its name says, you can concatenate GTFs with it. With the ones we already saw we can e.g., make a concatenation of a vibrato and a glissando and apply it to one notes pitch. We will have to specify what portion of a duration will be dedicated to what GTF. This is done by means of `absolute`, `relative` and `proportional` who all indicate a point in time within the duration of a note in their own way. The argument list of `concat-gtf` is built up from alternating GTFs and indications of time. An example of the use is given here (also see Figure 3.8):

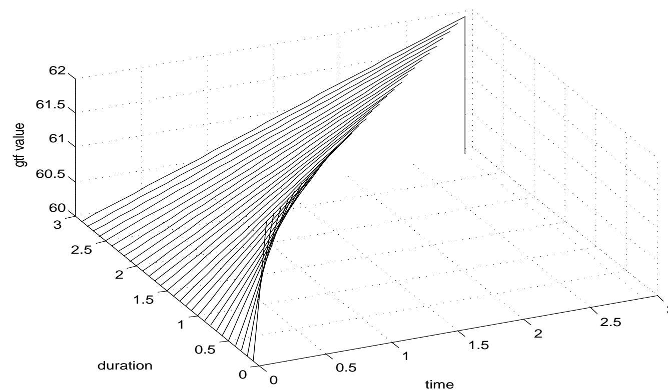


Figure 3.7: Glissando with various durations ranging from 0 to 3 seconds plotted in time.

```
(draw (note :pitch (concat-gtf (glissando 60 62)
                               (proportional .5)
                               (vibrato :around 62 :period .2))))
```

You might wonder what `(proportional .5)` stands for. It means that what is before this expression may take half (hence .5) of the duration of the musical object, in this case a note, and what is after this expression may take the other half. This means that `(glissando 60 62)` in this case may take up half a second, since the default duration of a note is 1 second, and `(vibrato :around 62 :period .2)` may take the rest which is also half a second. If we transform the note by means of `stretch` the durations of the GTFs will transform proportionally with the new duration. Instead of proportionally determined points in time you can also use `relative` points

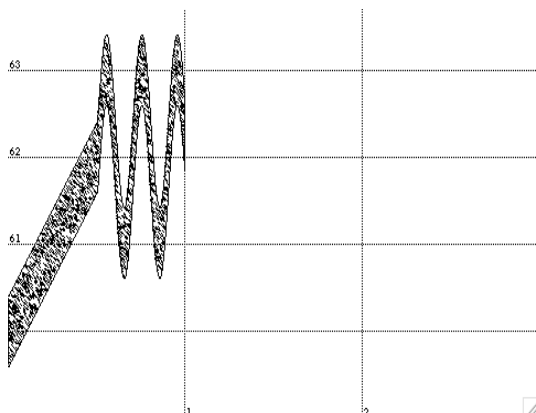


Figure 3.8: A glissando and vibrato within one note

and `absolute` points. Relative means in this case: seen from the start of the note. Absolute means: seen from the start of the whole piece. Let us give a complex example to demonstrate how `relative` and `absolute` behave under transformation (also see Figure 3.9):

Listing 3.4: Complex example of concat-gtf

```
(draw (seq (note :duration 4 :pitch (concat-gtf
                                     (glissando 60 62)                 2
                                     (proportional .25)
                                     (vibrato :around 62 :period .2)   4
                                     (relative 1.8)
                                     (glissando 62 64)                 6
                                     (absolute 3.5)
                                     (glissando 64 60)))               8
           (note :duration 5 :pitch (concat-gtf
                                     (glissando 60 62)                 10
                                     (proportional .25)
                                     (vibrato :around 62 :period .2)   12
                                     (relative 1.8)
                                     (glissando 62 64)                 14
                                     (absolute 7.5)
                                     (glissando 64 60))))              16
      :y-min 58 :y-max 65)
```

You see this quite a complex beast. In this example there are two notes in a sequence. The first has a duration of 4 seconds and the second has a duration of 5 seconds. You can see that `(proportional .25)` (on line 3 and 11 of Listing 3.4) indicate different points counted from the starts of the notes, as it stands for $4 \times .25$ (= 1) in the first note and $5 \times .25$ (= 1.25) in the second note. The expression `(relative 1.8)` indicates the same point in time counted from the starts of the notes, namely 1.8 seconds after the start. This always holds, even when a note has been transformed. The expression `(absolute 3.5)` indicates the point in time that is 3.5 seconds after

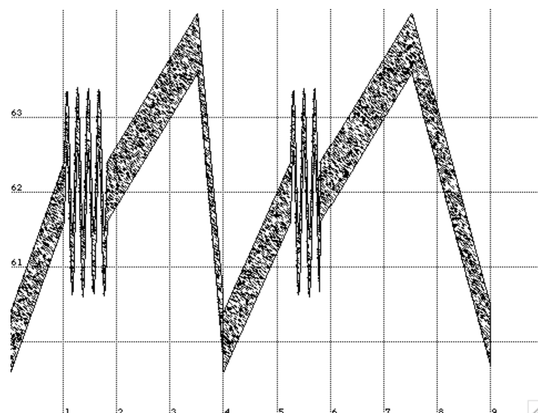


Figure 3.9: Complex example of concat-gtf

the start of the entire musical object, in this case a sequence holding the two notes. `(absolute 7.5)` of course indicates the point in time 7.5 seconds after the start of the entire musical object.

Now answer these questions for yourself before trying out the encoding of the problems in NoteFun:

- How will the musical object look when you add a `(pause .5)` in the sequence in front of the notes?
- What will happen when you change the duration of the last note from 5 to 4?

In both cases, first try to calculate the new points in time where one GTF ends and another begins. Let us call those points *switch points* from now on. Afterwards look at the corresponding pictures and see if your guess was correct.

Now for a malicious use of concat-gtf, have a look at what happens here (also see Figure 3.10):

```
(draw (note :pitch
            (concat-gtf                                         2
             (glissando 60 62)
             (absolute .5)                                      4
             (glissando 62 60)
             (absolute .25)                                     6
             (glissando 60 62)
             (absolute 1.5)                                     8
             (glissando 60 62))))
```

The glissando on line 3 takes half the note, as you can expect. But what must be the behavior of the glissando on line 5, since the point in time indicates by line 6 is before the point in time indicated by line 4. . . ? `Concat-gtf`

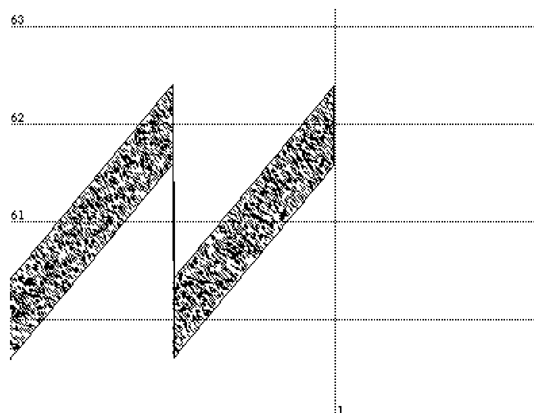


Figure 3.10: Complex example of concat-gtf

prints a warning on the screen saying: "Times being re-arranged!" `Concat-gtf` checks if the successive switch points of its argument list are monotonously increasing. Also it check whether all the switch points remain between the bounds of the start and end time of the note. If not, it changes the switch points so they will obey these two properties. In our last example `concat-gtf` changes the switch point on line 6 into `(absolute .5)` (first property) and the switch point on line 8 into `(absolute 1)` (second property). The result is that the glissando of line 5 will not appear since it has a time interval of exactly 0 seconds. The glissando of line 5 will appear on the interval $[.5, 1]$. The glissando of line 9 will not appear since it has a time interval of 0 seconds. In other words, the last example is equivalent to the following code:

```
(draw (note :pitch
            (concat-gtf
             (glissando 60 62)
             (absolute .5)
             (glissando 60 62))))
```

Try this out and see for your self.

We conclude this subsection with a 3D graph of a complex GTF with duration from the interval $[0, 3]$ plotted in time. See Figure 3.11. The NoteFun code for this GTF is:

```
(concat-gtf
 (glissando 60 62)
 (proportional .25)
 (vibrato :around 62 :period .2)
 (relative 1.8)
 (glissando 62 64)
 (absolute 3.5)
 (glissando 64 60))
```

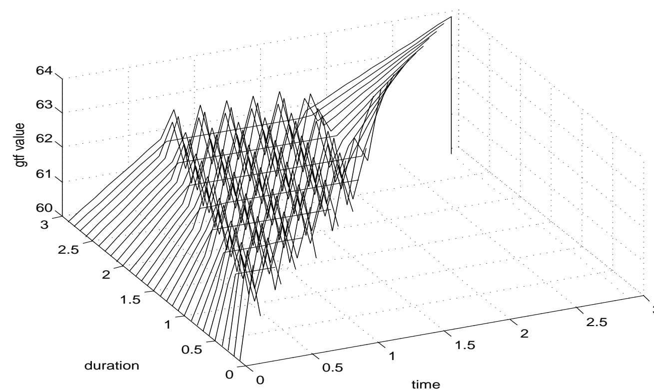


Figure 3.11: Complex GTF with various duration plotted in time

Intermezzo: guide to a wider application of GTFs

Note that until now we have only used GTFs in combination with the `pitch` of a `note`. To stimulate your creativity we must remark that GTFs can also be used in combination with a lot of other possible features of the `note` like `amplitude` and midi related parameters like *brightness* and *pan*. To simulate a fade-out you can use a GTF that goes from 1 to 0, where 1 indicates full volume and 0 indicates total silence (also see Figure 3.12):

```
(draw (note :amplitude (ramp 1 0)))
```

As you might guess `ramp` is similar to glissando. In fact, it is actually the same function. It only has a more general name and unlike glissando its name will inspire us to use it for all kinds of things, like here for `amplitude`.

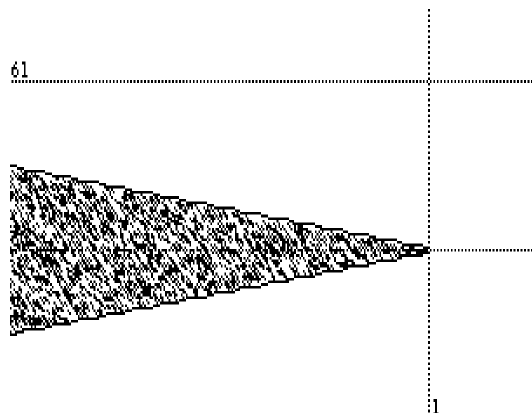


Figure 3.12: Note with a fade out

To make use of the midi controller related features of note, you can add the synthesizer specific controller number and the range in which you want to control it yourself. The code for the controllers that we have put in for our own synthesizer[4] looks like this:

```
(add-midi-controller :pan 10 -1 0 1)
(add-midi-controller :balance 8 0 .5 1)
(add-midi-controller :timbre 71 0 .8 1)
(add-midi-controller :brightness 74 0 .8 1)
```

You can make up any keyword your want for your specific controller. The syntax for add-midi-controller is

```
(add-midi-controllers
  <keyword>
  <midi control number>
  <minimal-value>
  <default-value>
  <maximum-value>)
```

For a panning from left to right within a note we could write:

```
(note :pan (ramp -1 1))
```

You can remove a midi controller by executing (remove-midi-controller <name>). For example, if we want to remove the controller for brightness, we write: (remove-midi-controller :brightness). To get a list of all currently available midi controllers you write: (show-midi-controllers).

Complex GTFs continued

Another tool for merging simple GTFs into a complex one is switch-gtf. It works slightly different than concat-gtf but has the same way of calling.

[4]A Yamaha MU90R

Instead of concatenating GTFs it switches between GTFs at certain switch points and the GTFs are all spanned over the notes entire duration. Let us give an example based on two glissando's, one going up and one going down. Within the note we will switch between them using `switch-gtf` (also see Figure 3.13):

```
(draw (note :duration 3
            :pitch (switch-gtf (glissando 60 65)
                               (proportional .25)
                               (glissando 65 60)
                               (proportional .5)
                               (glissando 60 65)
                               (proportional .75)
                               (glissando 65 60))))
```

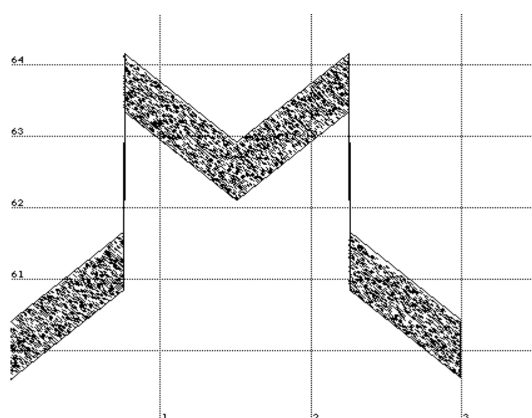


Figure 3.13: `Switch-gtf` switching between two glissando's

`switch-gtf` corrects badly successive switch points in the same way as `concat-gtf` does.

Another way of building new GTFs from other ones is called *time warping*. Instead of going through a GTF from left to right in time, we can e.g., do it from right to left; from the beginning to the end. This operation can be done with a time warp that we called `time-reverse`. Its use is illustrated in the following example.

```
(setq some-gtf (concat-gtf
                 (glissando 60 62)
                 (proportional .25)
                 (vibrato :around 62 :period .2)
                 (proportional .5)
                 (glissando 62 64)
                 (proportional .75)
                 (glissando 64 65)))

(draw (note :pitch some-gtf) :y-max 66)
(draw (note :pitch (warp-gtf some-gtf (reverse-time)) :y-max 66)
```

First we make a reference to a GTF to be able to write more readable code. We use the name `some-gtf` as the reference name. Then we draw the result of using `some-gtf` as a notes pitch (see Figure 3.14). Lastly we draw the result of a transformed version of `some-gtf` by means of the time warp `time-reverse` (see Figure 3.15).

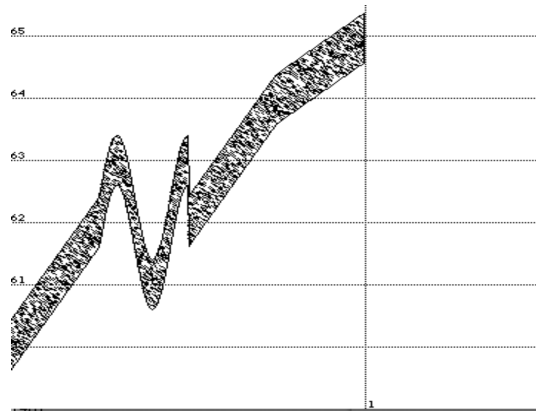


Figure 3.14: Note with `some-gtf` as a pitch

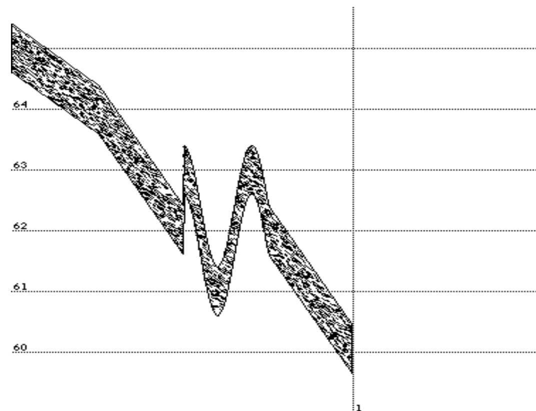


Figure 3.15: Note with `some-gtf`, transformed by a time warp, as a pitch

Another handy time warp is `make-time-cyclic`. `Make-time-cyclic` has one argument indicating a point time within the duration of the note by means of `absolute`, `relative` and `proportional`. I will show the use of `make-time-cyclic` by means of an example (see Figure 3.16):

```
(draw (sim (note :duration 3
                 :pitch (warp-gtf (glissando 62 65)
                                  (make-time-cyclic (proportional .5))))
```

```
          (note :duration 3
                :pitch (warp-gtf (glissando 60 62)
                                 (make-time-cyclic (relative .5)))))
      :y-max 66)
```

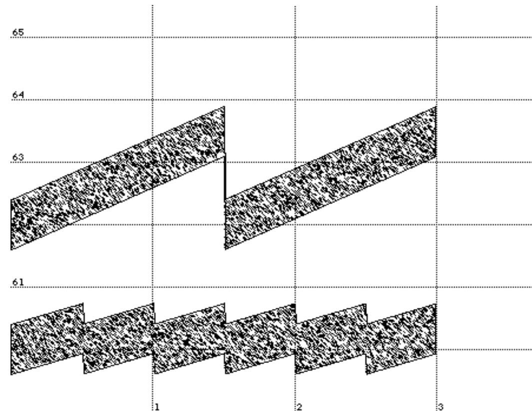


Figure 3.16: Note with a glissando which is transformed by a time warp, as a pitch

By now it might be clear how `make-time-cyclic` works. The point in time indicated by the time switch is the period length which indicates after what amount of time the GTF should start over again as if it were from the start.

There are more facilities that let us create GTFs out of other ones. One is called `mix-gtf`. By means of an example we will show its use (see Figure 3.17):

```
(draw (note :duration 3 :pitch
            (mix-gtf
             (glissando 60 70)
             1
             (sine-gtf :period 2 :amplitude 1) 1))
      :y-max 75)
```

The syntax in which you can use `mix-gtf` is: `(mix-gtf [<gtf> <factor>]*)`. The factors in this expression can be real numbers or GTFs[5]. The result of this expression will be a gtf which is the weighted result of the succeeding gtfs and factors in the argument list. Our last example will be a gtf which is the addition of a glissando from 60 to 70 and a sine with an amplitude of 1 and a period of 2 seconds. Check Figure 3.17 if it looks like you expect. Because of the fact that we can either fill in a real number or a GTF as

[5]The notation we have used for the argument list must be read as a regular expression in which `<gtf>` and `<factor>` can be arbitrary GTFs and numbers. The same holds for successive examples in which this notation is used.

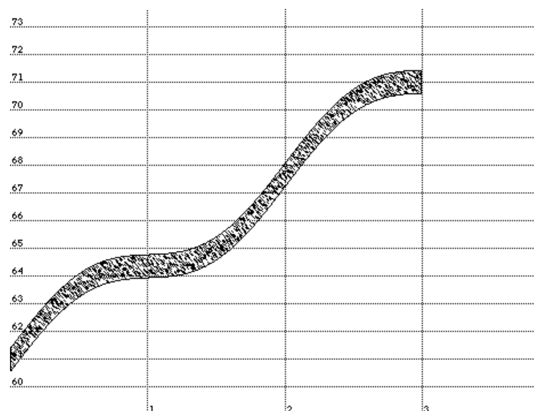


Figure 3.17: Example of mix-gtf

the factor, we are able to express things like: "let the impact of a GTF run from 2 to 0 along with the duration of the note". As follows:

```
(draw (note :duration 3 :pitch
            (mix-gtf (ramp 60 70)
                     1
                     (sine-gtf :amplitude 1)
                     (ramp 2 0)))
      :y-max 75)
```

The first GTF is kept constant (the succesive factor is 1) but the second GTF is multiplied by a ramp from 2 to 0. What we will expect is a ramp with a periodic deviation of which the amplitude will decrease as we go through the note in time. You can see the result in Figure 3.18.

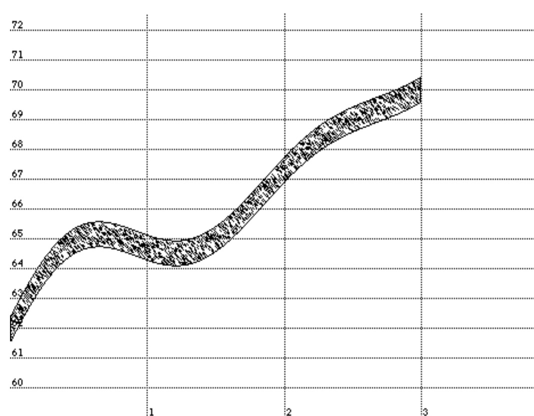


Figure 3.18: Second example of mix-gtf

Another similar combinator of GTFs is `interpolate-gtf`. As its name suggests, it interpolates between two GTFs. To make this clear let us interpolate between a glissando and a vibrato:

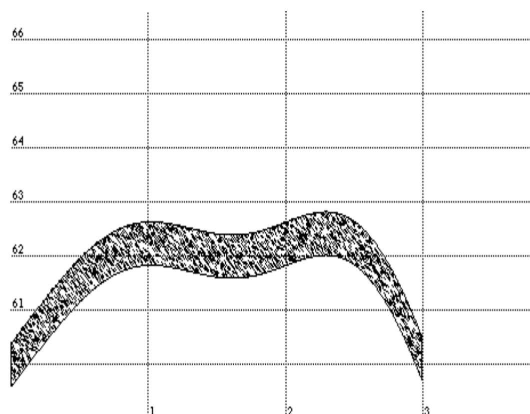


Figure 3.19: Example of interpolate

```
(draw (note :duration 3 :pitch (interpolate-gtf (glissando 60 70) (vibrato)))
      :y-max 70)
```

The interpolation can be defined in terms of `mix-gtf`:

`(interpolate-gtf gtf1 gtf2)` ≡ `(mix-gtf gtf1 (ramp 1 0) gtf2 (ramp 0 1))`

To give more elementary operations on GTFs we will introduce `gtf+` and `gtf*`. After that we will introduce a more general function with which you can make your own operation on GTFs.

A GTF that is the sum of multiple other GTFs can be made with `gtf+` with the following syntax: `(gtf+ [<gtf | number>]*)`. For example, we can add a glissando and another glissando so that they will compensate each other, resulting in a constant pitch:

```
(note :pitch (gtf+ (glissando 30 40) (glissando 30 20)))
```

The result is just a note with a constant pitch of 60.

The next example demonstrates the use of `gtf*` within the use of `gtf+`.

```
(draw (note :pitch (gtf+ 60 (gtf* (ramp 0 1) (sine-gtf :period .2)))))
```

The result is a note with a with-time-increasing periodic deviation from pitch 60 (see Figure 3.20).

If you require more operators on GTFs you can define them yourself using `time-fun-compose`. For example, if we would like to make the operator for division on GTFs we write:

```
(time-fun-compose #'/ [<gtf>]*)
```

More close to Lisp, a usable definition could be called `gtf/` and would look like this:

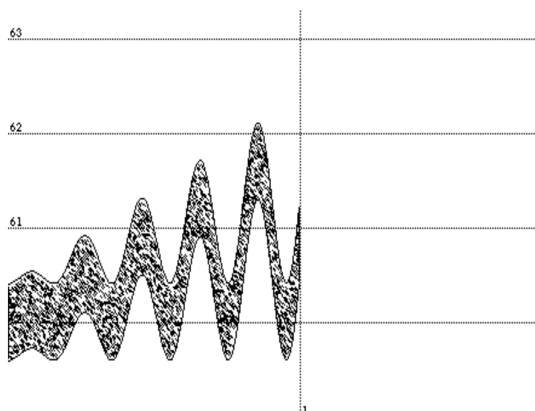


Figure 3.20: Example of gtf+ and gtf*

```
(defun gtf/ (gtf1 gtf2 &rest args)
  (apply #'time-fun-compose #'/ gtf1 gtf2 args))
```

We can use `gtf/` from now on as demonstrated in the following example (see Figure 3.21):

```
(draw (note :duration 2
            :pitch (gtf+ 60 (gtf/ (sine-gtf :amplitude 5) (ramp 2 10)))))
```

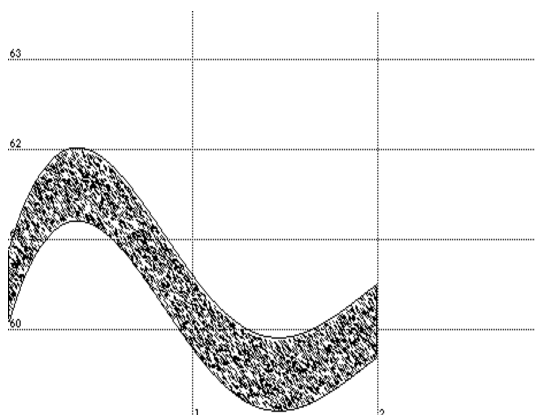


Figure 3.21: Example of gtf/ embedded in a gtf+

GTFs created from mathematical functions

In this section we will describe how one can build a GTF based on a mathematical function, like the sine function. This way of building gives one more degree of freedom of customizing your own GTFs as you will not be only

limited to the tools we provided before. Instead you can come up with any mathematical function you wish. You can specify how it should behave under transformation. Like we saw with the glissando, some functions should transform in a proportionally stretching way. Other functions, similar to the vibrato, should not be stretched. We call the creation of a GTF out of a mathematical function *lifting*.

A mathematical equivalent of the glissando might be a linear function, like (lambda(x) x). Suppose we want to have a mathematical function that corresponds to a glissando from 60 to 70. Note that we do not know the duration of the note, to which the function will be applied, since it can be any positive real number. For now we will just choose a function that goes from 60 to 70 in an interval of 1 second: (setq linfunc #'(lambda(x) (+ 60 (* 10 x)))) Now look at the following example (see Figure3.22):

```
(draw (note :duration 2 :pitch (lift linfunc (proportional-lift 0 1)))
      :y-max 72)
```

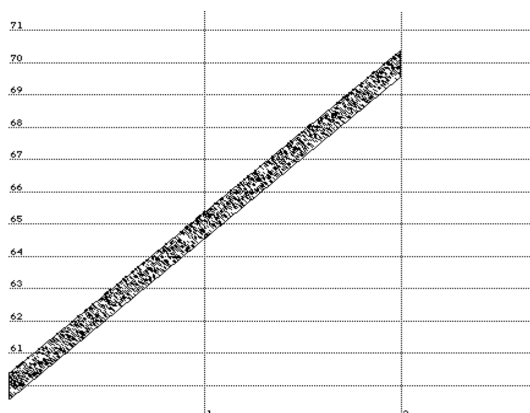


Figure 3.22: Example of a mathematical function lifted proportionally

What does (lift linfunc (proportional-lift 0 1)) mean? Informally it means, select the function linfunc on the interval $[0, 1]$ and stretch it according to the duration of a note.

More general, the syntax for lift is: (lift <mathematical function> <lift-function>). proportional-lift is a lift function that selects an interval out of a mathematical function. Try to modify the duration of the note in the last example and see that its pitch still runs from 60 to 70. This is how we want a glissando to behave under transformation.

Another lift function is relative-lift. It does not help us to stretch a selection out of a function but it lets us express something else. Let us try it out as a variation on our last example.

```
(draw (note :duration 2 :pitch (lift linfunc (relative-lift 0 1)))
      :y-max 72)
```

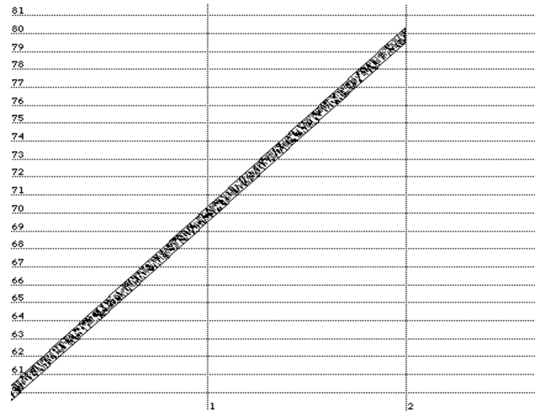


Figure 3.23: Example of a mathematical function lifted relatively

The behavior as result of this GTF is not the one we intend for a glissando, as you can see in Figure 3.23. But maybe you see where we *do* want to go with our example, because this is the kind of behavior we want to assign to a sine function to make it behave like a vibrato, for example. The arguments of `relative-lift` do not indicate an interval, but a starting point and an x-scale. This makes it easy to create a nice vibrato out of a sine. Like this (see Figure 3.24):

```
(draw (note :duration 2 :pitch (gtf+ 62 (lift #'sin (relative-lift 0 Pi)))
      :y-max 72))
```

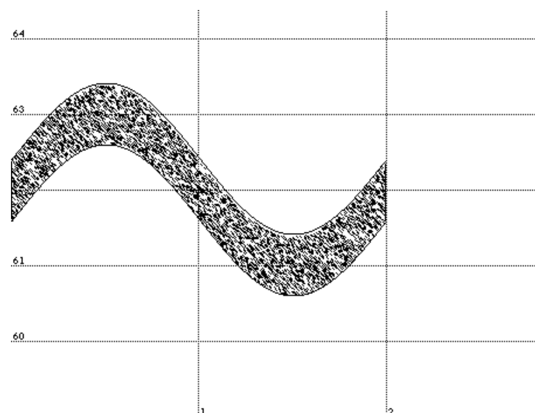


Figure 3.24: Example of a mathematical function lifted relatively

Now try to modify the duration of the note from the last example again and see what happens! Is it the behavior you desire from a vibrato?

The last lift function for simple functions we will discuss is `absolute-lift`. It is different from `relative-lift` in that it does not see the notes start as the starting point ($time = 0$) for the GTF, but it uses the start of the whole musical object as the starting point of the GTF. Let us illustrate this with a vibrato that is lifted using `absolute-lift` and that is used over a group of notes.

```
(draw (seq (sim
            (seq (note :duration 2 :pitch (gtf+ 60 lifted-sine))
                 (note :duration 1 :pitch (gtf+ 57 lifted-sine)))
            (seq (note :pitch (gtf+ 63 lifted-sine))
                 (note :pitch (gtf+ 64 lifted-sine))))
           (note :duration 2 :pitch (gtf+ 65 lifted-sine)))
      :y-min 55
      :y-max 70)
```

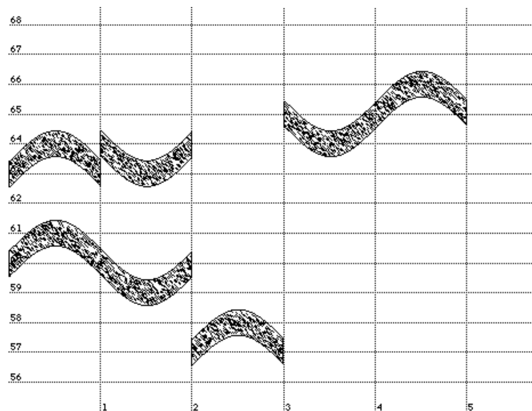


Figure 3.25: A group of notes, all with the same lifted sine function by use of `absolute-lift`

Because the starting point of the whole group of notes is taken as $time = 0$ you can see the sine wave continuously running through all notes, as if it does not know where one note ends and another begins.

Lifting tables

Another method of creating GTFs is by means of tables. You can specify a table with key and value pairs and lift it as if it were a mathematical function. An example is given below.

```
(setq my-table '(
             (0.0 60)
             (.1 61.4)
```

```
        (.2 62.3)
        (.3 61.2)
        (.7 65)
        (1 60)))
```

With the help of `table-fun` we can make a simple function out of a table:

```
(setq tablefunction (table-fun my-table))
```

Since `tablefunction` now is a simple function, we can use it in the same way as any other simple function. Hence we can lift it with the same constructions as demonstrated in the previous subsection. The function is made by a lineair interpolation between the values of the table, as demonstrated in the following code (see Figure 3.26):

```
(draw (seq (note :duration 2
                 :pitch (lift tablefunction (proportional-lift 0 .5)))
           (note :duration 2
                 :pitch (lift tablefunction (relative-lift 0 .5))))
      :y-max 66)
```

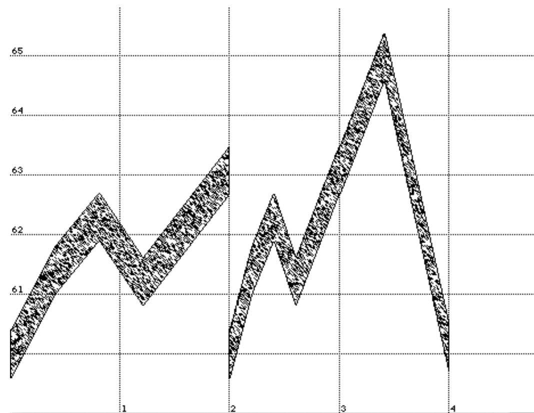


Figure 3.26: Example of a table, converted to a simple function and lifted

Creation of GTFs by writing your own Lisp code

In this last subsection on the creation of GTFs we will address you to write your own Lisp code as a means of creating your own GTFs. In order to do that we have to go into details of how a GTF is built up. We will do this, like you are used to in this tutorial, by means of example. First we will show how an existing GTF is built up. After that we will write a new one from scratch.

Do you remember the glissando GTF from all the previous examples? As we already said it is the same function as the `ramp` function. Here is its encoding in Lisp:

```
(defun ramp (from to)
  (gtf (start duration)                                    2
    (tf(time)
      (let ((progress (/ (- time start) duration)))        4
        (+ from (* progress (- to from)))))))
```

In general a GTF is built up using the construction

```
(gtf (start duration)
  (...
   (tf (time) (...))...)
```

`ramp` returns a function of the type $\mathbb{R}_+ \to \mathbb{R}_+ \to (\mathbb{R}_+ \to \mathbb{R}_+)$. The constructions `gtf` and `tf` work like normal functional abstraction. They are adapted versions of Lisps `lambda`, enhanced for our purpose here. `gtf` creates a function of the type $\mathbb{R}_+ \to \mathbb{R}_+ \to (\mathbb{R}_+ \to \mathbb{R}_+)$ and `tf` creates a funtion of type $\mathbb{R}_+ \to \mathbb{R}_+$. A GTF makes use of three important variables `time`, `start` and `duration`. The variable `time` indicates the time counted from the start of an entire musical object; `start` indicates the time at which the note in which the GTF will be used will begin; `duration` represents the time interval on which a note will be played. The dots on the second line of the general code are intended to consist of calculations with `start` and `duration`. Calculations that involve also `time` can only occur inside the `tf` construction. This two layer way of building a GTF is chosen because of efficiency reasons which we will not have to discuss at this stage. Look at section 6.3 for more on this. Because all of the calculations that we need in `ramp` involve `time`, they have to occur inside the `tf`.

Now let us look how ramp works. On line 4 of its code an expression with `time`, `start` and `duration` is bound to `progress`. If we look closer to the expression we see that it indicates the fraction of the duration that has already passed: "current time minus start time" divided by "total duration of the note". At the very start of the note, `progress` will have value 0. At the very end of the note "time minus start" will equal `duration` and therefore `progress` will have the value of 1. Now, with `(ramp <from> <to>)` we want to have a GTF that is a linear function from `<from>` to `<to>` on the interval `[start,start+duration]`. Note that the expression `(+ from (* progress (- to from)))))))` returns the right value on each point of time on in the interval.

Now that we have seen the implementation of `ramp` it may inspire us to make a different GTF. Wouldn't it be funny to make a GTF that gives random values from a certain range? In Lisp `(random <n>)` returns a random floating point number from the range $[0, n[$ (n self is excluded) when `n` is a floating point number. Hence `(random 1.0)` returns a floating point number between 0 and 1. The Lisp expression `(+ min (* (- max min) (random 1.0)))`

then returns a random value between `min` and `max`. This is all we have to return, so the code for `random-gtf` will be quite simple:

```
(defun random-gtf (min max)
  (gtf (start duration)
    (tf (time)
      (+ min (* (- max min) (random 1.0))))))
```

Let us see how `random-gtf` behaves as a pitch GTF:

```
(play-and-draw (note :pitch (random-gtf 61 62)))
```

Look at Figure 3.27 for the result!

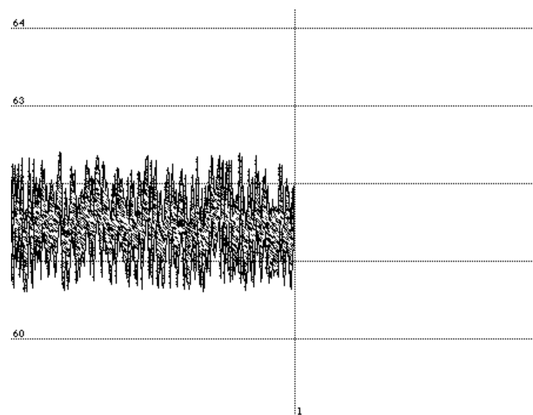


Figure 3.27: Example of the use of random-gtf

Of course we can also use `random-gtf` to make small random deviations

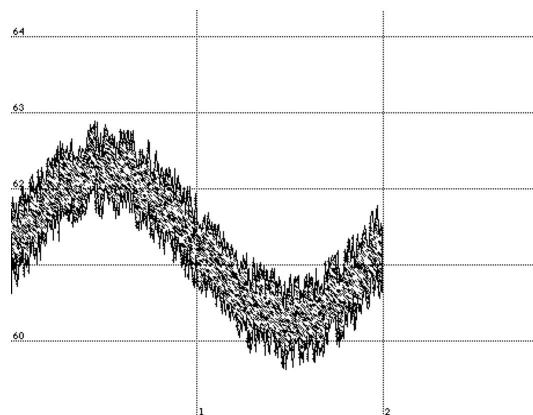


Figure 3.28: Example of random-gtf

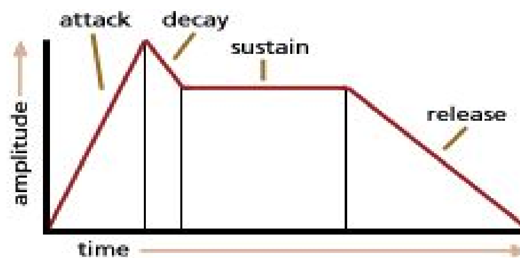
from other GTFs like in the following example (see Figure 3.28):

```
(draw (note :duration 2
            :pitch (gtf+ (vibrato :around 61)
                         (random-gtf 0 .5))))
```

.

Case study: ADSR

To conclude this tutorial, we will give you a description of how to build a GTF that behaves like an instrument. This GTF will be build up in analogy of a concept in the synthesizer world, namely ADSR. ADSR stands for *attack, decay, sustain* and *release*. The idea is that when a note starts, the volume of the note will go from 0 to a certain attack level. Immediately after reaching the attack level the volume will go down in a certain amount of time, till it reaches the sustain level. This level will be maintained for a certain length of time. Just before the end of the note (think of a key being released) the level will go from sustain level to 0 again. This is illustrated in Figure 3.2. We will discuss a simplified version of this idea for the sake of didactics and the length of this tutorial.



From the above description of ADSR it might be evident that we need four basic GTFs which we can concatenate later by means of `concat-gtf`. Conceptually this may look something like this:

```
(concat-gtf
 attack function                                    2
 time switch
 decay function                                     4
 time switch
 sustain function                                   6
 time switch
 release function)                                  8
```

We need a GTF for going from 0 to an attack level in a certain amount of time. This amount of time does not in- or decrease with the length of a note for most instruments (loosely speaking); it is a constant portion of time. Therefore, the attack function, let us call it `attack-fun`, must precede a time switch that is made with `relative`. The same goes for the decay

function, which we will call `decay-fun` and its succeeding time switch. The sustain, named `sustain-fun`, does depend on the duration of the note. The portion of time needed for the release, let us call it $release - fun$, from sustain level to 0 is again a constant portion. With this information we can make the concept a bit more concrete:

```
(concat-gtf
 attack-fun                                                    2
 (relative ...)
 decay-fun                                                     4
 (relative ...)
 sustain-fun                                                   6
 (relative 'end time minus' ...)
 release-fun)                                                  8
```

Let us specify the different functions a little more. We propose the following as an attack function:

```
(defun attack-fun (begin end)
  (gtf+ begin
   (gtf* (- end begin) (lift (sine 4) (proportional-lift 0 1)))))
```

The function `attack-fun` expects a `begin` and `end` value. In the following example you get an impression of how it works (see Figure 3.29):

```
(draw (sim (note :pitch (attack-fun 60 65))
           (note :duration 2 :pitch (attack-fun 62 70)))
      :y-max 72)
```

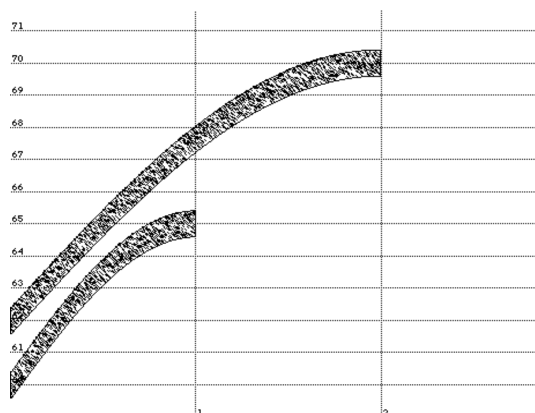


Figure 3.29: Example of attack-fun

For the decay function we propose the following:

```
(defun decay-fun (begin end)
  (time-fun-compose #'- begin
                    (gtf* (- begin end) (lift (sine 4) (proportional-lift 0 1)))))
```

We illustrate this function with the following example (see Figure 3.30):

```
(draw (sim (note :pitch (decay-fun 65 62))
           (note :duration 2 :pitch (attack-fun 70 64)))
      :y-max 72)
```

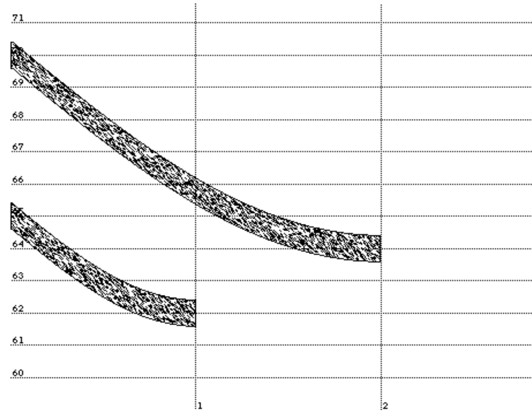


Figure 3.30: Example of decay-fun

The sustain function can be really simple. It just has to remain constant at a certain level:

```
(defun sustain-fun (volume)
  (gtf (start duration)
    (tf (time)
    volume)))
```

As a release function we can take the same function as the decay function:

```
(defun release-fun (begin end)
  (decay-fun begin end))
```

Finally we are ready to try out our ADSR. We build a function around it in which you can specify the different levels and time intervals.

```
(defun adsr (attack-time decay-time release-time attack-level sustain-level)
  (let ((att-time (relative attack-time))                          2
        (dec-time (relative (+ attack-time decay-time)))
        (rel-time (fun-funcall #'- #'end-time release-time)))      4
    (concat-gtf (attack-fun 0 attack-level)
                att-time                                           6
                (decay-fun attack-level sustain-level)
                dec-time                                           8
                (sustain-fun sustain-level)
                rel-time                                           10
                (release-fun sustain-level 0)
                )))                                                12
```

The expression `(fun-funcall #'- #'end-time release-time)` evaluates to the end time of the note minus the time interval indicated by `release-time`. Because `end-time` and `release-tim` are not normal numbers (actually they are functions) we have to use `fun-funcall` in this case.

Now let us try out `adsr`. We define two different ADSRs named `inst1` and `inst2` as follows and use both in a group of notes:

```
(setq inst1 (adsr .1 .4 .2 .9 .7))
(setq inst2 (adsr .2 .2 .4 .4 .2))

(draw
 (sim
  (for-all-notes :amplitude inst1
                 (seq  (note :pitch 62 :duration 1)
                       (pause 1)
                       (note :pitch 63 :duration 2)))
  (for-all-notes :amplitude inst2
                 (seq (note :pitch 63 :duration 2)
                      (note :pitch 62 :duration 1.5)
                      ))))
```

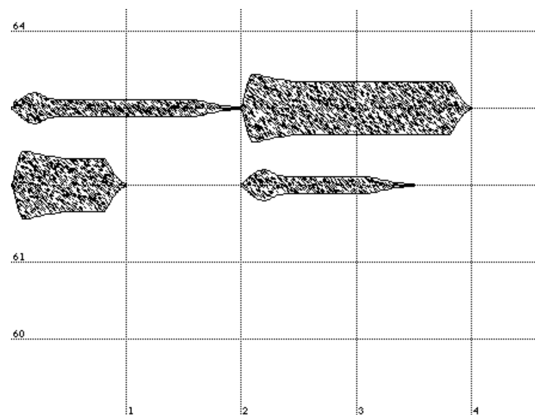


Figure 3.31: Example of the use of `adsr`

You see that the only the length of the sustain phases depend on the lengths of the notes. The other phases are of constant length, no matter what the lengths of the notes are.

In the following graphs we show both ADSRs plotted in time against durations of lenghts from the interval $[0, 3]$ in seconds.

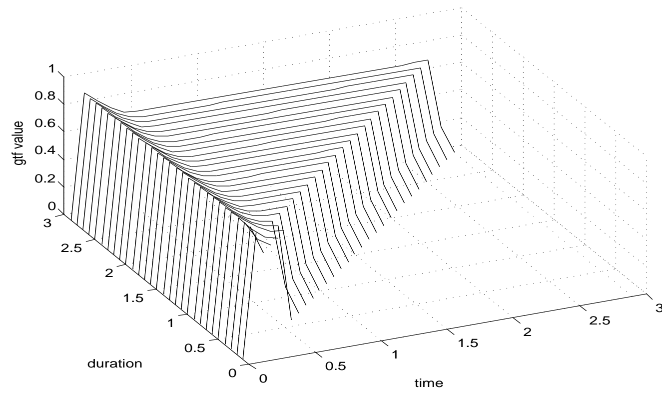


Figure 3.32: The GTF `(adsr .1 .4 .2 .9 .7)` plotted for durations from 0 to 3 seconds

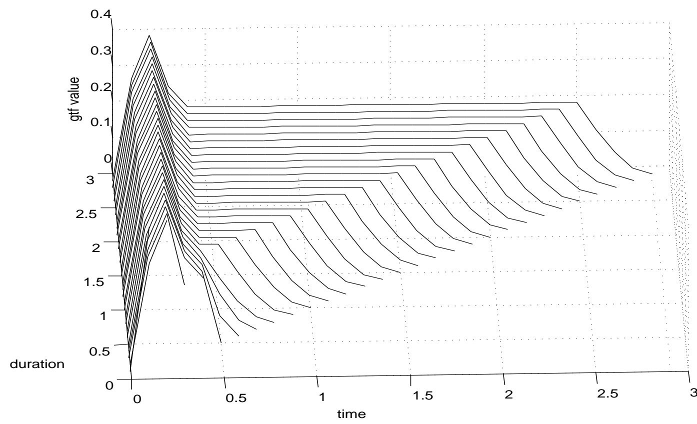


Figure 3.33: The GTF `(adsr .2 .2 .4 .4 .2)` plotted for durations from 0 to 3 seconds

Chapter 4

Overview of NoteFun's functions

In this chapter we give an overview of the functions that a NoteFun user can use to compose music. Each function name will be given, followed by the page number where the use of the function is explained. Possibly some additional comments are given. For all the examples of the use of these functions we redirect you to chapter 3.

4.1 Construction of musical objects

This section is an overview of the functions that allow you to build musical objects.

pause The most simple musical object is a pause. More information on page number 10.

note The most basic musical object next to `pause` is a note. It is described on page number 10 and further.

:program With this aspect of a note one can choose the program number should be used when the note is played on a synthesizer. More information on page 12.

:channel With this aspect of a note one can choose the channel number on which a note should be played. More information on page 12.

for-all-notes This is a construct to assign an aspect to a group of notes. More information on page **??**.

seq This function is used to make sequences of musical objects and is explained on page 11.

sim This function is used to make simultaneous compositions of musical objects and is explained on page 11.

trans With `trans` one can transpose an entire musical object. More information can be found on page 12.

stretch With `stretch` one can stretch an entire musical object. More information can be found on page 13.

4.2 Construction of GTFs

glissando This is a predefined GTF. More information on page 13.

vibrato This is a predefined GTF. More information on page 14.

ramp This is a predefined GTF. More information on page 22.

proportional This functions allows to express a point in time as a proportion of a note's duration. More information on pageref 18.

relative This functions allows to express a point in time as relative to the note's start. More information on pageref 19.

absolute This functions allows to express an absolute point in time. More information on pageref 19.

concat-gtf This functions allows to concatenate multiple GTFs as described on page 17.

switch-gtf This functions allows to combine and switch between multiple GTFs as described on page 23.

mix-gtf A function to mix GTFs as described on page 26.

interpolate-gtf A function to interpolate between GTFs, described on page 27.

gtf+ A function to sum GTFs, described on page 28.

gtf* A function to multiply GTFs, described on page 28.

time-fun-compose A function to compose GTFs using mathematical operators as described on page 28.

lift A function to create a GTF from a mathematical function, described on page 30.

proportional-lift A lift function used within `lift` as described on page 30.

relative-lift A lift function used within `lift` as described on page 30.

absolute-lift A lift function used within `lift` as described on page 32.

table-fun A function that creates a function from a table, described on page 33.

gtf The construction to create a GTF, as documented on page 34.

tf The construction that is used within `gtf` to create a GTF, as documented on page 34.

time-reverse A time warp, described on page 24.

make-time-cyclic A time warp, described on page 25.

4.3 Mediate musical objects

Show musical objects

draw This function shows a graphical representation of a musical object.

Play musical objects

play `(musical-object)`
The musical object will be translated into a midi sequence which will be interpreted by a connected midi synthesizer. A more detailed description can be found on page 12.

add-midi-controller Notes: the name of a midi controller, indicated by `name` should have the form of a Lisp keyword, preceded by a colon :. Optionally a range indicating the minimum, default and maximum of a controller can be given by giving three integer values `min`, `def` and `max`. By specifying a midi specific controller with e.g., the name `:brightness` it can be used in a note similar to a normal property of a note, like `:pitch`. A detailed example can be found on page 22.

remove-midi-controller With this function a synthesizer specific midi controller can be removed from the controller list. More information can be found page 23.

show-midi-controllers With this function the controller list can be printed on the screen.

Chapter 5

Other programming languages compared to Lisp

5.1 Introduction

Only less than four months ago I started to learn to program and think in Lisp. It was an educative time in which I grew enthusiastic about Lisp. But at times it was also a frustrating time in which I felt negative towards the language. In what follows I will try to stipulate how Lisp relates to other programming languages. I will try to give an objective-as-possible overview of the pro's and cons of Lisp in relation to other programming languages and their usages. I hope it will be clear that Lisp is a language that deserves serious consideration as language of choice in both software engineering and Artificial Intelligence research. Also I hope to clear up some myths around Lisp and declarative languages in general. In chapter 6 we will see a practical illustration of the characteristics of Lisp within the NoteFun project.

5.2 Why Lisp?

During the past few months I asked a lot of people who are active in the field of computer science why they preferred their language of choice over others. Together with my own knowledge and experience, certain characteristics of Lisp and the other languages became evident and need to be mentioned in this section. By means of these characteristics I will try to give an answer to the question: "Why Lisp?" In Norvig (1992) page ix, I found a summary of some important aspects of "Why Lisp?" These three more or less categorize the issues I would like to discuss. First I will give Norvigs three arguments.

Later I will discuss each of them in more detail and I will contrast them with possible counter-arguments. The three arguments Norvig names are listed here:

1. Lisp is a very popular language for AI programming. If you are going to learn a language, it might as well be one with a growing literature, rather than a language which is not widely used anymore and is doomed to be one of dead tongue.

2. Lisp makes it easy to capture generalizations in defining new objects. In particular, Norvig says, Lisp provides the means to define new languages that are problem specific. This is handy in AI applications which often manipulate complex information which is best represented in some novel form.

3. Lisp makes it very easy to develop a working program fast. Lisp programs are concise and are uncluttered by low-level detail.

Ad 1

I think this is a good argument for picking a language, but it might not be the strongest one to defend Lisp. Until a few months ago I rarely ever heard about Lisp. A class about parallel garbage collection was an exception in this case, but even there I did not see a fragment of Lisp code. Lisp is a popular language in the AI world, but particularly in the United States. Most of the people in Europe, where I happen to live, who I spoke to and did know about Lisp, were kind of skeptical about Lisp similar in the way people are skeptical about the artificial human language Esperanto nowadays. Where does this skepticism come from? During the past few months I got the feeling that, like Esperanto, Lisp has a hidden value that people must have forgotten over the past few decades or simply have not discovered yet.

Parentheses scary?

The reason of this may lay in the fact that at first glance both languages make a weird impression; they take some time to get used to, both in reading, thinking and writing. For example, a often-heard critic on the Lisp syntax is: "Too much parentheses!" When I think back of my history in programming in other languages, I think the parentheses in Lisp are no more a burden than the curly bracket pairs and semicolons we know from

e.g., Pascal and Java. Especially with a good editor[1] the parentheses and indentation do not form a big problem anymore.

Familiarity

One other reason could be that programmers nowadays more or less take confidence in only one programming style: either imperative, object oriented (or a mix of the latter two) or declarative (logical and functional). This could be simply out of lack of knowledge of other possible styles. Also it could come from the culture in which the programmer was educated. Stubborn propaganda sometimes is made for one specific paradigm which can have a huge impact on how the programmer thinks about other languages. I see this not only at universities, but also at companies where sometimes tradition is the main factor in choosing a language.

Hybrid style: good or bad?

Another reason for suspicion is that Lisp combines a lot of paradigms. How can a language that is a mix of paradigms be a clean and concise language? In this respect it resembles Esperanto too, but in a different way. Esperanto is more or less a compromise between existing european natural languages. Lisp, on the other hand, is *not* a compromise between existing programming languages. It truly incorporates the full power of imperative, functional and object oriented programming. This should not be something to be scared off by. It should be considered as a challenge to travel beyond the boundaries of one paradigm!

Specific or general purpose?

A myth that still lives among imperative programmers is that declarative languages are problem dedicated languages and that their language is one of general purpose. Clearly this is not the case. The imperative languages reflect the inner state of a computer system. They do not reflect the way how we would mathematically, logically and intuitively describe a problem. Other than doing that, an imperative program consists mainly out of (possibly parallel) sequences of instructions. Some, if not most of these instructions describe how variables should be updated in the next iteration of the program. Most of these imperative languages are nothing more than a high level description of what the resulting machine code should look like.

[1] Like the editor that goes with MCL: Fred

This is not what I call a general purpose language because it has specialized task: to make machine code more readable and writable.

Side effects and reasoning about programs

An ideal that comes from the declarative programmer's society is that code should be *clean*. An interpretation of this statement is that by means of the code one should be able to reason about the results of the program or one should be able to prove the correct workings of the program by means of the code only. A consequence of this is that code should be *side effect free*. Any computation should not change the global state of other computations. Programming languages that do quite the opposite are the imperative languages described above. Therefore they are not regarded as suitable for e.g., explaining algorithms and proving their correctness. Often the syntax of an imaginary pseudo language or a purely declarative language is used for this purpose. Sometimes declarative languages therefore also are called *executable specification languages*. Lisp combines both side-effect-free and state-changing programming. For this reason Lisp might not be regarded as a pure and clean language by theoreticists and therefore will be put aside by them. An objection to this argument can be that a language is either at the same time really side-effect-free and useless or not really side-effect-free. Please note that reading from a file or keyboard and writing to a file or screen are state-changing operations. If a running program would totally have no state-changing properties, how would we ever notice that the program actually works correctly? There would be nothing in the world changed for us to be remarked. Another objection is that Lisp can be used in the same side-effect-free way of programming like pure functional languages. In this case, one simply must not make use of state-changing constructions.

Survival of the fittest?

Norvig points out that Lisp is an old language. Indeed it is. Its origins lay in the thinking of McCarthy, a Princeton student of Alonzo Church. McCarthy invented Lisp in 1958. Many people think this language has survived many others because of its flexible approach to self-reference and extension. In Lisp data and programs reside on the same level: both are objects built as a list, the key structure in Lisp. Therefore both can be treated as data and programs. This makes the alteration of programs within programs possible. This approach is widely exploited within macro's. A macro is used to generate Lisp program code, possibly by alteration of a

program passed as an argument. This might be one of the most important reasons why Lisp has made it until this day. By means of macro's other programming paradigms could be incorporated within Lisp. This makes Lisp distinct from other existing programming languages, both imperative and declarative.

Ad 2

Dynamical typing and regularity in representation

This characteristic of Lisp is something that is handy in not only AI applications but, I think, everywhere a program is being designed and developed. I have experienced this beneficial property of Lisp throughout the NoteFun project. As no other language I worked in previously, Lisp allowed me to be flexible and generalizing over data and even programs by means of macro's. One of the reasons of this flexible way of working may be that Lisp is a dynamical typed language. Another may be that all data and programs represented in Lisp are represented by a list. Dynamical typing and syntax regularity make it possible to write very general operations on data and programs which can be used in a wide variety of situation. This is unlike I am used to in statical typed languages where every data type must be accompanied by a set of functions to create, manipulate and destroy them, in order to work with them in a satisfying way. In these languages some time can be saved by means of overloading and generic programming, but still work has to be done explicitly for every type.

Ad 3

Code esthetics

In my opinion Lisp code is uncluttered by low-level detail, although not optimally. There still is some cluttering which probably scares people off here and there. I will give two examples that show how Lisp code is not as uncluttered as possible.

```
(setq func #'(lambda(x)(* 2 x)))
(funcall func 2)
```

On the first line we see an assignment of an anonymous function, a so called lambda expression, to the name `func`. In Lisp we have to make use of quotations to separate between data and programs. With a `'` we indicate that a list is meant as data and should not be evaluated. The `#` adds an indication that the object we are quoting is a function. This is something that might be regarded as ugly and cluttered syntax. On

line 2 we try to apply the function to an argument. In this case we have to use `funcall`. If the syntax would be clean I would at least expect that we could do something like this: `(func 4)`. This is supported in the Lisp variant called Scheme but not in Common Lisp. This is something I would like to see changed to the Lisp syntax, although I do not know what new problems it might introduce. Lisp programs, as Norvig states, are often easy to read and do not have many details that do not directly relate to the programming problem. But there are some other languages of which the syntax is more beautiful in my opinion. I will draw a comparison with the functional programming language *Amanda* here:

In Lisp, a new function that is constructed from a function by applying it twice, can be made as follows:

```
(defun double (fun)
  #'(lambda(arglist)
  (funcall fun (funcall fun arglist))))
```

To create and use a function that is twice the application of the function $f(x) = x^2$ we can write the following:

```
(funcall (double #'(lambda(x) (* x x))) 2)
```

The result is 16.

In Amanda we could do all the same with:

```
double f = args -> f (f args)
double (x->x^2) 2
```

Of course the result is also 16. It is to you to decide, but I think Amanda looks better in this case. Norvig is right that Lisp code is uncluttered and without low-level details, compared to languages like C, Pascal, Java. But I think the comparison cannot be drawn towards languages like Amanda, Haskell and Clean.

Debugging

As mentioned in the previous subsection the dynamical typing in Lisp brings an advantage in favor of rapid prototyping. But alas, it also comes with a disadvantage: it can make debugging much more harder. Error messages generated as a result from a programming mistake can occur at the bottom of execution. Therefore error messages sometimes do not get us quickly to the "place of crime" but can take us into a long and frustrating crusade.

Interactiveness

A last issue, this time in favor of the third argument, I would like to point out is that Lisp is an interactive language. By means of the REPL, the

Read-Evaluate-Print-Loop, code can be tested immediately. This can be contrasted against the way of working in other languages in which one often has to build an environment for reading and printing and compile complete programs in order to test. The interactive property of a Lisp environment speeds up the way of working considerably. During my internship I used an interactive editor in which one can select a fragment of code which can be executed after pressing a certain key. This is even a better alternative for a prompt at which one can insert code for execution and it helps to work even faster.

5.3 Conclusions

In this chapter I contrasted Lisp with other languages. Three claims about why Lisp ought to be a programming language of choice were discussed. In the discussion several characteristics of Lisp arose:

- Lisp is not as weird as it may seem: it is merely a question of familiarity.
- Lisp combines programming paradigms
- Lisp is a general purpose language
- Lisp can be used as a side-effect-free language
- By means of macro's Lisp survived many languages and can be accustomed to your programming style and paradigm
- In Lisp, both data and programs are represented in the same way. Regularity (and dynamical typing) provides us with a means to make strong generalizations.
- Lisp syntax is uncluttered and concise. Compared to a lot of (imperative) languages it is better to read, although there exist other languages that read even better.
- Lisp is an interactive programming language.

Chapter 6

Design and Implementation

In this chapter we will discuss some interesting cases from the process of designing and implementing NoteFun. During this chapter we assume the reader has a basic knowledge of functional programming and hopefully Lisp in specific. We will discuss some items that we found useful to demonstrate here for a number of reasons. One of the reasons is to show what important design decisions have been made. Another reason is to show where and why the characteristics of Lisp played an important role.

6.1 Notes on functional programming style

During the design and implementation of NoteFun we often discovered the usefulness of the functional programming paradigm. Having functions available as first class citizens often led to a better generalized design. To give a concrete example of this we will describe how we used the functional style in the design of `concat-gtf`.

When we first toyed with the idea of `concat-gtf` we constructed a function that had an argument list that consisted of GTFs and numbers expressing points in time. For example users could do something similar to this:

```
(concat-gtf (ramp 0 1) .1 (ramp 1 0) 3 (ramp 0 1))
```

We thought it would be handy if users could express points in time relative to the start of a note, as a proportion of the note's duration or just as an absolute point in time. Somehow these numbers had to be combined with an indication of what kind of time was meant. Then the idea arose to use a more functional way of expressing points in time. We introduced the following way the could be used in any function needing an argument that expresses a point in time:

- Absolute points in time n can be expressed like: `(absolute n)`

- Points in time n relative to the beginning of a note can be expressed like: `(relative n)`

- Points in time n as a proportion of the note's duration can be expressed like: `(proportional n)`

Now users had to be able to do things like:

```
(concat-gtf (ramp 0 1) (proportional .1) (ramp 1 0) (absolute 3) (ramp 0 1))
```

as is the case in our definitive implementation. Implementation-wise these expressions of time are functions that themselves have anonymous functions as a result. Let us take `proportional` as an example here:

```
(defun proportional (proportion)
  (time-expr (start duration)
    (+ start (* proportion duration))))
```

The construction `time-expr` is equivalent to Lisp's `lambda` with some additional error catching. The result of `proportional` thus is an anonymous function of two arguments, `start` and `duration`. When called with the start and duration of a note, this anonymous function returns a point in time between the start and end of the note, according to the proportion that is given as a numerical argument to `proportional`.

Because `absolute`, `relative` and `proportional` all have anonymous functions as a result of the same arguments and results, these anonymous functions can be treated alike by `concat-gtf`. The way of specifying a point in time is directly associated with the corresponding way of calculating the point in time, because the way of specifying is a Lisp expression itself. The calculation is delayed, by means of creating anonymous functions, until a certain moment on which the real `start` and `duration` of a note are known. This way of programming is only possible in a functional programming language, a language in which functions are first class data objects: they can serve as arguments and results of other functions.

6.2 Extending the language by use of macro's

One of the useful characteristics of Lisp is that it is an extensible language. Unlike languages like Pascal and Java, Lisp can be augmented with language constructions. A simple example is the `while` loop that is not standardly included in Lisp. A macro for adding it to the language is given below (taken from Norvig (1992), page 67):

```
(defmacro while (test &rest body)
  (list* ’loop
         (list ’unless test ’(return nil))
         body))
```

We will not go into a deep level of detail here. For a full explanation of this macro and macro’s in general please read the section about macro’s in Norvig (1992), pages 66 to 68. What happens when we use this macro with a test, a piece of code that results in to t or nil, and a body, a block of executable Lisp code, is that the result is Lisp code that is understood in terms of normal Lisp code. The following translation takes place:

(while <test> <body>) → (loop (unless <test> (return nil)) <body>) This explains the back-quotes you see in the code. The back-quotes indicate that the word following it should be literally generated. This also explains why we don’t see a preceding back-quote before test and body, namely because they are anonymous pieces of Lisp code.

An example of the use of while:

```
(let ((n 0))
  (while (< n 5)
    (progn
      (print n)
      (incf n))))
```

and the result:

```
0
1
2
3
4
NIL
```

With macroexpand-1 we can see the macro-expansion of a macro, which is the code that is generated as a result of the use of a macro. Applied to our example above,

```
(macroexpand-1 ’(while (< n 5)
                    (progn
                      (print n)
                      (incf n))))
```

gives

```
(LOOP
  (UNLESS (< N 5)
    (RETURN NIL))
  (PROGN (PRINT N) (INCF N)))
```

One can imagine that the while construction is a very handy addition to the language. During my internship I have written a macro that is somewhat more complex but turned out to be very useful in different situations.

It is an iterative construction very similar to Lisp's `dolist`. Instead of iterating over every single element in a list, our macro makes it possible to iterate over sets of neighboring elements in a list, which we call a window. The language construction is called `do-window-list` and it is demonstrated in the examples below.

For the sake of the example we will first define the Fibonacci sequence:

```
(defun fib (n)
  (cond ((equalp n 1) 1)
        ((equalp n 2) 1)
        (t (+ (fib (- n 1)) (fib (- n 2))))))

(defun fibonacci (n)
  (cond ((equalp n 1) (list 1))
        (t (append (fibonacci (- n 1)) (list (fib n))))))
```

The function `fibonacci` returns the first `n` elements of the Fibonacci sequence, where `n` is the argument of `fibonacci`.

```
(fibonacci 10)
```

results in:

```
(1 1 2 3 5 8 13 21 34 55)
```

Suppose that we would want to have the product of every three succeeding numbers in this list. With `do-window-list` this can be easily done:

```
(do-window-list (window 3 (fibonacci 10))
  (apply #'* window))
```

and the result is:

```
(2 6 30 120 520 2184 9282 39270)
```

In the above code fragment `window` is an arbitrarily chosen variable name which can be used in your program code to refer to the window that has a given size and iterates over a given list. In this case the window size 3 and the list over which is iterated are the first 10 elements of the fibonacci sequence. In your program code you can do anything you like with the windows. Suppose that we only want to give the product of the window of which the first element is 13, then the following code would seem to be right:

```
(do-window-list (window 3 (fibonacci 10))
  (if (equalp (first window) 13)
    (apply #'* window)))
```

but as we might expect the result is the following:

```
(NIL NIL NIL NIL NIL NIL 9282 NIL)
```

Instead only the number `9282` as a result would be desirable. This is why we implemented `do-window-list` with keyword `ret-all` which is set to `t` by default.

One can indicate with it whether all the elements should be returned or if only the first element that is not `nil` should be returned. In this case the latter would seem appropriate:

```
(do-window-list (window 3 (fibonacci 10) :ret-all nil)
  (if (equalp (first window) 13)
    (apply #'* window)))
```

and the result is, as we desired:

```
9282
```

In my internship I have implemented many versions of `concat-gtf`. The most recent version makes use of `do-window-list`. Its code is given here:

```
(defun concat-gtf (&rest args)
  (gtf (start duration)                                          2
    (let* ((times (append
                   (cons                                         4
                    #'start-time
                    (loop for x in (rest args) by #'cddr collect x)) 6
                   (list #'end-time)))
           (calc-times                                           8
            (mapcar
             (tf (time)                                          10
               (funcall-maybe time start duration)) times))
           (monotonized-times (monotonize calc-times))           12
           (new-args
            (zip                                                 14
             monotonized-times
             (loop for x in args by #'cddr collect x))))         16
      (tf (time)
        (destructuring-bind (t1 gtf t2)                          18
                            (do-window-list
                              (window 3 new-args :skip 2 :ret-all nil) 20
                              (if
                                (in-between                      22
                                 time
                                 (first window)                  24
                                 (first (last window)))
                                window))                         26
          (tf-call (drop gtf t1 (- t2 t1)) time))))))
```

See page 17 to see how the argument list for `concat-gtf` looks. In lines 3 to 7 functional expressions representing the start time and the end time are respectively consed and appended to the argument list. In line 8 to 11 the functional representations of switch points are used to calculate the numerical times, which are monotonized on line 12. On line 13-16 a new argument list is formed which will be used for processing in the code that follows. The application of `do-window-list` is done on line 19 to 26. We iterative over every time-gtf-time subsequence in the new argument list, until the right one, the one to which the current time is related, is found. `:skip 2` indicates that the window should not move one place to the right after every iteration, but instead two places. The first window that concerns

the current time is returned and used for calculation on line 26. In more or less the same way `do-window-list` is used in `switch-gtf`. Another helpful application of `do-window-list` can be found in the definition of `table-fun` which turns a table into a function.

```
(defun table-fun (table)
  (apply #'combine-funs
   (do-window-list (window 2 table)
    (let ((win1 (first window))
          (win2 (second window)))
      (make-lin-fun (first win1) (first win2) (second win1) (second win2))))))
```

Again, we will not go into full detail here. Enough is to say that `combine-funs` combines the domain/value pairs of multiple functions into one function and that `make-lin-fun` returns a linear function that resembles a line segment as the connection of the points $\{(x_1, y_1), (x_2, y_2)\}$ where `(x1 x2 y1 y2)` forms the argument list of `make-lin-fun`. The linear function that is created is only defined on the interval $[x_1, x_2]$. The representation of a table in NoteFun can be found on page 32. As an exercise, I leave it up to the reader to figure out how `table-fun` creates a function out of a table.

For the interested reader, without further explanation, here is the code for `do-window-list`:

```
(defmacro do-window-list
          ((varsym window-size list &key (skip 1) (ret-all t))
           &body body)
  (if (not (symbolp varsym)) (error ''Not an appropriate symbol name.''))
  (let ((ivar (gensym))
        (lvar (gensym))
        (wvar (gensym))
        (skp (gensym))
        (result (gensym))
        (retall (gensym)))
    `(loop with ,wvar = ,window-size
           with ,lvar = ,list
           with ,skp = ,skip
           with ,retall = ,ret-all
           for ,ivar from 0 to (- (length ,lvar) ,wvar) by ,skp
           as ,varsym = (subseq ,lvar ,ivar (+ ,ivar ,wvar))
           as ,result = (progn ,@body)
           if ,retall
           collect ,result
           else unless (null ,result) return ,result)))
```

6.3 One layered versus two layered GTFs

An important design decision in NoteFun was the alteration of the definition of a GTF. We started off with a function definition of a GTF that had three arguments, `start`, `duration` and `time` (see Listing 6.1). During the development of NoteFun we have changed this function definition into two nested function

definitions. The function on the highest level is a function of only `start` and `duration` and the function on the lowest level is a function of one argument, `time` (see Listing 6.2).

Listing 6.1: Old way of defining GTF

```
(gtf (start duration time)
 .... all code needed in a GTF
))
```

Listing 6.2: New two layered way of defining GTF

```
(gtf (start duration)
 .... ;;; code concerning calculations with start and duration
  (tf (time)
   ..... ;;; code concerning also time (and possible start and duration)
))
```

We made this decision because of efficiency reasons. When playing and drawing notes with GTFs attached, for every moment in time a value is calculated by means of a GTF that is constructed by the arguments `start`, `duration` and `time`. In the old situation the whole GTF would have to be calculated from scratch, because the `time` argument differs every other moment in time and possibly affects the whole body of a GTF. In the new implementation calculations concerning only `start` and `duration` and those who also concern `time` are separated in two layers. The reason for this is that throughout a note `start` and `duration` do not change. Calculations only concerning these two only have to be carried out once.

To demonstrate the improvement in efficiency we have taken a function that decides the minimum and maximum value of a GTF by taking samples in steps of 0.01 seconds between `start` and `duration`.

Listing 6.3: Demonstration of old method

```
(defun min-max-gtf-old (gtf start duration)
  (list                                                      2
   (loop
     for time from start by *resolution* to duration         4
     minimize (funcall-maybe gtf start duration time))
   (loop                                                     6
     for time from start by *resolution* to duration
     maximize (funcall-maybe gtf start duration time))))     8
```

Listing 6.4: Demonstration of new method

```
(defun min-max-gtf (gtf start duration)
  (let ((calc-gtf (drop gtf start duration)))                2
    (list
     (loop                                                   4
       for time from start by *resolution* to duration
       minimize (tf-call calc-gtf time))                     6
     (loop
       for time from start by *resolution* to duration       8
       maximize (tf-call calc-gtf time)))))
```

In Listing 6.3 the old method of using an old definitionof a GTF is demonstrated. Assume that `*resolution*` is a global variable with value `0.01`. In the first loop the minimum of all the samples is decided and in the second loop the maximum of all the samples is decided. The samples are taken by calculating the GTF with `start`, `duration` and `time`. Note that if the interval indicated by `start` and `duration` is n seconds, then $100 \times n + 1$ samples will be taken and an equal amount of times the GTF has to be calculated.

Let us take an example of a GTF and draw a couple of notes with it (see Figure 6.1):

```
(setq gtftest (gtf (start duration)
  (let ((val (sin duration)))
    (tf (time)
      (* time val)))))

(draw (sim (note :pitch 60 :duration .75 :amplitude gtftest)
           (note :pitch 62 :duration 1 :amplitude gtftest)
           (note :pitch 64 :duration 1.25 :amplitude gtftest)))
```

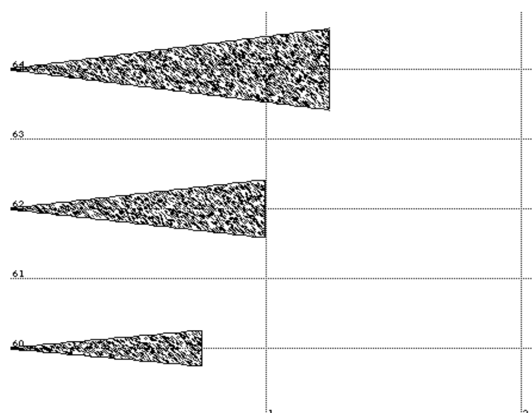


Figure 6.1: Three notes with `gtftest` as the GTF for amplitude

The old way of defining the GTF would have been like this:

```
(setq gtftest-old (lambda (start duration time)
      (* time (sin duration))))
```

Of course both ways of calculating GTF values work:

```
(min-max-gtf gtftest 0 1)
(min-max-gtf-old gtftest-old 0 1)
```

and their results are the same:

```
(0.0 0.833056274959818)
?
(0.0 0.833056274959818)
?
```

It is only the efficiency in which these calculations are carried out that differ. Note that in the one second interval 101 calculations will be carried out. Note that in the old method the expression with the sine in it has to be calculated 101 times. On line 2 of Listing 6.4 the GTF is calculated once with `start` and `duration` and the result is a function of only `time`, represented by:

```
(tf (time)
 (* time 0.8414709848078965))
```

in which no expression makes use of a sine function, because this was already calculated. This function of time is now used 101 to calculate the sample values. As you can imagine the time function is more efficient to use than `gtftest-old`. You can also imagine that with more complex GTFs and longer time intervals the benefit in efficiency will only increase.

6.4 Midi output

During the design and implementation of NoteFun the desire arose to hear the musical objects that could be constructed with it. This is why a translation of NoteFun objects to midi sequences was an evident thing to implement. A midi sequence is a serie of midi messages that can be interpreted by a synthesizer. Examples of midi messages are commands that select a program number and messages that set a note on and off on certain midi channels. Different notes can sound on different channels, each channel having its own program number; multiple instruments can sound on the same moment. For a full description of the midi protocol we direct you to read The complete MIDI 1.0 Detailed Specification.

Translation of pitch related GTFs

To be able to play objects from NoteFun means to be able to have continuous control of the pitch, volume and other aspects which are involved in a note. In midi the pitch of a note can be controlled by a note number. For instance, midi note number 60 corresponds with the pitch of the middle C: $261.63Hz$ and midi note 61 corresponds with the succeeding semitone of the middle C, a C-sharp of $277.18Hz$. Furthermore pitch can be controlled by means of pitch bend. Every synthesizer has a certain range in which the pitch can be deviated from the original frequency of a note. This is done by setting the range first or using the default pitch bend range. Then a note has to be set on on a certain channel, after which pitch bend command can be send to deviate from the note's original pitch. This is the method we used to

project GTFs, that are used as a note's pitch, to actual sound. We will now show a little example of a glissando and its corresponding midi sequence.

The example is a very short note with a glissando from note 60 to 61. We have taken a short note to keep the length of the corresponding midi sequence relatively short.

The NoteFun code look like this: `(note :duration .1 :pitch (glissando 60 61))` and the corresponding midi sequence looks like this:

```
0.000 begin
0.000 midi  (176 100 0)
0.000 midi  (176 101 0)
0.000 midi  (176 6 48)
0.000 midi  (192 80)
0.000 midi  (224 0 64)
0.000 midi  (176 7 102)
0.000 midi  (144 60 127)
0.010 midi  (224 34 64)
0.020 midi  (224 68 64)
0.030 midi  (224 102 64)
0.040 midi  (224 8 65)
0.050 midi  (224 42 65)
0.060 midi  (224 76 65)
0.070 midi  (224 110 65)
0.080 midi  (224 17 66)
0.090 midi  (224 51 66)
0.100 midi  (224 85 66)
0.100 midi  (128 60 127)
0.100 end
```

A graphical representation of this note can be seen in Figure 6.2.

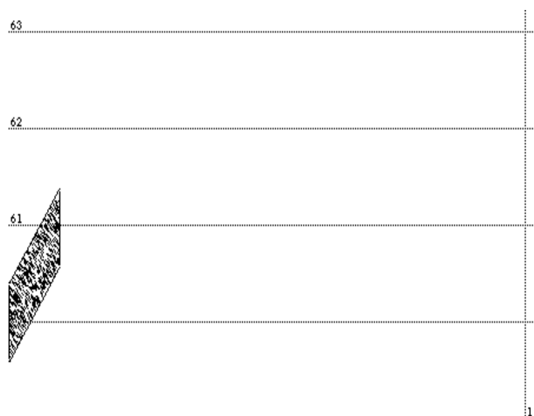


Figure 6.2: A very short note as a glissando

The leftmost column indicates the time on which the midi message is interpreted. The rightmost column contains numerical representations of the midi message itself. We will not go into detail about all the messages and parameters but just give a very general description of it. The first

three midi messages starting with 176 on line 2 to 4, take care of the pitch bend range being set at 48. This means that a note can maximally be deviated 48 semitones from the original note's pitch. On line 5 the program number is being set to 80. On our synthesizer this is a square lead sound. On line 6 the pitch bend is being set in the middle, so that initially the note's original pitch will sound. This is done because previous notes might have left the pitch bend at some undesired value that make our current note sound off tune. On line 7 we put the volume at our default value. A previous note might have left the volume at an undesired value, e.g., total silence, so we have to make sure the value is set to default again. On line 8 the "note on" message is sent and by default it is note number 60, the middle C with a velocity of 127. The velocity is an indication for the force with which a note is being played, comparable with the force of pressing a piano key. Then for 10 milliseconds the note sounds at its original pitch until at $time = 0.010$ a pitch bend message is sent. The pitch bend message, indicated by number 224 has two parameters. The first one represent the least significant bytes (LSB) and the second one represents the most significant bytes (MSB). Every other 10 millisecond the pitch bend value ascends, until at $time = 1.000$ the note reaches the pitch that corresponds with midi note number 61. Lastly a "note off" message is sent at $time = 1.000$ that turns the note off, as you might have guessed.

The 10 millisecond period is an arbitrarily chosen period that is used to generate a finite amount of midi messages corresponding to a conceptual continuous representation of a function. Testing with this period length proved that the sounds sounded convincingly as if they were really continuous, instead of discrete steps in pitch, volume, etc.

Translation of other aspect related GTFs

The method used for other aspects of a note, like volume, are somewhat less complex, as their control structure permits a more direct mapping with functions. Let us demonstrate an example in which the volume of a note goes from maximum (full loudness) to minimum, (total silence) and also inspect the corresponding midi sequence.

The NoteFun representation of this note is: `(note :duration .1 :amplitude (ramp 1 0))`. The corresponding midi sequence is given here:

```
0.000 begin
 0.000 midi  (176 100 0)                                   2
 0.000 midi  (176 101 0)
 0.000 midi  (176 6 48)                                    4
 0.000 midi  (192 80)
 0.000 midi  (224 0 64)                                    6
```

```
0.000 midi  (176 7 127)
0.000 midi  (144 60 127)                                    8
0.010 midi  (176 7 114)
0.020 midi  (176 7 102)                                     10
0.030 midi  (176 7 89)
0.040 midi  (176 7 76)                                      12
0.050 midi  (176 7 64)
0.060 midi  (176 7 51)                                      14
0.070 midi  (176 7 38)
0.080 midi  (176 7 25)                                      16
0.090 midi  (176 7 13)
0.100 midi  (176 7 0)                                       18
0.100 midi  (128 60 127)
0.100 end                                                   20
```

A graphical representation of this note can be seen in Figure 6.3.

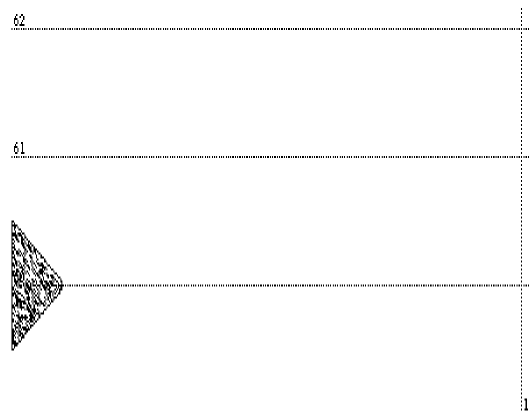


Figure 6.3: A very short note with a fade out.

The first 8 lines of the midi sequence we have already explained in the previous example. We have not yet explained that midi messages starting with 176 are special control messages. The first argument of this message is a control number and the second argument is a control value with can take on values between 0 and 127. The control number for volume is 7. If we would send the message `(176 7 0)` we would set the volume to total silence and if we would send the message `(176 7 127)` we would set the volume to the maximum. In our example we let the volume change linearly from the maximum to the minimum. In NoteFun we have chosen to indicate the volume with values between 0 and 1. What our implementation does is translate these values to volume control messages with corresponding values between 0 and 127. As you can see the control value gradually changes 127 to 0 throughout the duration of the note, just like we desired.

Having explained these two examples we have pretty much covered the key method of our translation from NoteFun structures to midi sequences.

Chapter 7

Final words

As I already mentioned in the Preface, my internship was a valuable time in which I learned a lot. It was a very good opportunity to combine various of my passions: computer science, higher order programming and music. During my internship I have broadened my insight in the application and crossing field of informatics and music, not only by working on this assignment, but also by talking with people, watching documentaries, reading books and by breathing the atmosphere at the MMM group. I think that everything there is to say about my experiences during my internship already has been said in previous sections, especially in chapters 5 and 6. I hope for you reading this report was a pleasure.

Bibliography

Matlab, the language of technical computing. The Mathworks, 1996.

The complete MIDI 1.0 Detailed Specification - Incorporating all Recommended Practices - document version 96.1. The Midi Manufacturers Assosication, 1997.

Peter Desain and Henk-Jan Honing. Time functions function best as functions of multiple times. *Computer Music Journal*, 16(2):17–34, 1992.

Charles Dodge and Thomas A. Jerse. *Computer Music; synthesis, composition and performance*. Schirmer, 1997. ISBN 0028646827.

Henk-Jan Honing. The vibrato problem: comparing two solutions. *Computer Music Journal*, 19(3):32–49, 1995.

Guy L. Steele Jr. *Common Lisp : The language*. Digital Press, 1990. ISBN 1555580416.

Peter Norvig. *Paradigms of Artificial Intelligence Programming.* Morgan Kaufmann, 1992. ISBN 1558601910.

Curtis Roads. *The Computer Music Tutorial.* MIT Press, 1996. ISBN 0262680823.

David A. Watt. *Programming Language Concepts and Paradigms.* Prentice Hall, 1990. ISBN 0137288662.