The MUS Music System Manual

by

Martin Richards

`mr@cl.cam.ac.uk`

`http://www.cl.cam.ac.uk/~mr/`

Computer Laboratory
University of Cambridge

Revision date: Tue May 7 10:58:36 BST 2019

Abstract

This document is being worked on again but is still in the early stages of development.

MUS is a language for describing music, primarily for performance and not for typesetting. It is currently in the early stage of development. It will eventually allow the user to specify, in great detail, not only the notes to be played but also subtleties of timing, phrasing, touch and many other aspects of the performance. It will be the source language for `playmus`, a program to read .mus files and play them on MIDI devices such as a Roland or Casio electronic piano or the Roland SonicCell. Playmus is also able to generate MIDI files. The program will eventually be able to accompany a soloist using a microphone microphone synchronisation. In the distant future, `playmus` may be able to listen to an ensemble and tactfully fill in missing parts when players are absent or get lost.

Currently playmus only uses MIDI and works best with digital pianos.

Keywords

Contents

Preface

(While the MUS language and `playmus` are being developed this is essentially just a working document outlining the current state of the system. It is incomplete and may describe features that are not yet implemented. As might be expected the language keeps changing in the light of experience. During its development it is likely that `playmus` will be somewhat bug ridden.)

This document is intended to provide a description of the MUS language and the program `playmus` that uses it. Even in their current incomplete state, this manual `musman.pdf` is freely available via my home page [2] and `playmus` is a standard command included in the standard BCPL distribution. Demonstration `.mus` files and their MIDI and .mp3 translations can be found in the Musprogs distribution in my home page.

The `playmus` program is being developed in BCPL under Linux but also runs under Windows XP and Windows 10. Currently I can only connect BCPL to my various MIDI devices under Windows.

A further restriction is that the digital signal processing of the microphone input will probably take advantage of the BCPL MC package that dynamically generates native machine code, but this is currently only available on Pentium based machines under Windows, MAC OS-X, Cygwin and Linux.

Chapter 1

The MUS Language

The MUS language is a text based representation of music allowing the detailed description of musical compositions involving separate instrumental parts. Its design was influenced by the excellent freely available music typesetting system Lilypond [1]. However MUS was designed for performance rather than typesetting and so contains features that allow subtle variations of tempo, volume, phrasing and many other nuances of the performance. The language is primarily designed to generate satisfactory musical performances, but can also be used to provide a synchronised accompaniment to a soloist.

The language has a built-in macrogenerator (or preprocessor) based on GPM originally designed by Christopher Strachey. The resulting text is then broken into lexical tokens before being parsed to form a tree. This is finally translated into sound events. The `playmus` program performs this conversion and either writes the sound events as a MIDI file or plays them directly on a MIDI device, possibly using a microphone to synchronise them with a soloist.

1.1 The Macrogenerator

The macrogenerator is based on GPM designed in 1964 by Christopher Strachey [3]. It is somewhat ideosyncratic but is simple and easy to implement. This version uses the following special warning characters.

Character	Purpose
$	Start of macro call
!	Macro argument separator
;	End of macro call
<	Open quote
>	Close quote
#	Copy argument character
%	Comment character

Normally all other characters pass straight through the macrogenerator. On encountering the start of macro call character (`$`) output is diverted to an internal stack to accumulate the macro's arguments. Arguments are separated by exclamation marks (`!`) and the last is terminated by the macro call character (`;`). When all the arguments have been read, the macro is called.

The arguments are numbered from zero with the zeroth argument being the name of the macro to call. It is looked up in the current list of defined macros and processing continues by scanning its body. If, within the body, a hash sign (`#`) is encountered, it will be followed by digits specifying the number of the argument to copy. Thus `#0` will be replaced by the name of the current macro, `#1` will be replaced by a copy of the first argument, and so on. Note that `#12` will copy the 12th argument, but `#1<2>` will copy argument 1 immediately followed by the digit 2.

It is sometimes necessary to treat special characters as though they were ordinary. This is done by enclosing text in quotation marks (`<` and `>`). All characters between the open quote character (`<`) and the matching close quote character (`>`) are copied as though they were ordinary characters. So, if the macrogenerator was processing the string `<$abc!<#1>xyz;>`, it would generate the text `$abc!<#1>xyz;` without treating `$`, `!`, `<`, `>` or `;` as special. Quotation marks can be nested.

When the macrogenerator encounters the comment character (`%`) from normal input, it skips over this character and all characters up the end of the line. It then skips over white space characters, including newlines, until it finds the next non white space character. If this is the comment character, the macrogenerator will skip over another comment. This form of comment is often preferable to the MUS `//` comment since the text following `//` may contain macro calls that expand to multiple lines, of which only the first will be commented out.

At the start of processing a number of builtin macros are predefined. These are: `def`, `set`, `get`, `eval`, `char`, `rep`, `rnd`, `urnd`, `lquote`, `rquote`, `comment`, and `eof`. These are described below.

`$def!`*name*`!`*body*`;`

This causes a macro with name *name* and body *body* to be added to the list of defined macros. The macro body is normally enclosed in quotation marks since it often containing macro calls and hash signs. As an example, the following sequence of definitions and calls:

```
$def!ww!World;%
$def!aa!Andy;%
$def!abc!<#1 $#2;>;%
$abc!Hello!ww;
$abc!Hi!aa;
```

would generate:

```
Hello World
Hi Andy
```

Note that the call `$#2;` within the definition of `abc` calls the macro `ww` the first time and `aa` the second time.

There is one subtlety about the `def` macro. If it is used within the argument list of a macro call, the new definition is removed just after the call has been expanded. The following example shows that this allows conditional macros to be defined without permanently corrupting the list of defined macros.

```
$def!ifeq!%
  <$def!=#1=!#4;%
   $def!=#2=!#3;%
   $=#1=!#1!#2;%
  >;%
$ifeq!xx!xx!<#1 is the same as #2>!<#1 is not the same as #2>;
$ifeq!xx!yy!<#1 is the same as #2>!<#1 is not the same as #2>;
```

The above fragment generates:

```
xx is the same as xx
xx is not the same as yy
```

This example works because a new definition of a macro supercedes older ones with the same name.

`$set!`*name*`!`*value*`;`
  This copies *value* into the body of the macro with name *name*. This is rather like an assignment in a conventional programming language. The maximum length of the value of *name* is the length it had when first defined. The argument of `set` is truncated to this length, if necessary.

`$get!`*filename*`;`
  This temporarily suspends the currently input stream and selects input from the specified file. When this file is exhausted input resumes from the previously selected stream. It is as if the call `$get!`*filename*`;` were replaced by the text of the specified file. The file name is first looked up in the current working directory then in the directories specified by the `MUSHDRS` environment variable.

`$eval!`*expression*`;`
  This evaluates its argument which must be an expression composed of numbers, character constants, parentheses, and the operators `+`, `-`, `*`, `l`, `r`, `/`, `&` and `|`. It uses scaled fixed point arithmetic with three decimal digits after the decimal point. Numbers such as 4, 3.14, 1.818 or .5 are allowed. A character constant is represented by a single quote (`'`) followed by a single character. It generates the

ASCII integer code for the given character. For example, `$eval!'A;` generates
`65`. If the result of `eval` is an integer, it appears as such, otherwise it is a decimal
number with leading and trailing zeros removed. Thus `$eval!1.5+2.5;` gener-
ates `4` while `$eval!12.5/2;` generates `6.25`. This form of scaled arithmetic is
appropriate the numbers used in the MUS language. The operators `l`, `r`, `&` and
`|` first truncate their operands to integers before performing bitwise Left shift,
Right shift, And and Or operations, respectively. The result is then converted
back into scaled decimal form.

`$rnd!`*expression*`;`
      This evaluates the expression yielding a scaled number n, say, which must be
greater than zero. The result is a uniformly distributed random number in the
range $-n$ to n.

`$urnd!`*expression*`;`
      This evaluates the expression yielding a scaled number n, say, which must be
greater than zero. The result is a uniformly distributed random number in the
range 0 to n.

`$rep!`*expression*`!`*value*`;`
      This evaluates the expression rounding it down to the nearest integer which
specifies how many copies of the value are to be generated.   For example,
`$rep!4-1!xy;`

`$char!`*expression* `;`
      This evaluate the given expression and converts it into an ASCII character.
For instance, `$char!'A+1;`expands to `B`.

`$lquote;`
`$rquote;`
`$comment;`
      These generate the the characters `<`, `>` and `%`, respectively

`$eof;`
      This generates and end-of-file character.

1.2   Lexical analysis

The stream of characters from the macrogenerator are converted into a sequence
of tokens by the lexical analyser.
      All characters from `//` until the end of the current line are removed by the
lexical analyser as are multi-line comments enclosed in `/*` and `*/` brackets. Such
multi-line comments may be nested. Note that macrogenerator comments intro-
duced by percent (`%`) have already been removed. Be aware that the text following
`//` may contain macro calls that expand to more than one line with only the first
being treated as a comment.

The remaining (non commented out text) is converted into lexical token described below.

1.2.1 Numbers

A number is a possibly signed fixed point decimal with at most three digits after the decimal point, such as: `4`, `-123.456`, `24.45` or `.001`. Numbers are used in shape sequences to specify values such as volumes or tempos. In note sequences, they are only used to specify octave numbers. The global `numval` is set appropriately as a scaled fixed point value. For instance `-123.456` would cause `numval` to be set to `-123456`.

1.2.2 Notes

A note consists of a note letter (`a` to `g`) optionally followed by accidentals (`es`, `eses`, `is` or `isis`) representing a flat, double flat, sharp or double sharp, respectively. This is followed by an optional octave adjustments consisting of single quotes (') or commas (,) to raise or lower the note by a number of octaves. An optional length number (`0`, `1`, `2`, `4`, `8`, `16`, `32`, `64` or `128`) follows to specify the nominal length of the note. The note lengths in qbeats are respectively 8192, 4096, 2048, 1024, etc. If no length number is given, it has the same length number as the most recently occuring note, rest or space. Thus, `c4 d e f` is a sequence of four crotchets (or quarter notes). A note may end with one or more dots having the same meaning as in conventional music notation. Thus `c4..` is a double dotted crotchet with a duration of 7 semiquavers. Finally, a note may end with a tilde (`~`) indicating that it is tied with a later note with the same pitch. For more information about ties see Section 1.3.4. Spaces characters are not allowed within note tokens.

The globals `noteletter`, `notesharps`, `reloctave`, `notelengthnum` and `notedots` are set appropriately.

1.2.3 Rests

Rests are represented by `r` optionally followed by a length number and possibly followed some dots. Thus `r4..` is a double dotted crotchet rest of length 7 semiquavers. Spaces are similar to rests but use the letter `s`. The only difference is that, if the music were to be typeset, rests would appear as rest symbols while spaces are just blank. The globals `notelengthnum` and `notedots` are set appropriately.

A special space of zero length is represented by `z`. It usually occurs as the left hand operand of dyadic shape operators, as in `z\vol50` which specifies that the volume is 50 at this point of the note sequence. It is also useful in shape lists when an instantaneous change of shape value is required.

1.2.4   Barlines

Bar lines are represented by vertical bars (`|`) and double bars by `||`. They are primarily used as a debugging aid to check that bars are of the correct length, but are also used by some keyboard commands while `playmus` is accompanying a soloist. The first bar of a piece has number one and the bar number is incremented at each barline. A warning is given if the position of a barline in a part does not agree with the bar layout specified by the conductor part.

Repeat marks are represented by the special symbols `\colon_bar`, `\bar_colon` and `\colon_bar_colon`, although these are normally invoked using the macros `$:|;`, `$|:;` and `$:|:;`. As in normal musical notation these may occur in the middle of a bar and do not increment the bar number.

1.2.5   Strings

Strings are encloses in double quotes (`"`) and may contain the following escape sequences `\\`, `\'`, `\"`, `\t`, `\n` to repesent the characters `\`, `'`, `"`, tab and newline, respectively. The global `stringval` points to the packed characters of the string. String are used in various constructs such as those giving composer or instrument names.

1.2.6   Other tokens

Parentheses `(` and `)` are used in note and shapes sequences for grouping, and `[` and `]` are used to enclose simultaneous note sequences and chords. Curly brackets `{` and `}` are used in note sequences to enclose blocks that limit the scope of shape data, as described in Section 1.3.14. The tokens `*` and `*˜` are used in shape sequences as are also described in Section 1.3.14.

There are several built-in named tokens consisting of a backslash (`\`) followed by letters and underscores. These are as follows: `\altoclef`, `\arranger`, `\bank`, `\barlabel`, `\bassclef`, `\colon_bar_colon`, `\colon_bar`, `\composer`, `\conductor`, `\control`, `\delay`, `\d`, `\delayadj`, `\da`, `\instrument`, `\instrumentname`, `\instrumentshortname`, `\interval`, `\keysig`, `\legato`, `\l`, `\legatoadj`, `\la`, `\major`, `\minor`, `\nonvarvol`, `\opus`, `\part`, `\patch`, `\pedon`, `\pedoff`, `\pedoffon`, `\portaon`, `\portaoff`, `\score`, `\softon`, `\softoff`, `\tempo`, `\t`, `\tempoadj`, `\ta`, `\tenorclef`, `\timesig`, `\title`, `\transposition`, `\trebleclef`, `\tuplet`, `\tup`, `\vibamp`, `\vma`, `\vibrate`, `\vra`, `\volmap`, `\vol`, `\v`, `\voladj`, `\va` and `\varvol`.

1.3 The syntax

1.3.1 Scores

MUS files consist of a sequence of one or more scores each representing separate musical a composition or movement. Scores may be separated items such as \interval(2.5) specifying a 2.5 second period of silence between scores. Syntactially, a score has the form:

\score *string* [*part-list*]

The string is the name of the score and the *part-list* is a collection of parts that are played simultaneously. A part is analogous to a stave in an orchestral score. Often each player has a single part but sometimes two players may share the same part (eg two oboes on the same stave), and it is usual use separate parts for a pianist's two hands. Most parts start with the keyword \part followed by a note sequence enclosed in parentheses. One part is introduced by \conductor instead of \part. The conductor part contains no sounding notes but provides the bar layout of the composition and controls tempo variations and other features of the performance. Every score must have a conductor part and at least one other part. The word \part is replace by \solo for the part played by the soloist. When playmus asked to play an accompaniment it compare notes expected in the solo part with sound received by the microphone and synchronises the accompaniment appropriately. Normally playmus will only play a solo if that solo line is not detected. It attempts to do this tactfully.

The playmus program generates Midi data using a different Midi channel number for each part. Midi only allows 16 channels and so the number of parts in a MUS score is limited to 16. Although percussion is not yet available in MUS, it will use the conventional Midi channel number 10 further reducing the number of instrumental parts available. If and when playmus generates its own sounds without the aid of Midi, these restrictions will be lifted.

1.3.2 Quantum beats and milli-seconds

A crotchet (or quarter note) often has the length of one beat and a quaver half a beat, but these lengths should not be confused with the time they take to play, since this depends on the tempo and how the notes are to be played (legato, staccato etc). In the MUS language the nominal length of a crotchet is 1024 qbeats (quantum beats) where one qbeat is the finest grain of beat time available. Since 1024 is a power of 2, all normal notes, such as quavers and semiquavers, have integral numbers of qbeats. The finest grain of real time is the milli-second and so, at a tempo 60 crotchets per minute (or 1 crotchet per second), each qbeat lasts $1000/1024 = 0.977$ msecs.

1.3.3   Note items

Musical compositions are built from sequences of note items such as clefs, time signatures, key signatures, notes, chords, rests and barlines. There are other note items that provide low level control of the Midi devices that `playmus` currently uses to generate sound. The MUS language has textual representations of these items and they are all described below.

Sequences

A list of note items separated by spaces or newlines and enclosed in parentheses. They allow, for instance, a sequence of notes to be the left operand of a shape operator such a `\vol`.

Parallel groups

If square brackets (`[` and `]`) enclose a list of note items, each term will be played simultaneously. Thus `[c4 e g]` represents a C major triad of crotchets. The items in this construct can themselves be chords or sequences as in `[c4 ([f8 a] [e g])]` which means the crocket `c` is played while the two quaver chords `f` and `a` followed by `e` and `g` are played. A warning is given if the items in a parallel group are not all of the same qbeat length.

Blocks

If curly brackets (`{` and `}`) enclose a list of note items, the construct is called a block. It behaves like a sequence but has the additional property that it limits the scope of shape data specified within it to control such features as tempo and volume. The user can specify how the volume should change as the notes within the block are played. This takes the form of a graph consisting of line segments which together provide a mapping from the qbeat position in the composition to the corresponding volume. This information is called shape data and in held in the block. Such data only affects the playing of notes within that block. The user can explicitly create blocks using curly brackets but most blocks are created implicitly. For instance all the elements of a parallel construct are blocks. For more information see Section 1.3.14.

Notes

A note is represented by a note token (such as `cis'4..`) as described in section 1.2.2. It specifies its letter name, the number of sharps or flats, the number of octaves up or down, its length number and the number of dots. The length number is 1, 2, 4, 8, etc corresponding to notes of length 4096, 2048, 1024, 512

qbeats etc. If no number is explicitly given, the length number of the most recent note is used. The number of dots extends the length and in normal music notation.

If a note is not qualified by single quotes or commas, it will have a pitch that is no more than a fourth away (ignoring accidentals) from that of the most recent note in the text. Thus, if the previous note was a `c` then the notes `d`, `e` and `f` would be higher on the stave, and the notes `b`, `a` and `g` would be lower. The rule depends only on the note letters and is not affected by any accidentals added by `es`'s or `is`'s. So, for instance, `ces` followed by `fis` is still a rising interval even though it sounds like a perfect fifth. If this rule does not place the note in the desired octave, it can be corrected using of single quotes or commas.

It is sometimes convenient to specify the octave of a note explicitly. There are 10 octaves (numbered 0 to 9) with the notes from middle C to the B above having octave number 4. The octave below has number 3 and one above has number 5. If a number appears in a note sequence, it specifies the octave number of the next note. So the number 4 behaves as if the preceeding note was `f` in octave 4, so that `c`, `d` and `e` will be in octave 4 below this `f` and `g`, `a` and `b` will be in octave 4 above. There is no need to have a space between the octave number and the note letter. For example the first few notes of "Twinkle twinkle little star" could be input as `4c4 c 4g g a a g2` or alternatively `4c4 c g' g a a g2`.

1.3.4   Note ties

Notes are sometimes tied across bar lines or even within a bars, causing the first note to continue playing until the end of the second without being struck or tongued again. The first note must be immediately followed by a tilde (`˜`) to indicate that it is the start of a tie. In the MUS language identifying which notes are tied to which is not always obvious and requires an understanding of note threads.

A sequence (typically enclosed in parentheses) specifies a thread of note items that are performed from left to right as in normal printed music. The thread thus has a direction corresponding to increasing time. A parallel construct (enclosed in square brackets) may occur in a sequence but splits the thread into multiple threads, one for each component of the construct. Each such thread starts from the open bracket (`[`) and ends at the closing bracket (`]`).

For a two notes to be tied, the first note token must have the tie qualifier (`˜`), the second must have the same pitch (Midi note number) as the first. It must also start immediately after the end of the first (ignoring both legato and delay effects) and both must belong to the same instrumental part. In addition, there must be a forward going thread path from the first note to the second. This thread constraint causes the second note of a tie to be textually later than the first and not in a different parallel sequence.

Rests

Rests (eg `r4.`) and spaces (eg `s2`) represent periods of silence with a qbeat length deduced from its length number and dots. If the music is to be typeset a rest generates a rest symbol while a space is just blank. A zero length rest (`z`) is useful as the left operand of some shape operators.

Tuplets

The construct

note-sequence `\tuplet` *note-sequence*

causes the two note sequences to be played simultaneously with the second scaled to have the same number of qbeats as the first. Thus, `(4c8 e)\tuplet(g8 a b)` would play the quavers `c` and `e` while playing the triplet quavers `g`, `a` and `b`. If only the triplet were required, the left hand operand could be `s4` as in `s4\tuplet(g8 a b)`. The operator `\tuplet` can be shortened to `\tup`.

Transposition

For transposing instruments such as clarinets and horns it is often convenient to input the music using the notes the player reads rather than concert pitch. For instance, the player of a horn in F sees a written C it should sound as F, a fifth below. The note item `\transposition(f,)` will allow notes, from now on, to be input as the horn player sees them. The transposition of one tone down for a Bb clarinet would be specified by `\transposition(bes)`. Notice that the tranposition is always relative to C and the octave is corrected using single quotes and commas (as with ordinary notes). The argument of `\transposition` is not a played note and so does not affect the octave of the next note to be played.

Time signatures

The time signature is specified by `\timesig(`*number number*`)`. For example, `\timesig(3 4)` sets a time signature of three crotchets per bar. A check is made when processing the conductor's part that bar lines and time signatures only occur just before beat one.

Barlines

Single and double bar lines are represented by the tokens `|` and `||`. They are used in the conductor's part to provide a mapping between bar numbers and corresponding qbeat positions. There is a similar mapping between beat numbers and qbeat positions. Bar and beat numbers used in error messages and by some keyboard commands when `playmus` is accompanying a soloist. Bar lines are optional in the other parts but, as a debugging aid, they must occur at the qbeat

positions of barlines in the conductor's part. It is therefore advisable to put all barlines in every part.

Repeat marks may appear in note sequences. These are: `\colon_bar` to mark the end of a repeated section, `\bar_colon` to mark the start of a repeated section and `\colon_bar_colon` to mark both the start and end of repeated sections. Repeat marks only affect the printed form of the music, and they do not cause the bar number to change.

Key signatures and clefs

The key signature is specified by note items such as `\keysig(f\major)` or `\keysig(b\minor)`. The following note items specify the clef: `\trebleclef`, `\altoclef`, `\tenorclef` or `\bassclef`. Key signatures and clefs are ignored by `playmus`.

Text items

Text can be associated with a score, parts and bars using the following items.

Item	Purpose
`\title` *string*	The title of the score
`\composer` *string*	The composer
`\arranger` *string*	The arranger
`\opus` *string*	The opus number
`\instrument` *string*	The instrument used by this part
`\instrumentname` *string*	The full instrument name, eg: Horn 1
`\instrumentshortname` *string*	The shortened instrument name, eg: Hn1
`\barlabel` *string*	A bar label

1.3.5 Control items

At present `playmus` generates sounds using a variety of Midi devices. To provide low level device dependent control some special control items have been provided.

Pedals

The items `\pedon`, `\pedoff` and `\pedoffon`, control the sustain pedal of piano instruments. When the pedal is on good Midi devices will simulate the effect on a piano including causing sympathetic resonance of other strings on the instrument. The command `\pedoffon` is provided for convenience since it is common to raise the pedal and almost immediately press it again when the harmony changes.

The items `\softon` and `\softoff` simulate the effect of pressing and releasing the soft pedal of a piano.

The items `\portaon` and `\portaoff` turn the portamento feature on and off. When portamento is on, the transition from one note to another rapidly sounds all the frequencies between them. When portamento is off the transition is abrupt.

Midi control items

The item `\bank(x y)` set the bank number for the current Midi channel as specified by the two 7-bit integers x and y. The item `\patch`n sets a new patch number for the current Midi channel. The item `\control(n x)` cause the Midi device to set controller n to value x. Both n and x are integers in the range 0 to 127. The effect of these operations is Midi device dependent.

The item `\varvol` is used to specify that an instrument such as a flute can change its volume while a note is being played, and `\nonvarvol` specifies that the instrument is like a piano where the volume of a note is dependent only on how hard the note is struck. These control items are typically only used in the predefined macros used to specify instruments as in `g/gshdr.mus` or `sonichdr.mus`.

1.3.6   Shape operators

Tempo, volume and other aspects of the performance are controlled by shape data of various kinds introduced by shape operators such as `\tempo` or `\vol`. Shape data consists of a sequence of shape pairs (q, x) where q is a qbeat value and x is the corresponding shape value. In practice only the x-values are given since the corresponding q-positions are deduced implicitly. An example shape might be: `(50~ s4 80~ s4 50)`. Since `s4` has length 1024 qbeats, this shape corresponds to the pairs: $(0, 50)(1024, 80)(2048, 50)$. This could be used to specify that the volume increases linearly from 50 to 80 during the first crotchet and then decreases back to 50 during the second. If two numbers occur consecutively they are separated implicitly by a space of the same length as the previous space if any, or `s4` if not. So the above shape sequence could have been written as `(50~ 80~ 50)`. Without a tie mark (`~`) the shape value remain constant until the next explicit change. Thus, `(50 80 50)` corresponds to following pairs: $(0, 50)(1023, 50)(1024, 80)(2047, 80)(2048, 50)$.

Within a shape sequence, the special shape values `*` and `*~` are allowed. They denote the value of the same kind of shape looked up in the enclosing block environment as described in section 1.3.14. Shape sequences containing only one shape item do not need to be enclosed in parentheses.

It is sometimes convenient to scale the shape values, particularly when specifying tempos. This is done using and colon item such as

`:s4.`

whose qbeat length is 1536. This causes all subsequent shape values to be multiplied by 1536/1024. This scaling is useful, for instance, in `\tempo(:s4. 76)` meaning that the tempo is 76 dotted crochets per minute rather than having to write the equivalent `\tempo104`. Numbers can be used instead of space items, for instance, `:1536` and `:s4.` are equivalent.

Shape sequences normally occur as the right hand operand of dyadic shape operators such as `\vol` or `\tempo` and their q-values are always scaled so that the qbeat length of the sequence is the same as the length of the of the left operand. For example, `(c4 d e f)\vol(50~ 80~ 50)` corresponds to four crotchets with the first starting at volume 50, increasing linearly until reaching volume 80 at the start of the this crochet (`e`). The volume then decreases reaching 50 at the end of the fourth crotchet (`f`). If the shape sequence has only one item it is padded on the right with with `s4` before scaling.

The only non dyadic shape operator is `\volmap` and its q-values are scaled to the range 0 to 100. For more details see section 1.3.12.

1.3.7 Tempo

The tempo of a composition and indeed separate parts typically vary as they are played. This variation can be specified using the `\tempo` construct. Its left hand operand is a note sequence to be played while its right hand operand is a shape sequence whose values represent tempos, typically measured in crotchets per minute. This allows the nominal tempo at any qbeat position to be determined, but this tempo is adjusted by multiplying it by the percentage value given by the `\tempoadj` shape data. The scope of the tempo data is the current block, but, unlike other kinds of shape, tempo values are further scaled to cause the total playing time of the block to be the same as if it contained no tempo data.

Normally the conductor part gives the tempo mapping for the entire composition with all other parts obeying the conductor's wishes. However, the `\tempo` construct can be nested inside another `\tempo` construct. If this happens, the note sequence of the inner construct will have the tempo values of its shape scaled to cause the playing time (in msecs) of the note sequence to be the same as for the unqualified note sequence. This allows a sequence notes to be played with a degree of rubato while ensuring that it starts and ends in strict time.

The operators `\t` and `\ta` are synonyms of `\tempo` and `\tempoadj`.

1.3.8 Delay

The `\delay` operators specifies the delay in qbeats before a note should be sounded. A delay value of 1024 causes a delay corresponding to one crotchet, and -1024 causes a note to be played one crochet early. An example of its use is `c4\delay512~ d16 e f g a b c4\delay0` which cases the first note to be delayed by the time equivalent to a quaver. The subsequent six semiquavers are

delayed by successively smaller amounts until the end of the final c is reached exactly on time. The corresponding adjustment operator is \delayadj and both may be shortened to \d and \da.

Note that, if the delay value reduces rapidly, notes may unexpectedly sound in reverse order. To avoid a note being stopped before it has started, every note has a real time duration that is positive, typically at least a twentieth of a second.

1.3.9   Volume

The volume envelope of a note sequence can be specified using the construct: *note-sequence* \vol *shape-sequence*. The qbeat length of the shape sequence is scaled to the length of the note sequence. Shape values are in the range 0 (silence) to 100 (full volume). For wind and string instruments the volume can change while a note is sounding, but for pianos the volume value only affects how hard a piano key is struck. The \voladj construct allows percentage adjustments of the current volume shape values. These operators may also be written as \v and \va.

1.3.10   Legato

The legato operators allow the user to specify what proportion of the nominal qbeat length of a note should be played. For example (4c4 c g' g)\legato50 would cause the four crotchets to be played for 50% of their nominal length. This effect could have been obtained (less conveniently) by (4c8 r c r g' r g r) The playing time of a note depends on the nominal length of the note and the legato value at its start. It is not affected by changes in the legato value while the note is played. The \legatoadj operator can be used to modify legato values by percentage amounts. These operators may also be written as \l and \la.

1.3.11   Vibrato

The vibrato operators are not yet implemented but will probably be as follows. The construct: *note-sequence* \vibrate *shape-sequence* specifies rate of the vibrato in cycle per second, and *note-sequence*\vibamp *shape-sequence* specifies the amplitude of the vibrato with 100 representing vibrato between plus and minus a semitone. These operators may also be written as \vr and \vm. The adjustment operators are called \vibrateadj and \vibampadj with shortened forms \vra and \vma.

1.3.12   Volmap

The \volmap operator is the only non dyadic shape operator. It *q*-values are scaled to the range 0 to 100 and is used to scale volume values. The default volume

map is the identity function `\volmap(0~ 100)`, but a more compressed volume range can be specified by a setting such as `\volmap(0~ s8 30~ s1 70~ s8 100)`. The volume map applied to all notes in a part from the time it is set until the end or the next `\volmap` statement. Each part starts with the default volume map.

1.3.13 Shape ties

A shape value can be terminated by a tilde (`~`) to indicate that it is tied to the next shape value. This causes the value to change linearly between the two given values. Consider the following example.

```
4c4\vol50~ c g'\vol80 g |  // bar 1
 a a g2                 |  // bar 2
 f\vol70~ f e e         |  // bar 3
 d d c2\vol50           || // bar 4
```

Here, the first c starts with volume 50. The volume then increases to 80 on the third beat (g). This volume continues until the start of bar three when it suddenly reduced to 70 then gradually reduces to 50 on the minim c on the third beat of bar 4. This c remains at volume 50 throughout its two beats.

A shape sequence containing just one value and no space symbols has `s4` inserted at its end. So `(120~)` is equivalent to `(120~ s4)` and `(120)` is equivalent to `(120 s4)` which is equivalent to `(120 120)`.

More details of the exact interpretation of shapes are given below when the various shape operators are described.

1.3.14 Shape environments

Notes within a score occur within an environment giving the mapping between qbeat value and shape value for each kind of shape (tempo, volume, legato, etc). The default environment at the outermost level give a tempo of 120 crochets per minute, a volume of 70, a legato value of 90 and a zero delay. The default values for `\vibrate` and `\vibamp` are 4 cycles per second and 25 corresponding to about a quarter of a semitone. All adjustment values are 100 giving no change. The next environment level is specified by the conductor part. This overrides the default settings and provides the environment for each of the other parts in the score.

Within a part, a sub-environment can be created explicitly using a block enclosed in curly brackets (`{` and `}`). Any shape data occurring within a block overrides the outer environment but has its scope limited to that block. Most blocks are created implicitly as follows. The conductor and all other parts are

blocks. The components of parallel constructs are blocks, and the left hand operand of `\tuple` is a block.

Every block has starting and ending absolute qbeat positions. To lookup the value of a particular shape within a block, the following steps are taken. If the block has no shape data of the right kind, the lookup searches the enclosing environment block, otherwise the appropriate shape data in the block is inspected. The special shape value * has a value corresponding to its absolute qbeat position looked up in the environment of the block one level further out. These shape values are sometimes inserted automatically. If first shape item of a block does have the qbeat position corresponding to the start of its block, *~ is inserted at the start. If its last item does not occurs at the end of the block, * is appended at the end.

To look up the shape value associated with a given qbeat position, the shape data is searched to find the nearest shape pairs that enclose the given position and the shape value computed from these using linear interpolation, thus shape data is essentially a list of line segments. In the unusual situation when the given qbeat position is outside the block (possibly caused by a `\delay` construct), the value returned is taken from the first or last shape pair, whichever is appropriate.

Adjustments

All dyadic shape operators have variants that allow the shapes to be adjusted by effectively multiplying two shapes together. If at a particular qbeat position the `\vol` value was 60 and the `\voladj` value was 50, the the resulting volume at that moment would be 30, being 50% of 60. Both the basic shape and adjustment values are looked up using the mechanism described in the previous section.

1.3.15   Nested shape operators

It is syntactically possible for a shape operator such as `\vol` to be nested, as in

```
( 4c4 c g’\vol90 g )\vol(50~ 70)
```

but this is disallowed since its interpretation is unclear. The user should either place all the shape values at the same level as in

```
( 4c4\vol50~ c g’\vol90 g z\vol70)
```

or

```
( 4c4 c g’ g )\vol(50~ s2 90~ s4 70)
```

or protect the inner shape by enclosing it in a block, as in

```
( 4c4 c {g’\vol90} g | a a g2 |)\vol(50~ 70)
```

Chapter 2

The `playmus` Program

The `playmus` program has the following argument format:

```
FROM,START/N,END/N,TADJ/K/N,TO/K,UPB/K/N,
PP/S,LEX/S,TREE/S,PTREE/S,STRACE/S,NTRACE/S,MTRACE/S,
MIDI/K,PLAY/S,ACC/S,PITCH/N
```

These arguments have the following effects.

FROM	The filename of the MUS source file.
START/N	The number of the first bar to play.
END/N	The number of the last bar to play.
TADJ/K/N	Adjust the initial playing speed.
TO/K	The output filename.
UPB/K/N	The Upper bound of the work space used by the macrogenerator.
PP/S	Output the result of macro generation.
LEX/S	Trace the lexical tokens.
TREE/S	Output the parse tree just after syntax analysis.
PTREE/S	Output the parse tree after the tree has been translated into note events.
STRACE/S	Trace the creation of parse tree nodes.
NTRACE/S	Trace the generation of Midi event as they are generated.
MTRACE/S	Trace Midi commands while playing.
MIDI/K	The filename to contain the Midi translation.
PLAY/S	Cause the composition to be played on a Midi device.
ACC/S	Cause the composition to be played on a Midi device synchronised with the soloist using a microphone and keyboard input.
PITCH/N	Cause all sound to be transposed up or down by the specified number of semitones.

If the given source file name does not contain a dot, the extension `.mus` is appended. The same transformation is also applied to files accessed by the `get` macro. `START` and `END` give the first and last bar numbers of the composition to be played. `TADJ` gives the percentage adjustment of the initial playing speed. If `ACC` is specified, the playing speed will be further adjusted during the performance.

`PP`, `LEX`, `TREE`, `PTREE`, `STRACE`, `NTRACE` and `MTRACE` are primarily debugging aids and cause the generation of various kinds of trace output either to the screen or to the `TO` file if specified. `PP` causes the result of macro expansion to be output. This output includes line number items of the form *<fno/ ln>* giving the file number and line number of the next character whenever this changes. `LEX` outputs the sequence of lexical tokens and `TREE` outputs the parse tree just after syntax analysis. `PTREE` outputs the tree after it has been used to generate note events. This allows the collected shape data and other modifications to the tree to be inspected. `STRACE` traces the creation and modification of parse tree nodes. `NTRACE` outputs the note events as they are generated and `MTRACE` outputs them are they are being played.

If the `MIDI` argument is given, `playmus` generates a Midi file with the given name. `PLAY` causes `playmus` to play the composition directly to a Midi device. `ACC` causes `playmus` to act as an accompanist using microphone input and input from the keyboard to synchronise the performance with the soloist. It will tactfully play soloist parts when they are not playing or seem to be lost.

`PITCH` will transpose every note sounded up or down by the specified number of semitones.

2.1   The macrogenerator

The macrogenerator used by `playmus` is called BGPM and is a re-implementation of GPM[3] designed Christopher Strachey in 1964 to help with the portable implementation of CPL. Unlike GPM, BGPM uses two stacks. The lefthand stack holds incomplete macro calls while they are being built up. The variable `bg_f` points to the start of the current incomplete call and `bg_h` points to the start of the latest argument being read in. When the argument separator character (`!`) is encountered the length of the latest argument is filled in and `bg_h` positioned to the next available position in the lefthand stack.

When the macro call character (`;`) is encountered, the final argument is completed and the complete call is moved from the lefthand stack to the righthand one. The variable `bg_p` is set to point to this complete call after saving certain variables to enable the previous state to be reinstated when this macro has finished expansion. The zeroth argument of the complete call is looked up in an environment chain pointed to by `bg_e` and scanning continues from start of its body. If the macro being called is `def` then the complete call is converted in situ

into environment item and inserted at the start of the environment chain pointed to by `bg_e`.

The function `bggetch` returns the next character to be processed either from memory or from an input file. As a side effect it ensures that the global variable `lineno` holds the packed file number and line number of the character it return. The least significant 20 bits of `lineno` hold the line number and the other bits hold the file number. File names are held in the vector `sourcenamev`. If `fno` is a file number then `sourcenamev!fno` is its name. If `bg_c` is zero `bggetch` reads characters from file via the currently selected input stream, but if `bg_c` is non zero it points to a position in the body of the current macro being expanded. Arguments held in memory and the bodies of defined macros contain a mixture of characters and line number items allowing `lineno` to be set correctly whenever `bggetch` returns a character.

In `playmus` the macrogenerator is implemented as a BCPL coroutine. It reads characters from file using the standard `rdch` calls, but passes the expanded text to the lexical analyser by calls of `cowait(ch)` in the function `bgputch`. The lexical analyser thus obtains the expanded text by successive calls of `callco(bgpmco)`, and for each character its receives the variable `lineno` correctly set to identify the character's file and line number. Characters generated by some build in macros such as `$lquote;` orginate from a source and so are given the file and line numbers of the semicolon (;) of the macro calls.

More details of the BGPM implementation can be found by reading its source code `com/playmus.b` in the standard BCPL distribution.

2.2   The lexical analyser

The lexical analyser is implemented by the parameterless function `lex` which reads characters supplied by the macrogenerator and sets the global `token` to the code of the next lexical token every time it is called. It also sets the global variable `tokln` to the file/line number of the first character of the token. For some tokens other globals (`numval`, `noteletter`, `notesharps`, `reloctave`, `notelengthnum`, `dotcount` and `stringval`) hold additional information. The character reading is performed by `rch` using the assignment `ch := callco(bgpmco)`. The characters and their file/line numbers are placed in the circular buffers `chbuf` and `chbufln` to enable the context of an error to be output by the error message functions.

On entry to `lex`, `ch` hold the first character of the next token. White space is ignored. Numbers consist of a sequence of decimal digits possibly containing a decimal point and optionally preceeded by a minus sign. The resulting token is `s_numtied` or `s_num` depeding on whether the number was terminated by a ties sign ~ or not. The number's value is placed in `numval` as a scaled value with three decimal digits after the decimal point. So `15` would set `numval` to `15000`, and `-1.234` would set it to `-1234`.

The tokens `s_note` and `s_notetied` start with the letters `a` to `g` optionally followed by `is`, `isis`, `es` or `eses` denoting sharps and flats. Then follows an optional sequence of commas (`,`) and quotes (`'`) to adjust the octave and possibly a note length number (0, 1, 2, 4, 8,...128) and some dots. A tied note is terminated by a tie sign (`~`). The globals `noteletter`, `notesharps`, `reloctave`, `notelength` and `dotcounts` are set appropriately to allow the syntax analyser to calculate the pitch of the note.

All reserved words such as `\vol` and `\timesig` are introduced by `\`. They are looked up in a preset hash table using the function `lookupword` and if not found there an `Unknown keyword` error message is generated.

In addition to the macrogenerator comments introduced by `%` there are comment consisting of `//` up to the end of the line, and nested comments enclosed in `/*` and `*/` brackets.

Strings are enclosed in double quotes and are packed into newly allocated space pointed to by `stringval`.

Rests and spaces are introduced by `r` or `s` optionally followed by a length number and dots. The remaining tokens are `s_null` (`z`), `s_lsquare` (`[`), `s_rsquare` (`]`), `s_lparen` (`(`), `s_rparen` (`)`), `s_lcurly` (`{`), `s_rcurly` (`}`), `s_colon` (`:`), `s_star` (`*`), `s_startied` (`*~`), `s_barline` (`|`) and `s_doublebar` (`||`) are easily recognised.

2.3   The syntax analyser

The syntax analyser parses the sequence of lexical token supplied by the lexical analyser to for a tree of nodes. The first three field of each node are a link field, a node operator and a file/line number value. The link field is either zero or points to the next node in a sequence or item in a parallel construct. The operator (eg `s_note`, `s_rest` or `s_barline`) is a lexical token indicates the kind of node and the file/line number field holds the packed file and line number of that token. The size of nodes and the purpose of their other fields depends on the operator as described in the next section.

2.3.1   The parse tree nodes

The complete set of possible parse tree nodes are listed below, with `-` and `ln` representing the link field and file/line number fields.

```
[-, Note, ln, <letter,sharps,n>, qlen]
[-, Note, ln, <letter,sharps,n>, qlen]
```

These represent notes and tied notes with `<letter,sharps,n>` holding three packed bytes giving the note letter, the number of sharps or flats and the untransposed Midi note number of the note. The nominal length of the note is held in `qlen`.

```
[-, Rest,  ln, qlen]
[-, Space, ln, qlen]
```
These represent rests and spaces with the nominal qbeat length held in `qlen`.

```
[-, Seq, ln, note-list, qlen]
```
This represents a sequence of note items pointed to by `note-list` and chained together using the link fields. The total qbeat length of the sequence will be placed in `qlen`.

```
[-, Par, ln, note-list, qlen]
```
This represents the list of note items of a Par construct. They are pointed to by `note-list` and chained together using the link fields. The qbeat length of the longest item will be placed in `qlen`.

```
[-, Tuplet, ln, noteitem, noteitem]
```
This represents a Tuplet construct with the two note items representing the left and right operands repectively.

```
[-, Block, ln, body, envblk, shapeitems, qstart, qend]
```
This represents a Block node either explicitly created using of curly brackets (`{` and `}`) in the source text or created automatically by the system where found to be necessary. The `body` is the enclosed note item, `envblk` will be set to zero or the enclosing environment block, `shapeitems` is the list of shape items declared within this block.

The final two fields `qstart` and `qend` will hold the absolute (scaled) position of the first and last quantum beat of the block.

```
[-, Delay,      ln, noteitem, shapelist]
[-, Delayadj,   ln, noteitem, shapelist]
[-, Legato,     ln, noteitem, shapelist]
[-, Legatoadj,  ln, noteitem, shapelist]
[-, Tempo,      ln, noteitem, shapelist]
[-, Tempoadj,   ln, noteitem, shapelist]
[-, Vibrate,    ln, noteitem, shapelist]
[-, Vibrateadj, ln, noteitem, shapelist]
[-, Vibamp,     ln, noteitem, shapelist]
[-, Vibampadj,  ln, noteitem, shapelist]
[-, Vol,        ln, noteitem, shapelist]
[-, Voladj,     ln, noteitem, shapelist]
```
These represent applications of the shape operators. The note item is the left hand operand and `shapelist` is a List node containing a list of shape values. Shape values are Num, Numtied, Space, Star and Startied nodes.

```
[-, Num,     ln, value]
[-, Numtied, ln, value]
[-, Space,   ln, qlen]
```

```
[-, Star,       ln]
[-, Startied, ln]
```
These are the shape value nodes used in shape lists.

```
[-, Volmap, ln, shapelist]
```
This represents the Volmap comstruct with `shapelist` having the same structure as in the other shape nodes.

```
[-, Barline,   ln]
[-, Doublebar, ln]
```
These represent bar lines.

```
[-, Score,      ln, name, conductor, parts]
[-, Conductor, ln, noteitem]
[-, Part,       ln, noteitem, channel]
[-, Solo,       ln, noteitem, channel]
[-, Interval,  ln, msecs]
```
The complete parse tree is Score or Interval node or a List node containing a sequence of Score and Interval nodes chained together using the link fields. A Score node has a Conductor part and a collection of non solo or solo parts. Multiple parts are combined using a Par node. Part and Solo nodes have channel fields holding values in the range 0 to 15 representing Midi channels in the range 1 to 16.

```
[-, Transposition, ln, semitonesup]
```
This represents a transposition statement with semitonesup being an integer specifying how many semitones to raise the notes by from now on. Each part has no transposition until a transposition statement is encountered.

```
[-, Pedon,     ln]
[-, Pedoff,    ln]
[-, Pedoffon, ln]
[-, Portaon,  ln]
[-, Portaoff, ln]
[-, Softon,    ln]
[-, Softof,    ln]
```
Pedon, Pedoff, Pedonoff, Softon and Softoff simulate the effect of a piano's sustain and soft pedals. Portaon and Portaoff turn on and off the portamento effect. When on the transition between two consecutive notes slides through all the intermediate frequencies between them.

```
[-, Patch,    ln, patchnum]
[-, Bank,     ln, msb, lsb]
[-, Control, ln, controller, val]
```
These provide low level control of a Midi device.

```
[-, Varvol, ln]
[-, Nonvarvol, ln]
```

A Varvol node specifies that from now on notes can change their volume while being played as is typical of wind or string instruments. A Nonvarvol node specified that the volume of notes are entirely specified by how hard they are struck as is typical of keyboard instruments.

```
[-, Timesig, ln, value, value]
[-, Keysig,  ln, <letter,sharps,n>, mode]
```
Timesig nodes describe the time signature and are used in the conductor's to build up a mapping between beat numbers and the position in the composition.

```
[-, Title,              ln, string]
[-, Composer,           ln, string]
[-, Arranger,           ln, string]
[-, Opus,               ln, string]
[-, Instrument,         ln, string]
[-, Instrumentname,     ln, string]
[-, Instrumentshortname, ln, string]
[-, Barlabel,           ln, string]
[-, Partlabel,          ln, string]
```
Some of these are used in error messages but are otherwise ignored.

```
[-, List, ln, itemlist]
```
This construct is used to combine multiple scores and also shape values in shape lists.

2.3.2 The parser algorithm

The parser uses a simple recursive descent mechanism to parse the `.mus` text. It call `lex()` every time it needs another lexical token from the lexical analyser. This update the variables `token`, `tokln` and possibly others variables associated with the latest token. It builds parse tree nodes using calls such as `mk4(0, s_rest, ln, qlen)`. Errors detected during syntax analysis are typically genrated by calls such as: `synerr("Bad note length %n", lengthnum)` which might generate the following:

```
Error near t4.mus[40]: Bad note length 5
...
<4/143>1
<4/139> 1)
<4/140>  \patch 0<2/35><2/36>
<2/37>    3c4\l100 d f | //1
<2/38>
<2/39>a4 c d |//3
<2/40>    a5

Files: <1>=built-in <2>=t4.mus <3>=mushdr.mus <4>=gshdr.mus
```

This specified that the error was detected near line 40 of file `t4.mus`. The context of the error is shown by outputing recent characters supplied by the macrogenerator. When the file or line number changes an item such as `<2/40>` is inserted showing that the following characters came from line 40 of file number 2 namely `t4.mus`. In this case the error was that `a5` had a bad note length number. It should probably have been `a4`.

The main functions os the syntax analyser are briefly described as follows.

formtree()

This is the main function of the syntax analyser. It first initialises the table

of reserved words (for symbols such as `\part` or `\tup`) before calling `rdscores` to parse the text. However, if the `lex` option were specified it tests the lexical analyser by reading and outputing the sequence of lexical tokens.

rdscores()

This parses a sequence of scores and intervals forming a Score, Interval or List

node as appropriate.

rdscore()

This parses a single score or interval construct. If a score is found, it reads

the conductor part and instrumental parts and creates a node of the form `[-,` `Score, ln, conductor, parts]` where `conductor` is a Conductor node, and `parts` is a Part or Solo node or a Par node containing a collection of such nodes.

rdnoteprim()

This reads a basic note item up to but not including a dyadic operator such

as `\vol` or `\tuplet`. Basic note items include notes, rests, clefs, time and key signatures and many others including sequences enclosed in parentheses, parallel constructs enclosed in square brackets and blocks enclosed in curly brackets.

rdnoteitem()

This reads in a basic note item and combines it with any dyadic constructs

that follow. The result is either the original note item, or a nested collection of shape (eg Vol or Tempo) and Tuplet node.

rdnoteitems()

This reads a sequence of note items chaining them together using their link

fields. It is only called when parsing a construct enclosed in parentheses, square brackets or curly brackets.

rdshapelist()

This is only called to parse the shape lists following such operators as `\vol`,

`\legato` or `\volmap`. A shape list is either a number, a tied number, a star, a tied star or a space, or it can be a sequence of these items enclosed in parentheses.

Colon items such as `:256` or `:s8` may be embedded within shape lists to change the scaling factor from the default value of 1024. This scaling factor only applies to subsequent numbers and tied numbers within the current shape list.

```
insertblocks(noteitem, envblk)
blockneeded(noteitem)
initblock(noteitem, envblk)
shapescan(noteitem, envblk)
insertblocks(noteitem, envblk)
```

This function performs a depth first search over the tree rooted at `noteitem` inserting Block nodes wherever necessary and filling in the `qlen` fields of all Seq and Par nodes encountered. It does not explore the note item's link field. On entry, the global variable `qbeats` holds the local qbeat position of the note item, and `scbase`, `scfaca` and `scfacb` are the scaling parameters used by `qscale` (described below) to convert local qbeat positions to absolute values. The current environment block is supplied in the argument `envblk`.

It ensures that the body of the conductor node is a block, and that the bodies of Part and Solo nodes are blocks if they contain unprotected shape operators. Similarly, the elements of Par constructs and the left hand operands of Tuplet nodes are blocks, if needed.

The function blockneeded(noteitem) returns `TRUE` if `noteitem` contains unprotected shape data and this is used by `insertblocks` to determine whether a block is really necessary. For each block it finds or inserts it fills in the `parent`, `qstart` and `qend` fields, and initialised the shape item list by calling `initblock` which in turn calls `shapescanblock` for each kind of shape.

`shapescanblock` searches for shape data belonging to its block. It searches everywhere except for the elements of a Par constructs, the bodies of an inner blocks or the lefthand operands of a Tuplet constructs. It does this by calling shapescan to do most of the work. But before returning, `shapescanblock` inserts a final shape data item if previous item was not positioned at the end of the block.

Having initialised the block, `insertblocks` continues its depth first search by calling calling itself recursively on the block's body.

```
initshapeitems(part, envblk)
shapescanblock(noteitem, envblk)
```

This is called after `insertblocks` has inserted and partially initialised all Block nodes. This function performs a second depth first search over its give tree initialising the shape item lists of every Block node it finds by calling `shapescanblock` for each kind of shape to populate the block's list of shape items. It then continues searching the body of the block. By processing the blocks in this order, the values of star items in shape lists can be looked up correctly.

```
qscale(q)
```
This...

2.4   The Midi generator

2.4.1   Self expanding vectors

2.5   `writemidi`

2.6   `playmidi`

2.6.1   Keyboard input

2.6.2   Microphone input

2.6.3   Synchronisation

Bibliography

[1] Lilypond. *Lilypond: music notation for everyone.* www.lilypond.org.

[2] M. Richards. *My WWW Home Page.* www.cl.cam.ac.uk/users/mr/.

[3] Christopher Strachey. A General Purpose Macrogenerator. *Computer Journal*, 8(3):225–241, 1965.