Gsharp
The Extensible, Interactive Score Editor

Robert Strandh

September 2003

Contents

Chapter 1

Introduction

1.1 Purpose and goals of Gsharp

Gsharp is an interactive editor for musical scores. It is interactive in the sense that the entire score is reformatted after each keystroke on the part of the user. There is no special button to click or command to type in order for the final layout to appear; the final layout is always the one that is presented.

One of the main goal of Gsharp is to provide a program that can produce high-quality scores. For that reason, we have put in some effort to respect at least one set of rules of musical typography (or *engraving*). But music engraving is hard and littered with very complex rules that do not necessarily cover all cases. For that reason, Gsharp also provides [1] a number of ways in which the user can help the layout engine make the right decision.

We wanted Gsharp to be extensible at a number of level. The ordinary end-user should be able to add and redefine keystrokes, to create macros, and to alter global parameters that affect the end result. The more advanced end-user should be able to write more complex extensions. Usually, programs that permit this use a Gsharp scripting language, which is usually different from the programming language that was used to implement the basic functionality of the program. Gsharp has a different approach. The entire program is written in Common Lisp which both is as fast as a lower-

[1] This feature is not yet implemented. We ultimately intend to provide a large spectrum of ways for the user to tweak the layout decisions, but for now, only automatic layout exists.

level language and provides the ability to dynamically replace any part of the program.

Since Gsharp makes extensive use of graphics and related functionality, we needed a good library for providing such functionality. While the Common Lisp standard does not talk about such libraries, there as been a semi-standard library available for many years and which only recently exists in a freely distributable version, namely CLIM, which stands for the Common Lisp Interface Manager. Many of the basic functions of Gsharp are due to CLIM, and the advances Gsharp user can take advantage of this fact in order to improve the program.

We deliberately did not try commercial score editors so as not to be influenced by their view of the editing task. This fact has both positive and negative consequences. On the positive side, Gsharp (we hope) provides novel ideas concerning user interaction. We are also hoping to avoid being sued by having a program that looks like a copy of some commercial one. On the negative side, users familiar with existing commercial editors will not be able to take advantage of this knowledge when using Gsharp. Also, some commercial editors probably have some very good ideas that would have been interesting to take advantage of in Gsharp, which because of this policy will not be possible.

Gsharp is intended to be “user friendly”, but since people do not agree on a definition of the term, this is what we mean here (largely inspired by Cooper). We do not try to cater to new or casual users. Instead we try to make the average user with at least a few days of experience as efficient as possible. This might mean that the user has to occasionally read the manual in order to figure out how to operate Gsharp. We think that is a reasonable price to pay in order to be efficient.

1.2 Gsharp is free software

By “free software”, we mean that Gsharp can be used, redistributed, and improved by anyone. Technically, the main parts of Gsharp are distributed according to the GNU General Public License (GPL). Without going into detail here, this means that improvements to the software must (if distributed at all) also be distributed according to the GPL. Ordinary users are not affected by the license, which covers only improvements and redistribution.

Some parts of Gsharp are distributed according to the GNU Lesser General Public License (LGPL). Again, without going into detail, this license is used for more general-purpose code such as extensions to CLIM, and allows a user to distribute it as part of commercial software. As with the GPL, contributions to this code must be redistributed according to the same license.

1.3 Conventions

This manual mostly reflects the current state of things. Occasionally, we documented a feature that is not yet implemented, so that you would have an idea what it would be like. In that case, we added a footnote to indicate that the feature is not yet implemented.

1.4 Releases and release numbering

Releases are numbered A.B, where A is the major release number and B is the minor release number or version number. The first release is 0.1. We will increase the minor release number with at least one for each new version of the program made available to the public. The major release number is incremented only when a major overhaul has been done, or when large chunks of code have been added.

Major release 0 is not intended for prime time, but only for end users who want to get an idea of the direction we are heading. Release 1.0 will be declared when the program is usable and stable enough for general use.

There is no upper bound on the minor release number.

1.5 Contributions

We accept contributions in the form of bug fixes, improvements to the existing code, and new code. Since Gsharp is distributed according to the GPL, nothing prevents any user to add new code and distribute the result. We hope that this will not happen immediately, since we would like to guarantee some consistency of the program, both regarding its external appearance and its internal organization. For that reason, it would be preferable to discuss with us before contributing some code.

Contributors must have a reasonable level of knowledge of Common Lisp. As with any language, it is possible to write poor code with Common Lisp. It is our goal to try to keep the code at a reasonably good level. For that reason, contributors are encouraged to put some effort into the quality of the code, to make sure it compiles without warnings (as much as possible) and to make sure it works.

We also accept contributions to this document. We intend to use some kind of free license for this document, probably the GNU Free Documentation License, but we have not had time to figure out how it works yet.

1.6 Organization of this document

This document is organized in several *parts*.

The *getting started* part is intended for users who look at Gsharp for the first time, and who want to know how to use the most fundamental aspects of it, in particular the basic architecture of the program. It is not intended to be a complete user's guide, but instead structured around a few typical use cases encountered by users.

The *reference manual* is the part that is most useful to the user that has graduated beyond the first impression, and who want to know more about the features of Gsharp. The structure of this part is not meant to be pedagogical, but instead a complete enumeration of the features of Gsharp that a moderately experienced users needs to know.

The *internals* part is intended for advanced users who need to know how to customize Gsharp in non-trivial ways, as well as for the programmer who would like to extend or modify Gsharp.

Finally, the *appendices* part provides information on topics that are neither related to the user of Gsharp, nor to the programmer who wants to extend it, but to readers who simply want to know more about music engraving or about how Gsharp was created, and why it was created that way.

Part I

Getting started

Chapter 2

Installing and running Gsharp

2.1 Installing Gsharp

We have not worked on the installation procedure at all, and in fact, we have not decided exactly in what form the releases will be made available to the public.

For now, you have to use CMUCL (perhaps SBCL will work as well, but it has not been tested) with a core image that contains CLIM and MK:DEFSYSTEM. Start Lisp, load the file `system.lisp` into the running Lisp image and then type:

```
(operate-on-system :gsharp :compile)
```

which will compile and load all the files. To start Gsharp type

```
(gsharp::run-gsharp)
```

2.2 The different editing panes

You should be seeing a main window (known as a *frame*) with a menu bar and four sub-windows, known as *panes*.

The largest pane is the *score pane* where you will do most of the editing, The pane at the bottom is a CLIM *command pane*. In it, you can type commands that do not have keyboard shortcuts.

In the upper-right corner, you will see a the pane that determines the *input state* The input state shows a stylized version of a notehead, possibly with a stem, augmentation dots, and beaming information. If the stem is down, all the new clusters entered will have their stems down. If the stem is up, all the new clusters entered will have their stems up. If the input state shows both a stem up and a stem down, all new clusters will have a stem direction of *auto*, indicating that Gsharp will determine stem direction based on position of the notes on the staff and beaming information.

In the lower-right corner, you will see a pane that gives more detail about *current element*, i.e., the element (if any) immediately to the left of the cursor), namely the *current element pane*. (We may eliminate this pane in the future if we can figure out a way to display this information in a not to disturbing way directly in the score pane).

2.3 The cursor

In the score pane, you will see a yellow vertical line across the single staff on display and two parallel vertical red lines that are usually near the yellow line. This is the Gsharp *cursor* where all editing takes place.

The cursor is either before all the elements, after all the elements, or between two element. Newly created elements will be inserted immediately to the left of the cursor (which will make the newly inserted element the *current* one.

The yellow line indicates the staff to which newly inserted elements will belong. The red lines indicate the interval of pitches that will be used for newly inserted notes. This interval is centered around the last note that was inserted. You can change this interval by moving it up or down one octave using the commands ‘L’ (Lower) and ‘H’ (Higher).

Chapter 3

Basic editing and cursor motion

3.1 Entering single-note elements

To enter single-note elements, you use the keyboard and type names of notes such as ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘a’, or ‘b’ (in lower case). Each time you type the name of a note, a new single-note element will be inserted to the left of the cursor.

Gsharp will try to guess which octave you want from the previous note you typed, so that the pitch distance between the two is minimized. If this choice is not right, you can type a capital ‘L’ (for “Lower”) in order to tell Gsharp to decrease its idea of what notes to enter next by one octave, or ‘H’ (for “Higher”) in order to tell Gsharp to increase its idea of what notes to enter next by one octave.

The visible cursor will indicate what interval Gsharp thinks is the best for the next inserted note with two parallel vertical red lines.

3.2 Adding notes to existing elements

In order to add a new note to the current element, you type the note name with a capital letter such as ‘C’, ‘D’, ‘E’, ‘F’, ‘G’, ‘A’, or ‘B’.

Again, Gsharp will try to guess which octave you want from the previous note you typed, so that the pitch distance between the two is minimized.

Currently, there is no way to remove a single note from an element (though we may implement that in the future). For now, if you add a note by mistake, the only way you can fix it is by erasing the entire element and start over.

3.3 Adding a rest

A rest is another kind of element. You enter a rest by typing ‘,’ “Insert Rest”. Conceptually, every rest has a notehead as well as rbeams and lbeams in addition to augmentation dots. These are used to determine the duration and what kind of rest to display.

3.4 Deleting elements

To erase the current element, type ‘Control-h’ (“Erase Element”), i.e. hold the ‘control’ key down while hitting the ‘h’ key.

It is also possible to delete the next element after the cursor by typing ‘Control-d’ (“Delete Element”.

In general, we speak about *erasing* when the material removed is to the left of the cursor, and *deleting* when it is to the right.

3.5 Moving by element

To move backward and forward over the elements that you have typed, you can type ‘Control-b’ (for “Backward Element”) or ‘Control-f’ (“Forward Element”).

3.6 Adding and removing augmentation dots

If you would like to add an augmentation dot to the current element, or more generally, to add another dot to the ones already there, you type ‘.’ (“More

Dots”). To erase one of the dots, type ‘x.’ (“Fewer Dots”).

The key ‘x’ can be used in many other situations as a prefix to indicate the opposite command from the one indicated without the ‘x’ prefix.

3.7 Adding and removing beaming and flags

Gsharp maintains beaming information for each element in the following way. Each element has a number of right beams (or *rbeams* for short) and a number of left beams (or *lbeams* for short).

If the element has neither lbeams nor rbeams, the element is considered as having neither flags nor beams. Its duration as far as layout is concerned will then be determined by the notehead and the augmentation dots. The duration of the element will be determined by the maximum number of lbeams and rbeams. As usual, the value of the notehead is divided by two for each beam (and then increased by any augmentation dots).

If the element has rbeams or lbeams, but neither the element to its left has any rbeams nor does the element to its right have any lbeams, the element will be displayed with a number of flags corresponding to the maximum of its lbeams and rbeams.

If the element has lbeams and the element to the left has rbeams, then a beam will be displayed between the two elements. The number of beams that will be displayed is the minimum of the rbeams of the element to the left and the lbeams of the element itself. Remaining beams are displayed as fractional beams [1]. The symmetric rule holds for the element to the right.

To increase the number of rbeams of the current element, type ‘]’ (“More Rbeams”). To decrease the number or rbeams of the current element, type ‘x]’ (“Fewer Rbeams”). The corresponding commands for the lbeams are ‘[’ (“More Lbeams”) and ‘x[’ (“Fewer Lbeams”).

Thus, to obtain a two-element beam group, the first with a C and the second with a D, you would type the sequence ‘c]d[’ (think of it as “c, beam right, d, beam left”).

[1] This feature is not yet implemented. We only show a single beam at the moment

3.8 Changing the notehead

In order to change the notehead of the current element, you type 'Meta-h' ("Rotate Notehead"). If the notehead of the current element is *whole*, it will change to *half*, if it is *half*, it will change to *filled*, and if it is *filled*, it will change to *whole*.

When the notehead is *whole*, no stem, flags or beams will be displayed, nor taken into account to compute the duration of the element. The information will not be removed if present, however, so that if you change the notehead from *filled* to *whole* and then back to *filled*, you will be back to the original element.

When the notehead is *half*, a stem will be displayed, but no beams or flags. Again, the information is not erased, in case you would like to change back to a *filled* notehead.

Chapter 4

Changing individual notes in an element

Gsharp maintains a concept of a *current note* of the current cluster. A note that has just been inserted or added to an existing element becomes the current note. Several commands act upon the current note.

To add a pitch alteration to the current note, you can type '#' ("sharper") to make the key sharper (if you repeat this command again, it will become double sharp) or '@' ("flatter"). Whether or not this operation results in an accidental being displayed, depends on the key signature of the staff. You can change the key signature of the staff by using the command 'Meta-#' (More Sharps) to add another sharp sign to (or to remove a flat sign from) the signature or the command 'Meta-@' (More Flats) to add another flat sign to (or to remove a sharp sign from) the signature.

To change the pitch of the current note, you can type 'Meta-u' (Up) to increase it or 'Meta-d' (Down) to decrease it. Notice that Gsharp does not allow for two different notes having the same pitch in a cluster. An attempt to move the pitch of the current note so that it ends up on top of an existing one, will remove one of the notes.

The commands 'Meta-u' and 'Meta-d' can also be used on a rest, in which case they change the vertical staff position of the rest. This is useful if you have several voices on one staff and you would like to have one voice higher on the staff than the other. In the future, we will implement the possibility for a voice to have a default staff line for rests.

In a multi-notehead cluster, you will have to change the current note. You can do this by typing 'p' to make the note with the next higher pitch the current one, or 'n' to make the note with the next lower pitch the current one.

Occasionally, you might want to display an accidental even though the layout engine decided it was not necessary because the staff position of the note has the same pitch alteration inherited from the key signature. [1].

[1]This feature is not yet implemented. This is a matter of adding another slot to a note

Chapter 5

Changing the input state

If you have a long sequence of elements, each with (say) three flags to type, you would not want to enter these flags for each element, since you would have to type something like ‘c]]]d]]]e]]]f]]]g]]]’. A better idea would be to change the input state (or istate for short) so that each new element has three rbeams and then just to type ‘cdefg’.

Changes to the input state are made in the score pane, but they have an immediate effect on the presentation in the input state pane.

Commands to change input state mimic those of changes to the current element, except prefixed by ‘i’. Thus, to add in rbeam to the input state, you would type ‘i]’, and to erase one rbeam you would type ‘ix]’. Adding and removing lbeams and dots work in the analogous way.

Similarly, in order to change the notehead of the input state, you type ‘in’ (“Istate Rotate Notehead”). If the notehead of the input state is *whole*, it will change to *half*, if it is *half*, it will change to *filled*, and if it is *filled*, it will change to *whole*.

Chapter 6

Operations on measures

You add a measure bar at the cursor location by typing '|'. As with other element, the cursor will be positioned right after the newly created measure bar. In fact, a measure bar is just like any other element as far as editing operations are concerned.

Thus, to delete a measure bar, you simply use the ordinary commands for erasing or deleting elements.

Chapter 7

Operations on layers

To change the current layer, you can type 'Meta-p' ("Previous Layer") or 'Meta-n' ("Next Layer".

To insert a new layer, you have to go to the command pane and type the full command, either "Insert Layer After" or "Insert Layer Before" according to whether you would like the new layer to be ordered before or after the current one. The order between layers only affects the order in which they are presented to you with the "Next Layer" and "Previous Layer" commands. You will be prompted for the name of a staff to use for displaying material in this layer. Later, you can modify the relation between layers and staves.

To delete the current layer, you have to use the command pane (since this is an operation that is presumably rare). The command to use is "Delete Layer".

A layer can be displayed on one or several *staves*. Most layers will have a single staff associated with them, but is occasionally useful to have two or more. Clusters within such a layer with notes in different staves will have their stems span those staves. At other times, it is useful simply to have some elements of the layer displayed on one staff and some other elements on a different staff.

A layer can have a preferred stem direction for all clusters in it[1]. The user can override this stem direction for individual elements, but if the stem direction of the cluster is *auto*, then it will be displayed with the stem direction

[1] This feature is not yet implemented. It is not hard to do, though

determined by the layer. If the stem direction of the layer is also *auto*, the layout engine will determine the stem direction.

Chapter 8

Operations on staves

To add a new staff, use the command "Add Staff". You will be prompted for the name of the new staff.

To delete a staff, use the command "Delete Staff". Again, you will be prompted for the name of the staff to delete.

To add a staff to the current layer (so that material in that layer can be displayed on that staff), use the command "Add Layer Staff". You will be prompted for the name of a staff to add.

To delete a staff from the current layer, use the command "Delete Layer Staff". You will be prompted for the name of a staff.

To alter the key signature of a staff, use the commands 'Meta-#' "More Sharps" and 'Meta-@' "More Flats".

Part II

Reference manual

Chapter 9

The user model of Gsharp

In order to be an effective user of Gsharp, you need to understand its *user model* , i.e., the way the user is supposed to think about the structure of the score being created.

9.1 Buffer

The basic abstraction of Gsharp is that of a *buffer* . The buffer is the in-memory representation of an entire musical score. Gsharp can have several buffers simultaneously in memory.[1].

Loading a Gsharp file into the editor creates a new buffer, and a buffer can be saved either to the file from which it was initially loaded, or to some other file. A new buffer can be created (which would then not yet be associated with a file).

The page layout algorithm of Gsharp has been designed to be able to cope with very large buffers, while still updating the display in real time. Several hundreds of pages should not be a problem. Though, if you change some global parameter which effects the complete layout of the entire score, such as the spacing style, and you have a very large buffer, you might have to wait a while before Gsharp has completed the calculation of the new page layout. Ordinary local changes that do not impact page layout elsewhere

[1] This feature is not yet implemented. It is not hard to implement, but it just has not been done yet –RS 2003-08-16

only require a very small amount of work for Gsharp, so updating the screen should be immediate.

9.2 Segment

A buffer is divided into smaller pieces known as *segments* . A segment roughly corresponds to a musical phrase. Segments do not overlap, or only slightly. Gsharp contains a number of useful commands that operate on an entire segment. You can copy a segment or listen to it played as a Midifile.

9.3 Layer

A segment can have several *layers*. The layers of a segment are temporally superimposed. You might think of a layer as corresponding roughly to a *voice* or a *part* of the music.

The difference between a layer and a part (gs does not know about parts) is that a layer has a particular *instrument* assigned to it, whereas a part can have several different instruments, for instance when the same musician plays several instruments (though not simultaneously). This different poses no problem, because it is possible to organize Gsharp layers so that their musical material does not overlap in time, even though layers conceptually do.

9.4 Slice

Each layer has exactly three parts, referred to as *slices* , the *head slice* , the *body slice* , and the *tail slice* . [2]

[2] This feature is not yet implemented. We only use the body slice at the moment, which of course is the most important one. However, the head slice is important for [and I do not know what this is called in English] the few notes that might logically belong to a phrase, but that temporally precedes it. I have no idea, though, what the tail slice might be useful for. –RS 2003-08-16

9.5 Element

Each slice is a sequence of *elements*. There are different kinds of element, *clusters*, *silences*, and *measure bars*. For the remainder of this section, we ignore the possibility of an element being a measure bar, since a measure bar does not have any interesting properties such as duration.

In the body slice, the first element starts at time zero from the beginning of the slice, and each subsequent element starts when the preceding one ends. The same thing is true for the tail slice. [3] The head slice is different, in that it is aligned from the *end of the preceding slice.* [4].

Each element (except the measure bar) has a particular *duration* . The duration is indicated by a *notehead* and a *number of dots.* Notice that the notehead property exists even if the element is a silence. It is used to indicate what kind of character to use for the silence.

9.6 Cluster

A *cluster* is roughly equivalent to a chord, in that it contains all the notes that should be played simultaneously within a layer. In addition to the properties inherited from the element, each cluster has a *stem direction* .

Clusters have zero or more *notes.* When there are zero notes in the cluster, it is not displayed at all, but the layout algorithm takes it into account with its ordinary duration. This allows the user to introduce spaces in the music that are not filled with explicit silence.

The stem direction indicated in the cluster may or may not be respected by the layout algorithm. First, if the stem direction is *auto*, the layout algorithm will use its knowledge about music engraving to display the cluster either with the stem up or with the stem down. When the stem direction is *down* or *up* as chosen by the user, the layout algorithm always respects this choice when the cluster is not part of a beam group. Currently, all the stems of a beam group are displayed in the same direction (we will change this in the future, at least for beam groups where the elements are on different staves). Thus, when the cluster is part of a beam group the stem directions used

[3] This feature is not yet implemented. We do not yet have a tail slice
[4] This feature is not yet implemented. We do not yet have a head slice

are determined by the first element of the group. When the first element of the group has stem direction *auto*, then Gsharp will compute the best stem direction for all the elements of the group. When the stem direction of the first element is either *down* or *up*, Gsharp will honor that choice. Obviously, in this case, stem directions of the other elements are ignored.

9.7 Silence

A silence is an other example of an element. Although always displayed with the traditional characters, a silence still has a notehead, which is used in this case to compute the duration of the silence, and thus what character to use to display it and to determine spacing.

9.8 Measure bar

A measure bar is another kind of element. It has no interesting properties in itself. Apart from being displayed, it is used to align musical material in different layers. Thus, Gsharp does not bother to verify that each layer has the same duration between two measure bars. It simply lines up every measure bar in every layer. If some layer has a measure that has a shorter duration than the corresponding measure in a different layer, the shorter one will be padded by empty space so that they line up.

Also, Gsharp does not verify that there is the same number of measure bars in each layer. When a layer runs out of measure bars, the remaining ones are lined up without taking into account the layer that ran out.

Chapter 10

Summary of commands

10.1 Altering the input state

Key	Command name	Description
‘L’	Lower	Decrement the interval of pitch of the input state by one octave
‘H’	Higher	Increment the interval of pitch of the input state by one octave
‘i.’	More Dots	Add another dot
‘ix.’	Fewer Dots	Remove a dot
‘i]’	More Rbeams	Add another beam to the right
‘ix]’	Fewer Rbeams	Remove a beam to the right
‘i[’	More Lbeams	Add another beam to the left
‘ix[’	Fewer Lbeams	Remove a beam to the left
‘ih’	Rotate Notehead	Change the notehead
‘is’	Rotate Stem Direction	Change the stem direction

10.2 Moving the cursor

Key	Command name	Description
‘Control-f’	Forward element	Move the cursor forward one element
‘Control-b’	Backward element	Move the cursor backward one element

10.3 Inserting and adding notes, rests and elements

Key	Command name	Description
‘c’	Insert Note C	Insert a new cluster having a single note C in it
‘d’	Insert Note D	Insert a new cluster having a single note D in it
‘e’	Insert Note E	Insert a new cluster having a single note E in it
‘f’	Insert Note F	Insert a new cluster having a single note F in it
‘g’	Insert Note G	Insert a new cluster having a single note G in it
‘a’	Insert Note A	Insert a new cluster having a single note A in it
‘b’	Insert Note B	Insert a new cluster having a single note B in it
‘,’	Insert Rest	Insert a new rest element
‘|’	Insert Measure Bar	Insert a measure bar
‘SPC’	Insert Empty Cluster	Insert a new cluster with no notes
‘C’	Add Note C	Add the note C to the current cluster
‘D’	Add Note D	Add the note D to the current cluster
‘E’	Add Note E	Add the note E to the current cluster
‘F’	Add Note F	Add the note F to the current cluster
‘G’	Add Note G	Add the note G to the current cluster
‘A’	Add Note A	Add the note A to the current cluster
‘B’	Add Note B	Add the note B to the current cluster

10.4 Deleting and erasing elements

Key	Command name	Description
‘Control-d’	Delete Element	Remove the element to the right of the cursor
‘Control-h’	Erase Element	Remove the element to the left of the cursor

10.5 Operations on the current element

Key	Command name	Description
'Meta-h'	Rotate Notehead	Modify the notehead (whole – half – filled –)
'Meta-s'	Rotate Stem Direction	Modify the stem direction (up – down – auto –)
'.'	More Dots	Add another dot
'x.'	Fewer Dots	Remove a dot
']'	More Rbeams	Add another beam to the right
'x]'	Fewer Rbeams	Remove a beam to the right
'['	More Lbeams	Add another beam to the left
'x['	Fewer Lbeams	Remove a beam to the left
'Meta-u'	Up	Move rest to a highter staff line
'Meta-d'	Down	Move rest to a lower staff line

10.6 Operations on the current note

Key	Command name	Description
'p'	Current Increment	Change the current note
'n'	Current Decrement	Change the current note
'#'	Sharper	Make the note sharper
'@'	Flatter	Make the note flatter
'Meta-u'	Up	Increase pitch of note
'Meta-d'	Down	Decrease pitch of note

10.7 Operations on the current layer

Key	Command name	Description
‘Meta-n’	Next Layer	Make the next layer current
‘Meta-p’	Previous Layer	Make the previous layer current
	Insert Layer After	Insert new layer after current
	Insert Layer Before	Insert new layer before current
	Delete Layer	Delete the current layer
	Add Layer Staff	Adds a new staff to the layer (promts for a name of an existing staff)
	Delete Layer Staff	Deletes a staff from the layer (promts for a name of an existing staff)

10.8 Operations on staves

Key	Command name	Description
	Add Staff	Add a new staff (promts for a name)
	Delete Staff	Delete a staff (promts for a name)
	Rename Staff	Rename current staff (prompts for a new name)
	Set clef	Set the clef of the current staff (promts for name of clef and line number)
‘Meta-#’	More Sharps	removes a flat or adds a sharp to the key signature
‘Meta-@’	More Flats	removes a sharp or adds a flat to the key signature

10.9 Playing as MIDI

Key	Command name	Description
	Play Segment	Play the current segment
	Play Layer	Play the current layer

10.10 File operations and quitting

Key	Command name	Description
	Quit	Quit Gsharp
	Load File	Load a Gsharp file into a new buffer (prompts for the name of a file
	Save Buffer As	Save the buffer into a file (prompts for the name of a file
	New Buffer	Create a new buffer (currently wipes the current buffer)

Part III

Internals

Chapter 11

Implementation of the user model

The implementation of the user model pretty much reflects the model itself, except that the implementation of the *slice* is not a simple sequence of elements.

Instead, a *slice* is a sequences of a user-invisible type called a *bar*. We introduced this additional type to simplify the layout algorithm. A bar contains all the non-measure-bar elements between two measure bars. The layout algorithm therefore does not have to start by finding the measure bars in the sequence, but can directly line up the material between measure bars.

The implementation of the model is actually spread out in three different files:

- `buffer.lisp`: This file is the one that most immediately corresponds to the user model. Generic functions allow insertion and deletion of the different types used to implement the model. Currently, the implementation of various sequences is in the form of simple Lisp lists. We might change that one day for performance reasons, but then again, it may not be necessary.

- `numbering.lisp`: This file observes the action (through :before, :after, and :around methods) of the basic operations in order to number different instances of the basic types. This allows us to know at any point in time whether an instance precedes another in a sequence. The implementation of the numbering scheme is not very efficient at the moment, but it might not matter since sequences are fairly short in general.

- `cursor.lisp`: This file implements cursor operations. It observes operations in the preceding two files in order to move around cursors.

A cursor has two slots, a bar in which it is located, and the position within the bar. Cursors are inserted on most of the model types (bar, slice, segment, buffer) so that when one of those elements is deleted, the cursor can be moved accordingly. Things get complicated because we want to allow for an arbitrary number of cursors, each attached to a different window, though we do not currently take advantage of that possibility.

Chapter 12

The Layout algorithm

12.1 Purpose

The purpose of the layout algorithm is to divide the entire score into displayed units. Currently, these units are *lines*, where a line is the part of system displayed in sequence from left to right. In the future, it will be possible to use *pages* as units which will make things a bit more realistic. Since we currently do not have pages, our display is mostly in the form of a "roll of paper".

12.2 What it needs to recompute

The layout algorithm is invoked after each keystroke. But in order to do its work correctly, it has to know what has been damaged between two invocations. The user can have executed some arbitrary code between two such invocations, and in order not to have to recompute the entire score each time, it preserves the result of its calculation between invocations and only recompute what is necessary.

To know what has been damaged, the layout algorithm (in the file `measure.lisp`) installs a different set of `:before`, `:after`, and `:around` methods to observe the operations on the fundamental implementation of the user model. Currently, it does its work in a very crude way. Whenever the structure has been modified in any way, the entire segment is marked as modified and will be totally recomputed at the next invocation.

Recomputing the layout involves computing, for each segment, a sequence of *measures*. This is a user-invisible type that corresponds to the superimposition of simultaneous bars in overlapping layers. We could be more careful about marking things as damaged, and preserve measures that contain bars that have not been modified. We might do that if performance requires it. At the moment, we have no reason to believe that it will be necessary.

12.3 The importance of segments

The importance of the concept of a segment should now be clear. If the user deletes a measure bar, all measures to the right of the point of deletion *within the same segment* must be recomputed, since the musical material will no longer line up the same way. Had the score been a single segment, we would have had to potentially recompute the entire score in such situations. In addition, it would be pretty annoying to the user not to be able to count on finished material far from the cursor to remain intact. Thus, the concept of a segment both helps the user organize his material and helps the implementor limit the damage that needs to be repaired by the layout algorithm.

12.4 Head and tail slices

There is a slight problem with *head slices* and *tail slices* which is why they are not implemented yet. While conceptually these slices below to the segment of their corresponding layer, the layout algorithm must align them with the last measure of the previous segment and the first measure of the next segment. To make it impossible for those slices to stick out arbitrarily far, we do not allow for head and tail slices to contain measure bars. This limits the stick-out to the previous and the next segments. We have not yet decided how to manage these slices. Probably, a modification would have to mark the previous or the next segment as damaged, or at least the last measure or the first measure in these segments.

12.5 The concept of a measure

A measure, as far as the layout algorithm is concerned, only contains some very limited amount of information about its bars so that it will know how much room the measure will take up on a given line of musical material.

The main problem we need to address is the fact that a measure takes up a different amount of space according to the contents of the other measures of the same line. The remainder of this section will discuss how we solve that problem.

First, let us introduce the concept of a *time line*. A time line represents a moment in time in a given measure such that the measure has some event (like notes) that start at that moment in time. The *duration* of the time line is the temporal distance between it and the following one in the measure, or, if it is the last one, the duration of the longest lasting event that starts at the moment in time.

From the art of music engraving, we know that a longer temporal distance requires a longer geographic distance on the score as well, but that is not all there is to it. In fact, we also know that *on a given line*, the smallest geographic distance possible (a global parameter of the score that determines denseness and that we will call w_min) is assigned to the shortest temporal distance on that line, *independently of the absolute value of that temporal distance*. The other geographic distances are adjusted accordingly. Thus, a measure can take up a lot more space on a line if other measures on the same line have shorter minimal temporal distances between time lines than itself.

From the articles published about the Lime score editor, we use the following formula relating the ratio between the geographic distance of two time lines to the ratio of their durations:

$w_2/w_1 = (d_2/d_1)^k$

where w_1 and w_2 are the "widths", i.e. geographic distances of the two time lines, and d_1 and d_2 are their durations. The parameter k takes on values between 0 and 1, where 0 gives constant spacing (space is independent of duration) and 1 gives proportional spacing. A good value for k seems to be around 0.6.

On a given line, with a smallest temporal distance d_{min} between any adjacent time lines, we can express the geographic distance w following any time line with duration d as:

$w = w_{min}(d/d_{min})^k$

Let us now define the *natural width* of a measure as:

$W_{nm} = \sum_i w_{min}(d_i/d_{min})^k$

where d_i is the duration of the ith time line of the measure, and the *natural width* of a line W_{nl} as the sum of the natural widths of the measures of the line.

We can rewrite the formula above as:

$W_{nm} = (1/d_{min})^k w_{min} \sum_i {d_i}^k$

Where $w_{min} \sum_i {d_i}^k$ is a property of a measure that remains constant as long as the two global parameters w_{min} and k do not change. We shall call this property the *measure coefficient*. To get the natural width of the measure, it suffices to multiply its coefficient with $(1/d_{min})^k$, which varies between different lines. Thus, we can compute the coefficient for a measure once and for all (again as long as the global parameters stay the same).

To compute the natural width of a line, we have to compute the sum of the coefficients of the measures on the line and the minimum of their respective minimal time line durations. This is a constant time operation that does not require looping over individual measures of the line.

Now that we know how to compute the natural width of a line, we need to divide the score into lines in such a way that we have the best possible layout. For that, we need to define the *cost* of a line.so that we can compare two different suggested lines. In Gsharp we currently use the amount we have to stretch or compress it compared to its natural width in order for it to fit on a line. Let us define the *compress factor* of the line as the quotient between the natural width of the line and the width of the page. We then define the *cost* of a line as the maximum between the compress factor and the its inverse. That is not the only way to do it. We might imagine penalizing compressions more than stretches, as it might be much harder to fit the musical material on the line when it needs to be compressed.

Our definition of the natural width of a measure does not take into account extra material such as accidental that need more room. We could do a more precise job by adding in some extra room for such material and count it toward the natural width of the line only if the measure does not need to be stretched (i.e. only if the minimum duration of the line is the same as that of the measure). This would only slightly complicate the method of combining measure parameters into line parameters and it might give a considerably better result.

However, it might not be very important to improve this calculation. The result of the calculation is only used to divide the score into lines and pages. Once we display the few visible pages of the score, we have great freedom to reorganize the material on the page. Dense measures might have to borrow space from less dense ones on the same line. As I recall, the article on the Lime music editor describes how to do this in great detail, although it is not yet implemented in Gsharp. Since usually even under very crowded conditions, space can be reorganized within the line, the only risk we run of not having a very precise calculation of the natural width of a line is that one line might look slightly denser than another one on the same page, because we did not take into account all of its accidentals.

12.6 Breaking measures into lines

Breaking the sequence of all measures in the score into subsequences corresponding to lines is done by an algorithm that is similar to *dynamic programming*. Essentially, we compute all possible line breaks and choose the best one, but thanks to dynamic programming we avoid a possibly exponential complexity.

But even the cubic complexity of dynamic programming would be unacceptable for our purposes. We transform that complexity from cubic to constant by a series of tricks, which are documented in a separate (unpublished) article in the same directory as this one.

Chapter 13

Gsharp as a CLIM application

Chapter 14

The Obseq library

Chapter 15

Fonts

In order for Gsharp to have a nice appearance on the screen, it was determined essential to use anti-aliased fonts. The problem is that the X Window System does not have such fonts in the standard. There is an extension to the XFree86 server for the X Window System which uses anti-aliased fonts, but it is not an official standard for the X Window System. In addition, this extension requires the *client* to convert glyphs to rasters, which traditionally has been the responsibility of the server of the X Window System. Gilbert Baumann has created a new server for the X Window System that is capable of anti-aliased fonts, but again, this is not available in all servers for the X Window System.

As mentioned above, in recent extensions to the X Window System, converting the glyph to a raster is a client-side operation. Usually, this operation is done by a library such as FreeType, which can use font of type TrueType or PostScript Type 1. Each of these systems has its own method for rendering low-resolution fonts.

I decided that if converting glyphs to rasters were to be a client-side operation, I might as well have full control over it, and use the METAFONT program for this purpose. METAFONT is a universal programming language that does not suffer from the limitations of the *hints* in systems such as PostScript Type 1. Using METAFONT, it would even be possible to create fonts specifically useful for anti-aliasing.

The font used by Gsharp (called SDL, for Score Drawing Library), is created by a METAFONT program that generates glyphs that are magnified a factor four. The resulting `gf` filed is parsed by a Common Lisp library that creates a CLIM pixmap which contains an anti-aliased version of the glyph.

The difference between this solution and the (more general) one based on a library such as FreeType, is that FreeType (together with the *render*-extension of the X Window System) provides *transparency* (also known as an *alpha*-channel), which greatly simplifies high-quality output, whereas using pixmaps makes it necessary to display musical output in a certain order, and to do a certain amount of cheating.

The great advantage of the Gsharp solution, though, is that it will work with any basic server for the X Window System, and that it is very fast.

The main problem with our solution (and this is why cheating is necessary) is that glyphs can partly overlap on the page. In particular, this occurs in stacks of noteheads that may or may not belong to the same cluster (they could come from different layers, only be placed a staff step apart), but also trivially between any music character and the staff lines. When that happens with ordinary fonts, no problem results, but when gray-scale fonts are used, they do not combine very well.

For staff lines, stems, and ledger lines, we solve the problem easily by imposing that these characters always be pixel aligned (and thus completely black). It then suffices to draw these characters *after* the ones they overlap with.

For stacks of noteheads, the problem is greater. We solve it by substituting different glyphs that correspond to combinations of noteheads in a stack. To avoid having an unbounded number of such combinations, we have one glyph for the lower half of a notehead, one for the upper half, and one for the combination of the upper half of one notehead and the lower half of the one on top of it. All these glyphs are properly anti-aliased and their bounding boxes do not overlap so combining them does not create any problems [1].

The way we are planning to substitute alternative glyphs is by using the *output recording* mechanism of CLIM. This mechanism is responsible for repainting windows that have had their contents partially lost due to being covered or invisible. It work by capturing output and storing output records in the pane. We plan to use it to introduce a new kind of output record that guarantees that repainting will be done in the right order, and that

[1] This feature is not yet implemented. We still need to use the CLIM output recording mechanism to remember positions of noteheads and to substitute these alternative glyphs

alternative glyphs will be substituted.

Chapter 16

Beam drawing

It might seem like drawing beams would be trivial. the X Window System provides adequate primitives to draw filled polygons that would be generalizations of beams in a score. As with fonts (see chapter 15) the problem is that anti-aliasing is called for in order for the visual appearance to be acceptable, and that the X Window System does not provide for anti-aliasing in the basic protocol.

A possibility would be to treat beams as any other character glyphs and to write a METAFONT program for creating beams. The problem with this solution is that there is a large number of different beam slopes possible, and that a font would have to include a very large number of them. Furthermore, beams having fairly small slopes, turn out to use only a small fraction of the 16 different gray-levels that we would like to use for anti-aliasing.

For that reason, we generate beams on an as-needed basis. A beam is divided into segments, each with a Δy of 1 (but each possibly with a different Δx. Each segment is drawn as a central black rectangle surrounded by two pixmaps, each of height 1 and the length of the segment. One of the pixmaps provides gray-levels from black to white and the other one from white to black. This method gives a smoother impression than if polygon drawing were used. Each new segment is similar to the previous one (its length might vary a little) and positioned one position higher or lower than the previous one.

Part IV

Appendices

Appendix A

Development history

The idea of Gsharp can be traced back to 1994, when Robert Strandh proposed a DESS project [1]. This project was conducted by Laurent Kerhoas (and ...), and resulted in the first real prototype of Gsharp. The main features of this prototype was that it was written in Tcl, and that its *user model* (see chapter 9) was quite different from the current one.

As a summer internship, what-is his-name continued working on the project. This work was important in that it revealed some severe problems with the user model.

Although Laurent Kerhoas had determined that Tcl was a good choice (I don't think he would say that today), I (i.e., Robert Strandh) was unsatisfied with this solution.

Richard Stallman had already (correctly, in my opinion) determined that (so called) scripting languages posed a problem because although they were supposed to be used as *glued* between modules written in a more efficient language such as C, and that their implementation was usually to slow for implementing basic functionality, users preferred them to the lower-level language. This created two major problems: first, applications written in the scripting language were slow (because of the poor implementation of the scripting language), and second, the part of the applications written using the scripting language was unmaintainable (due to the poor abstraction capa-

[1] A DESS is an obsolete degree (obtained after five years of study at the university) of the French educational system which has since been replaced by a *Master* degree

bilities of the scripting language). For that reason, he used a Lisp-derivative (Emacs Lisp) to create Emacs.

Emacs Lisp was created at a time when there were many, substantially different Lisp dialects. The choice was not obvious to Richard Stallman, so he created his own dialect, which was a reasonable solution at the time.

In 1994, there were two main dialects of Lisp remaining, Common Lisp and Scheme. At the time, I thought Scheme was more modern and that Common Lisp was going to lose, so I opted for Scheme.

The problem was that Scheme did not have any standard library for graphic user interfaces (determined essential for Gsharp); in fact it did not have any standard library for anything higher level at all. It was thus necessary to choose not only a language, but an implementation of a language. Based on my knowledge at the time, I choose Elk. It had a library for interfacing to the X Window System which was determined essential to Gsharp.

As it turned out, Elk had exactly the problems Richard Stallman had predicted: it was too slow to use for the main parts of the system, so one ended up writing major parts in C, which was too low level for this kind of software.

At about this time, the GNU project started promoting Guile, a new Scheme dialect to be used in GNU projects. The main idea of Guile was that, since it was a dialect of Scheme, and that Scheme is strictly more powerful that most other languages (mainly due to its first-class continuations), it would be possible to translate other languages (such as Tcl) in a fairly straight-forward way to Guile, and thus allow for any less powerful scripting language to be used as extension language for any future GNU application.

I jumped on the Guile bandwagon and converted Gsharp to Guile in around 1995 (with the help of some students, as usual of course). Unfortunately, the Guile project made some very unfortunate choices, such as basing it on Aubrey Jaffer's SCM. Although SCM was one of the fastest known interpreters for Scheme it was still an interpreter. And after the Guile team had added essentials (like classes and such) to SCM to obtain Guile, it was no longer very fast. In addition, Guile turned out to be a moving target for years to come, and often obliged me to rewrite parts of Gsharp that I thought were done. I did spend some effort on this, though, writing a library for accessing the X Window System and some other important infrastructure. Ultimately, Guile turned out to be too hard to track, and it was not clear that it was going to be fast enough to write an application in, nor that it would ever be stable enough for me.

For a while I was stuck and lost interest. After spending a sabbatical year in Austin (working with the RScheme research group), I realized that I had been distracted from the obvious choice of language many years before. I started looking into Common Lisp for the first time after having been mainly into Scheme since around 1985. It turned out that Common Lisp was the perfect choice for several reasons: it had a standard (ANSI); it had good, fast implementations that were free (in the GNU sense of the term); it had a semi-standard interface to the X Window System; it did not lack any fundamental features such as classes and exceptions.

So, I re-implemented Gsharp in Common Lisp starting late 1998. I now no longer had to worry about tracking a "standard" or whether my language implementation was going to be fast enough. And, I could safely use techniques from object-oriented programming without worrying whether my language implementation would support them.

But even this solution had a major problem. The interface to the X Window System, although very good, was not high level enough. I needed a real library for graphic user interfaces. Many such libraries existed, though not for Common Lisp at least not freely available. In fact Common Lisp had (and still does have) a semi-standard library called CLIM (Common Lisp Interface Manager) which turns out to be an order of magnitude more advanced than such libraries for other languages, and which has a public specification. The only (major) problem was that it had no freely available implementation. Common Lisp vendors had versions that they charged an arm an a leg for, and which would be unacceptable to Gsharp anyway, since not freely available.

I therefore decided to put the Gsharp project on hold in order to write a freely available implementation of CLIM. To make a long story short [2], it took a little more than three years, and lots of help from other contributors, to get the freely available implementation of CLIM to a state where it would be good enough to implement Gsharp.

Starting in 2002, I gradually integrated Gsharp with CLIM. Arnaud Rouaned worked on it for a while, but it was not until August of 2003 that I finally had time to spend time on it myself again. During the first two weeks of August, I

[2] This story is told in great detail in the McCLIM documentation, part of the McCLIM software distribution

re-implemented anti-aliased fonts, an improved version of the Obseq library, and a new system for graphics output based on CLIM recording streams.

Appendix B

The beaming algorithm

The beaming algorithm of Gsharp was inspired by the book by Ross. Unfortunately, his method is not easily converted int an algorithm, and his examples occasionally break his own rules. We thus had to invent our own algorithms that is as close to what we believe Ross intended as we could possibly achieve.

Generally speaking (we are simplifying the discussion to a single beam) the beaming problem takes as input the x- and y-positions of a number of clusters and computes the position of the endpoints of the beam to connect the clusters.

B.1 Beaming is symmetric

From studying the examples of Ross, one concludes that beaming is symmetric, i.e., the solution to a beaming problem is also a solution to the beaming problem obtained when the beam group is reflected either in the middle staff line, or in an arbitrary vertical line. This makes it necessary to handle only beaming problems with stems in one direction (say up) and with the first note (say) lower than the last of the beam group.

B.2 Only the notes closest to the beam count

Each cluster can be reduced to one with only a single notehead, namely the one closest to the beam. In our case, it would be the note on the highest staff position in each cluster.

B.3 Notes determining slope and position

The slope of the beam is determined only by the outermost clusters (though Ross shows some examples of exceptions to this rule). The position of the beam is translated vertically by an even number of staff steps, should any of the clusters in the group otherwise have too short a stem.

With these observations, we now only have to find the slope and initial position of a beam of a beam group consisting of two clusters, each with a single note and where the note of the leftmost cluster is on the same or a lower staff position than the one of the rightmost cluster.

B.4 General algorithm for single beam

In the description that follows, we use the Gsharp method of naming staff positions which is that we count in half staff steps above the lowest line of the staff). For example The first ledger line below the staff has position -2 and the upper line of the staff has position 8.

The primary beam can not be lower than the middle staff line (position 4), but it can hang under it.

Here is the rule we use (and which coincides with that of Ross in most cases):

- if the rightmost note is strictly lower than then the first ledger line below the staff (it has a position of at most -3), then the left side always hangs under the middle staff line and the right side either hangs under, straddles or sits on that same line. We vary the beam slope according to the position of the left note
  - if it has the same position as the right one, then the right side of the beam also hangs under the line

  - if it is one lower than that of the right note, then the right side of the beam straddles the line so that its slope reflects the difference in position
  - otherwise (i.e., the left note is more than one position below the right one), then the right side of the beam sits on the line.

- if instead the rightmost note is ON the first ledger line (position -2) we raise the beam a little, but it is still entirely on the middle staff line:
  - if the left note is also has position -2, both sides of the beam will straddle the middle line.
  - if the left note has position -3, then the beam hangs the line on the left and straddles the line on the right.
  - otherwise (the left note is at least two positions lower than the right one) the beam hangs on the left side and sits on the right side.

- if instead the rightmost note is below the lowest staff line (position -1) the beam is entirely ether on the middle staff line (position 4) or on the one above it (position 6).
  - if the left note has the same position (-1), then the beam hangs the line at position 6 at both ends.
  - if the left note is one position lower (-2) the beam is on the middle staff line (position 4), with the left side straddling it and the right side sitting on it.
  - otherwise (the left note is at least two positions lower than the right one) the beam is again on the middle staff line with the left side hanging under it and the right side sitting on it.

- if instead the rightmost note is on one of the normal staff lines (even position from 0 to 8 inclusive) then the right side of the stem always straddles a line (possibly not physically shown) six positions higher, which makes the right stem 3 1/4 staff steps (6 1/2 positions) long.
  - if the left note has the same position as the right one, the left side of the beam also straddles the same line.
  - if the left note is one position below the right one, the left side of the beam hangs under the same line.

  - if the left note is two positions below the right one, the left side of the beam straddles the staff line below the one that the right side of the beam straddles.
  - otherwise (i.e., the left note is at least three positions lower than the right note, the left side of the beam hangs under the staff line below the one that the right side of the beam straddles.

- if instead the rightmost note is between two ordinary staff lines (odd positions from 1 to 7 inclusive), then the right side of the beam always sits on a staff line 5 positions above the right note, which makes the right stem 3 staff steps long (6 positions).

  - if the left note has the same position as the right one, the left side of the beam also sits on the same line
  - if the left note is one positions lower than the right one, the left side of the beam straddles the same line.
  - if the left note is two positions lower than the right one, the left side of the beam hangs under the same line.
  - otherwise (the left note is at least three positions lower than the right one), the left side of the beam straddles the staff line immediately below the one of the right side of the beam.

- if instead the rightmost note is above the entire staff on an even position (on a fictive ledger line above the staff), then the right side of the beam sits on a fictive line four positions above the note. That makes the right stem extremely short (2 1/2 staff step, or 5 positions)

  - if the left note has the same position as the right one, the left part of the beam also sits on the same line.
  - if the left note has a position one below that of the right note, the left part of the beam straddles the same line.
  - if the left note has a position two below that of the right note, the left part of the beam hangs under the same line.
  - if the left note has a position three, four or five below that of the right note, the left part of the beam straddles the line below the one on the right.
  - otherwise (the left note has a position of at least six below the right one), the left part of the beam hangs under the staff line below the one on the right.

- otherwise, i.e., the rightmost note is above the entire staff in an odd position (below a fictive ledger line above the staff) then the right side of the beam always straddles a (fictive) line five positions above the note. This makes the right stem 2 3/4 staff step (5 1/2 positions) long.
  - if the left note has the same position as the right one, then the left part of the beam also straddles the same line
  - if the left note has a position one lower than the right note, then the left side of the beam hangs under the same line
  - if the left note has a position two lower than the right note, the left side of the beam straddles a line one below the one on the right.
  - if the left note has a position three, four, five, or six lower than the note on the right, then the left side of the beam hangs under the line below the one on the right.
  - otherwise (the left note has a position at least seven lower than the note on the right), then the left part of the note straddles a line two staff steps (four positions) below the line on the right. This is the only case with such a high beam slant.

B.5 Algorithms for more than one beam

Double beaming adds more constraints, so it is easier.

Triple beaming adds even more.

With more than three beams, things get messy because we cannot preserve normal spacing between beams.

B.6 Avoiding too high a beam slant

When the horizontal distance between the two outer notes is too small, normal beaming cannot be used, since it would result in beam slants that are esthetically too great. Ross only gives examples, but from those examples, one may conclude that beam slants more than 18 degrees are not used. Ross also does not give any hints as to how to go about finding a correct solution in this case, he simply hints that (for stems up, upward beam) the left part of the beam must be raised a bit. Our solution is to recursively try a new beaming with the left note one position higher up, until a solution is found. A solution is always found since ultimately, when the who notes have the same position, a horizontal beam will always be the result, and a horizontal beam has a slant of zero degrees.

Appendix C

Placement of accidentals

The rules of Ross essentially boil down to the following:

- If there are no suspended notes with accidentals on the right side of the stem, place all the accidentals in the order of the topmost one that can be placed closest to the stem.
- If there are suspended notes with accidentals on the right side of the stem (stem is up and there is an interval of a second), start by placing the accidental of the topmost suspended note with an accidental as close as possible to the stem, then place the other notes according to the previous rule.

To implement this algorithm, we keep a table of smallest distances between accidentals and between accidentals and notehead. It is a tree-dimensional table indexed by the object (note or accidental) already in place, by the accidental to place, and the vertical distance between the two (which is a number between -6 (the object placed is below the object to place) and 6 the object placed is above the object to place) inclusive). The entry in the table indicates how much further to the left (in staff positions, i.e, half staff-line distances) the object to place needs to be from the object already placed.

Here is a plausible table:

Object already placed is a notehead:

To pl	-5	-4	-3	-2	-1	0	1	2	3	4	5
d flt	0	0	0	3.5	3.5	3.5	3.5	3.5	3.5	1	0
flat	0	0	0	3.5	3.5	3.5	3.5	3.5	3.5	1	0
nat	0	3.5	3.5	3.5	3.5	3.5	3.5	3.5	1	1	0
shp	0	3.5	3.5	3.5	3.5	3.5	3.5	3.5	1	1	0
d shp	0	0	0	3.5	3.5	3.5	3.5	3.5	0	0	0

Object already placed is a double flat:

To pl	-5	-4	-3	-2	-1	0	1	2	3	4	5
d flt	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3	3	0
flat	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3	3	0
nat	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3	3	0
shp	4	4	4	4	4	4	4	4	4	3.5	0
d shp	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3.8	0	0	0

Object already placed is a flat:

To pl	-5	-4	-3	-2	-1	0	1	2	3	4	5
d flt	2	2	2	2	2	2	2	2	1.5	1	0
flat	2	2	2	2	2	2	2	2	1.5	1	0
nat	2	2	2	2	2	2	2	2	1.5	1	0
shp	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	1.5	0
d shp	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	0	0	0

Object already placed is a nat:

To pl	-5	-4	-3	-2	-1	0	1	2	3	4	5
d flt	2	2	2	2	2	2	2	2	2	1.5	1.5
flat	2	2	2	2	2	2	2	2	2	1.5	1.5
nat	2	2	2	2	2	2	2	2	2	1.5	1.5
shp	2	2	2	2	2	2	2	2	2	2	2
d shp	2	2	2	2	2	2	2	2	1	1	1

Object already placed is a shp:

To pl	-5	-4	-3	-2	-1	0	1	2	3	4	5
d flt	0	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.0	1.5	1.0
flat	0	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.0	1.5	1.0
nat	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.0	1.5	1.0
shp	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.0	2.0
d shp	0	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	0	0

Object already placed is a double shp:

To pl	-5	-4	-3	-2	-1	0	1	2	3	4	5
d flt	0	0	2.4	2.4	2.4	2.4	2.4	2.4	2	1	0
flat	0	0	2.4	2.4	2.4	2.4	2.4	2.4	2	1	0
nat	0	0	2.4	2.4	2.4	2.4	2.4	2.4	2	1	0
shp	0	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	0
d shp	0	0	0	2.8	2.8	2.8	2.8	2.8	0	0	0

Appendix D

Release notes

D.1 Release 0.2 (2003-09-06)

D.1.1 Features added from 0.1

- The command for inserting a bar line ('|') now works correctlhy in that it insert the bar line where the cursor is and not right before the next bar line.
- Erasing and deleting a bar line now works as announced.
- Rests get inserted on the current staff line.
- It is now possible to move rests vertically on their staff lines.
- Multiple staves.
- Key signatures.
- A new note inserted inherits the accidentals of the staff line it is initially placed on. This avoid for the user to have to constantly add accidentals to newly inserted notes in key signatures with many sharps or flats.
- Reading and writing Gsharp files is better organized so that it will be easier to bump the file version while maintaining backward compatibility. I do not intend to maintain the ability to write older version

files, which unfortunately makes Gsharp behave like commercial products with planned obsolescence, the difference being that the upgrade of Gsharp can be had for free.

- Much improved cursor display. The cursor now reflects the current staff as well as the interval used to add new notes.
- Double buffering of output to avoid flickering.
- The reference manual now has a list of commands with associated keys.

D.1.2 Bug fixes from 0.1

- Creating a new buffer did not erase the old input state, in particular the staff of the old input state.
- There was a bug in the beaming algorithm that created stems of abnormal length.
- Fixed a bug in the computation of the dominating note of a cluster that made stems to short in beamed multi-note clusters with stems down.
- Empty clusters are now displayed correctly (i.e., not at all) without signaling a condition.

D.2 Release 0.1 (2003-08-24)

D.2.1 What works

Basic music notation

- Noteheads
- Stems with the right dimensions and the right position calculated automatically,
- Augmentation dots, though placed a bit too close to the notehead.
- A single main beam with the correct placement and slant for all cases.

- Correct placement of all noteheads and accidentals in arbitrarily complicated clusters.
- Automatic generation of as many ledger lines as called for.
- Barlines, though currently drawn as lines rather than as rectangles.
- Rests.

Implementation issues

- The Obseq library is in pretty good condition.
- The (fairly complicated) beaming algorithm for a single beam is very satisfactory.
- The (fairly simple) algorithm for determining when a note should be suspended seems to work.
- The algorithm for placing accidentals in complicated clusters is also very satisfactory.

User interface

- Most of the basic user interface is in place.

D.2.2 What does not work

Basic music notation

As the reader can tell, most of the basic music notation is still missing. Here is a (probably incomplete) list from Ross:

- Augmentation dots in the presence of suspended notes
- Multiple beams. This is not particularly hard to put in.
- Multiple staves.
- Staves do not have time signatures yet. This is a problem, because the file format must be changed to include that.

- Most accents.
- Slurs and ties. I don't even know how to do it at this point.
- Repeat bars
- Key changes in the middle of a slice.
- Brackets and braces.
- Some minor spacing decisions are wrong (beginning of the line to clef, clef to first note, etc)
- Triplets and generalizations thereof.
- Lyrics
- Fermata
- Dynamics
- Cue notes and grace notes
- Passing tones
- Ending signs
- Fingering
- Trill
- Glissando
- Arpeggio signs
- Tremolo
- Bowing and stroking marks
- Pedal marks
- Repeat signs
- Octaves
- Pause

Implementation issues

- Reorganizing output records to substitute different glyphs.
- Need to implement the Lime algorithm for distributing space (or lack thereof) on a line. Currently clusters with complicated stacks of accidentals will overlap preceding clusters.
- We need to present a consistent interface to the combination of buffer module and the cursor module, and then write all primitives in terms of that interface. This would be the interface that is presented to the extension writer. Using this interface, it should not be possible to break any of the invariants of the other interfaces, such as numbering of all entities and maintaining modified flags for redisplay. In particular, this interface does not contain the concept of a 'bar', which should remain internal to the buffer module.

User interface

- It would be nice to mark the current note in the score pane as opposed to in a separate pane. Idea: draw a yellow square background under the current note. It may be possible to get rid of the current element pane altogether.

Appendix E

Plans for the future

E.1 Minor issues

There are tons of minor issues to fix. Here is a partial list.

E.1.1 Notation issues

- Elements now have an additional property that lets the user position them horizontally with an offset relative to their normal aligned position, so that notes with an interval of a second form different layers do not overlap physically. But we also have to introduce a set of primitives to alter this offset, and perhaps some visual indications that show the offset. Perhaps we should show a thin yellow line where the alignment is whenever there is a nonzero offset. I am suggesting yellow, because it would not destroy the overall visual impression of the score.

- The cursor is currently drawn by the drawing routine for the layer, but that routine is not supposed to know where the cursor is. We need to separate out cursor drawing from layer drawing.

E.1.2 Implementation details

- Playing as MIDI should create a temporary file with a unique name in `/tmp` as opposed a file in the current directory with a name that can clash with others.

- Someone who knows how pathnames work needs to find out how to use them in place of strings for filenames.

E.2 Major issues

E.2.1 Implementation details

- Different information might need to be attached conceptually to barlines, in particular: fermata, repeat bars, etc. It would be natural to physically attach this information to the bar, indicating properties of the barline at the end of the bar, but there is a conceptual problem with this, since some of this information is not unique to the layer, but sometimes global to all layers that are traversed by the resulting barline and sometimes even (repeat bars?) to the entire score. One could imagine a separate structure for the segment, indicating positions of repeat bars and such. It would have the additional advantage that if a barline is deleted in a layer which has a repeat bar further on, the visual effect would be nicer. Operating on this new structure should be possible from any layer.

- The fermata symbol is sometimes centered on a beam (tremolo). We currently have no way of attaching any information to a beam,

E.3 Minor projects

E.3.1 Default staff line for rests

When using several voices (usually two) on one staff line, one would like for rests to be placed on (say) the upper staff line for the upper voice and (say) the lower staff line for the lower voice rather than, as now, having all rests inserted by default on the middle staff line. It would be nice if a layer had a default staff line for rests so that changing hte layer automatically changes the default staff line for rests.

E.3.2 Default stem direction for a layer

When using several voices on a staff line, one somtimes wants the stems of the upper voice to be up and the ones of the lower voice to be down. It would be nice for a layer to have a default stem direction so that changing the voice would automatically change the stem direction.

Here is a possible specification of such a feature:

A new element is still created according to the stem direction of the input state without taking into account the stem direction of the layer. When a layer is displayed, the following final stem direction is computed:

- if the stem direction of the element is not *auto* then that stem direction is used;
- if the stem direction of the element is *auto*, but the default stem direction of the layer is not *auto*, then the default stem direction of the layer is used;
- otherwise, the stem direction is computed from the position of the notes of the element.

E.4 Major projects

E.4.1 Menu items with arguments

This is a CLIM project. Currently CLIM does not know how to read arguments when a command has been invoked from a menu item. Consequently, we are restricted to menu items that have no arguments.

E.4.2 Layout by page

We need to have a more sophisticated layout algorithm that divides the score into pages rather than just lines. One interesting idea seems to be to use the Obseq library in a nested kind of way, where an invocation of the cost combination method on the global level would provoke another call to the `solve` function on the page level. This would be really cute, actually, and it would show the power of the approach of the Obseq library.

Here is how it would be done. First it requires a minor modification to the Obseq library. Instead of the elements *inheriting* from `obseq-elem`, they need to be separate so that an element can be contained in more than one `obseq`. This modification has some consequences: the operations to compute the next and previous element need to take an `obseq` argument to dispatch on. The operations `last-undamaged-element` and `first-undamaged-element` must be modified and renamed to (say) `tail-valid-elements` and `head-valid-elements` and take a number instead of an element, since the `obseq` itself does not have a mapping from its internal elements to the client element. This requires the client to know how many elements there are, which is a minor inconvenience (but still an inconvenience). Then the seq-cost on the global level will be an obseq on the page level, and a call to combine a seq-cost and an element on the global level would trigger a call to solve on the page level.

We also need to have different cost combination strategies, for instance:

- Paper roll with last line not right justified. This involves assigning a low cost to the last line even when it has to be highly stretched in order to fill the line.
- Paper roll with last line right justified. This is all we have right now.
- Pages with last page not filled and last line on last page not filled.
- Pages with last page filled normally.
- Pages with a fixed number of filled pages.
- Pages with an arbitrary number of pages, but the number must be of the form $n = ax + b$.

Appendix F

CLIM terminology

Talk about frames, panes, streams, etc.

Index