# Computer-Aided Etude Composition in Haskell

Matt Munz
mmunz@pantheon.yale.edu

December 21, 2005

**Abstract**

The performer is perhaps the most significant mediator of the communication between a composer and her audience. Accordingly, the wide variation among performers in musicianship and aesthetic preference may produce widely varying interpretations of the same score. This research investigates the use of Haskore, [Hud00] a highly abstract music programming language, to develop models of performers to inform computer-aided composition. To demonstrate this approach, this paper describes a program for the computer-aided composition of saxophone etudes that was developed by the author.

## 1 Introduction

This paper describes a novel system for the computer-aided generation of etudes for the saxophone. At the core of the system is a model of a performer – a database of information about a given musician that is populated as he uses the system. Rudimentary etudes, tailored to meet the musician's demonstrated abilities, are then derived from this database. While perhaps the basis for a practical pedagogical tool, this program is primarily intended to serve as a proof-of-concept of the efficacy of performer models in computer-aided composition.

### 1.1 Motivation

The ostensible purpose of a musical composition is to induce some cognitive effect in the listener, and a score is, in one sense, an abstract description of the means, acoustically, to achieve that effect. Mediating this communication between the composer and her audience, however, is, perhaps most significantly, the performer, whose interpretation of the score depends on his

level of musicianship, aesthetic preferences, physical attributes, etc. While many score editors and other composition programs excel at typeography, sound synthesis, and similar tasks, they often leave issues of asthetics (including musical structure) or performance concerns entirely in the hands of the composer.

This is perhaps best illustrated by an example. Consider the difference between the mechanics of transposition and the art of transcription. Transposing a melody from a piano piece into the range of the trombone is a rather simple matter that most composition tools automate adequately. A proper transcription of that piece for the trombone, on the other hand, might require modifications to the rhythm and contour of the melody to accommodate the pecularities of typical trombone technique. Furthermore, one might want also to consider the level of musicianship of the trombonist who will perform the transcription. An expert might be able to play the piano part without modification, however a novice might require a simplified treatement of the melody. The author is unaware of any existing system that attempts to address these common problems in composition.

Offloading these tasks onto the computer may free the composer to focus more on the intent of his composition, and less on the mechanics of its performance, thus producing more effective, expressive compositions faster, with a greater degree of confidence in their success. Additionally, these techniques may be of particular use in conjunction with existing methods of algorithmic composition. Many techniques for the computer-generation of music can produce humanly unplayable melodies. A database of information about a performer's abilities, as described here, could be used to reliably arrange such a score such that it could be performed by a human, without sacrificing the work's innate musical qualities.

## 1.2 Related Research

One approach to creating music composition software interfaces with a high level of abstraction is to develop high-level music programming languages. Haskore [Hud00] is a high-level music programming language that is embedded in the functional programming language Haskell. [Jon03] This embedding gives Haskore programmers all of the power and expressiveness of Haskell, while presenting a simple, intuitive interface for music programming. The use of Haskell in audio processing [Thi04] and multimedia programming language design, [Hud03] demonstrates that a highly abstract theoretical model may be encoded cleanly and elegantly as a Haskell program. This research motivated the use of Haskore for this project.

## 2 Architecture

The system for computer-aided etude composition described in this paper is divided into several modules which can be arranged into a workflow that might reasonably become part of a musician's practice regimen. This workflow has an iterative aspect that causes the etude generation process to become more effective as it is used.

The `Scores` module extends Haskore with a data type for scores and a music typesetting function for printing those scores. A musician or music teacher may begin thier use of the system by first using this module to encode and then print a score. The musician may then perform this music, while making note of the phrases in the piece where difficulties arise. Then the musician or music teacher could use the `PerformanceEvaluation` module to annotate the score, on a per-phrase basis, with information about the musician's performance of those phrases. The `Performer` module would then be used to load a database of information about the musician from the disk, and subsequently the `PerformanceEvaluation` module would be used to update this database with information derived from the performance evaluation described in the previous step.

Finally, a new etude may be generated using the `EtudeGeneration` module. This etude would be based on the pitch and rhythmic content of the score, and would be comprised of a series of simple alternating two- and three-note melodies, designed to exercise the weaknesses in technique that were revealed by the performance evaluation mentionned previously. Of course, this etude would itself be a score, and as such it could be printed, performed, and that performance evaluated, to become the basis of even more etudes. As the musician continues to use the system, the database of information about his abilities will become more accurate, allowing the etude generator to become even more specialized to his abilities.

This system itself is specialized for saxophonists, although it could be extended with a little effort to accommodate most woodwinds, and with a little more effort all monophonic instruments. The performer database is composed of measures of proficiency at transitions between the various fingerings of the saxophone. For a given performer it is a descripition of the absolute ease or difficulty with which she may execute any transition from one note to the next. More accurately, it encodes the ease at which the musician may execute a transition from one *fingering* to another, since for some pitches there may be multiple fingerings on the saxophone. Accordingly, both the `PerformanceEvaluation` and `EtudeGeneration` modules make use of the `SaxophoneFingerings` module to convert melodies to and from sequences

of saxophone fingerings.

Finally, a few other modules provide additional datatypes and utility functions that are necessary to implement, test, and demonstrate the functionality already described.

# 3 Typesetting

While Haskore provides a very abstract and flexible representation of music, it is necessary to extend this representation somewhat to define a minimal representation of musical notation sufficient for the etude generation task described in this paper. The `Scores` module contains this Haskore extension (a Haskell data type), as well as a mechanism for typesetting scores encoded with that data type.

```
> module Scores where
> import Basics
> import Ratio
> import List
> import ComposersWorkbench
> import Miscellany
```

## 3.1 Score Representation

To simplify the development of analysis and typesetting algorithms, a restricted musical vocabulary is used in `Score`. In this simple model, a `Score` has only one time signature, tempo marking, and key throughout. Also, a `Score` must be monophonic, and not contain arbitrary tuples, although triplets are permitted. `Score` is essentially an extension of the Haskore data type `Music`, with notation-specific features like keys and time signatures.

```
> data Score = Score {title :: Title,
>                     timeSignature ::  TimeSignature,
>                     beatsPerMinute :: Integer,
>                     key :: Key,
>                     music ::  Music}
>     deriving (Show, Eq)
```

```
> data Key =  Key PitchClass Mode
>     deriving (Show, Eq)

> data Mode =  Major | Minor
>     deriving (Show, Eq)

> type Title = String
> type TimeSignature = Ratio Int

> -- timeSigToBPMeas converts a time signature to
> --   a number of beats per measure
> timeSigToBPMeas :: TimeSignature -> Integer
> timeSigToBPMeas = fromInt.numerator

> scoreWithTitle :: Title -> [Score] -> Score
> scoreWithTitle t s = (filter (\(Score t' _ _ _ _) -> t' == t) s) !! 0

> -- phraseNum finds the nth phrase in a score
> phraseNum :: Integer -> Score -> Music
> phraseNum n (Score _ _ _ _ m) = (phrases m []) !! (fromInteger n)
>   where phrases :: Music -> [Music] -> [Music]
>         phrases m' acc = case m' of
>           (Phrase _ _)  -> acc ++ [m']
>           (m1 :+: m2)   -> phrases m2 (phrases m1 acc)
>           (_ :=: _)     -> error "polyphony not yet supported"
>           (Tempo _ m0)  -> phrases m0 acc
>           (Trans _ m0)  -> phrases m0 acc
>           (Instr _ m0)  -> phrases m0 acc
>           (Player _ m0) -> phrases m0 acc
>           _             -> acc
```

### 3.2 LilyPond Scores

The back-end of the typesetting subsystem is LilyPond, [HWN05] an "automated engraving system" that is available on many platforms. LilyPond has a command-line interface for compiling scores encoded in the LilyPond input format into several printable formats, including PostScript.

The interface between the system described here and LilyPond is through the LilyPond input format. This format is actually a rather substantial

domain-specific language, implemented in Scheme, and only a subset of its functionality is required for this project. This functionality is modelled internally with the `LilyPondScore` data type.

```
> type LilyPondScore = [LilyPondSection]
```

In this (simplified) model, a `LilyPondScore` is a list of sections, each of which may be a comment, header, version declaration, a "relative octave" mark, or a block of more primitive LilyPond expressions. The header is primarily used to express the title of the piece, the version declaration specifies the version of LilyPond to which the input file conforms, and the "relative octave" mark tells the LilyPond interpreter to use the same octave for several consecutive notes.

```
> data LilyPondSection = LPComment String
>                      | LPHeader String
>                      | LPVersion String
>                      | Relative
>                      | LPBlock [LilyPondMark]
>     deriving (Show, Eq)
```

More importantly, the typographic elements for notes, barlines, ties, keys, and tempo markings are represented by the `LilyPondMark` data type. Each of these may appear anywhere within a given block of LilyPond expressions.

```
> data LilyPondMark = LPNote LPPitchClass LPDuration
>                   | Tie
>                   | LPKey Key
>                   | MetronomeMark {beatLength :: LPDuration,
>                                    beatsPerMinute' ::  Integer}
>                   | FinalBar
>     deriving (Show, Eq)
```

Pitches and are represented in the LilyPond input format by pitch class names modified with quote marks, the number of which denote the octave of the pitch. Flats and sharps are represented by adding the suffixes "es" and "is", respectively, to the pitch class name. Note durations are limited to the set of particular notehead glyphs that are available (for everything else, ties must be used). 1 denotes whole notes, 2 denotes half notes, 4 denotes quarter notes, etc. One example of a note in the LilyPond input format is bes"16, which is a B-flat 16th note in the second octave of whichever clef is being used.

```
> type LPPitchClass = String
> type LPDuration = String
```

### 3.3 Transforming Scores into LilyPond Scores

Typesetting a `Score` involves several steps. First, the `Score` to be rendered is transformed into an internal representation of a sufficient subset of the LilyPond notation. Then, this data structure is written to a file, using the LilyPond input format syntax. Finally, the LilyPond shell command is invoked programmatically to produce a Portable Document Format (PDF) file, suitable to print on almost any operating system.

The `score2LPScore` function converts `Score`s to `LilyPondScore`s. Primarily, this is achieved by converting Haskore music values into sequences of `LilyPondMarks`. As mentionned previously some Haskore constructs like polyphony are omitted for simplicity in implementation, although it is possible to add this functionality later.

```
> -- In the following, the quarternote beat is assumed
> --   for now, but this eventually, should be derived from
> --   the time signature
> score2LPScore :: Score -> LilyPondScore
> score2LPScore (Score t _ bpm k m)
>   = [(LPComment "Generated from a Score by Scores.lhs."),
>      (LPHeader t),
>      LPBlock ((LPKey k):(MetronomeMark "4" bpm):b),
>      (LPVersion "2.6.0")]
>   where b = ((music2NoteSeq m 0 4 1) ++ [FinalBar])
```

The conversion of `Score`s to `LilyPondScore`s is non-trivial, since Haskore note durations (in the interpretation used here) are measured in measures and may be of arbitrary length, but LilyPond note durations are limited to the durations of noteheads. In LilyPond, all other durations must be composed of multiple notes with ties. Haskore's Music data type, on the other hand, does not have a notion of ties at all.

While it is relatively straightforward to convert a duration like Haskore's `3%4` into three quarter notes tied together, this is, of course, contrary to conventional music typesetting practice. In this case, `3%4` should be written as a dotted-half note, if it occurs on the first or second beat of the measure in 4/4 time. Matters become even more complicated when a note does not begin or end on a beat. For example, a `1%8` rest followed by a `1%4` note should be written as an eighth-note rest followed by two tied eighth-notes.

The process for converting durations is as follows. First the duration is converted into a number of ticks, where one tick is one 32nd of a measure in length, and rounding is used as necessary.

```
> durToTicks :: (Integral a) => Ratio a -> Integer
> durToTicks d
>  =  if (isWholeNum ts)
>     then iRatioFloor ts
>     else error ("@todo cannot handle dur: " ++ (show d))
>   where ts :: Rational
>         ts = ((numTicksPerMeasure % 1) * (toRational d))

> numTicksPerMeasure :: Integer
> numTicksPerMeasure = 32
```

Then, given the current position in the current measure, the duration (in ticks) is divided up into a set of durations, such that none of the new durations crosses a beat. A recursive function, `makeLPRests`, is used to create dotted notes and larger note values in some cases. (Future versions of the system will include a revised version of this function which will work in all cases)

```
durToLPRests: converts a duration to a string of LP Rests
dur: duration, in ticks
pos: amount of the measure already used up, in ticks
```

```
tpb: ticksPerBeat

> durToLPRests :: Integer -> Integer -> Integer -> [LilyPondMark]
> durToLPRests 0   _   _   = []
> durToLPRests dur pos tpb
>   = (makeLPRests newDur numTicksPerMeasure) ++
>     (durToLPRests (dur - newDur) (pos + newDur) tpb)
>  where nextBeatPos = ((pos `div` tpb) + 1) * tpb
>        endPos      = pos + dur
>        newDur      = if endPos > nextBeatPos
>                      then nextBeatPos - pos
>                      else dur

makeLPRests: creates notes with dotting where possible
d: a duration in ticks
b: a base number of ticks

> makeLPRests :: Integer -> Integer -> [LilyPondMark]
> makeLPRests 0 _ = []
> makeLPRests d b
>  = if ((rem numTicksPerMeasure d) == 0)
>    then [lpRest (numTicksPerMeasure `div` d)]
>    else if d < b
>         then makeLPRests d halfB
>         else note : (makeLPRests newR b)
>  where r      = d - b
>        useDot = r >= halfB
>        note   = if useDot
>                 then lpRest' ((show (numTicksPerMeasure `div` b))
>                               ++ ".")
>                 else lpRest (numTicksPerMeasure `div` b)
>        newR   = if useDot then (d - (b + halfB)) else r
>        halfB  = if odd b
>                 then error ("Expected b to be even: " ++ (show b))
>                 else b `div` 2
```

Finally, the sets of durations for each note or rest are converted into note or rest marks. For notes, each note in the group is given a pitch class mark and it is tied to its neighbors in the group.

```
> -- measurePos is the amount of the measure already used up
> -- bpMeas: Beats Per measure
> -- ml: measure length
> music2NoteSeq :: Music -> Rational -> Integer -> Dur -> [LilyPondMark]
> music2NoteSeq m measurePos bpMeas ml
>  = case m of
>    (Note p d _)  -> intersperse Tie (map (f p) (dRests d))
>    (Rest d)      -> dRests d
>    (m1 :+: m2)   -> (music2NoteSeq m1 measurePos bpMeas ml)
>                     ++ (music2NoteSeq m2 (measurePos +
>                                           (durToRational (dur m1)))
>                                         bpMeas ml)
>    (_ :=: _)     -> error "polyphony not yet supported"
>    (Tempo _ _)   -> error "tempo not yet supported"
>    (Trans i m')  -> music2NoteSeq
>                         (transposeM i m') measurePos bpMeas ml
>    (Instr _ _)   -> error "Instr constructor not yet supported"
>    (Player _ _)  -> error "Player constructor not yet supported"
>    (Phrase _ m') -> music2NoteSeq m' measurePos bpMeas ml
>   where dRests d = durToLPRests
>                       (durToTicks d) (durToTicks measurePos) tpb
>         tpb = if (rem numTicksPerMeasure bpMeas) == 0
>               then (numTicksPerMeasure `div` bpMeas)
>               else error ("num ticks per measure: " ++
>                           (show numTicksPerMeasure) ++
>                           " is incompatible with " ++
>                           "beats per measure: " ++
>                           (show bpMeas))
>         f p (LPNote _ d') = LPNote (lpPitchString p) d'

> lpRest d' = lpRest' (show d')

> lpRest' d = LPNote "r" d

> lpPitchString :: Pitch -> String
> lpPitchString (pc, o) = (toLPPitchClass pc) ++ octaveString
>   where octaveString = (replicate (o - 3) '\'') ++ (replicate (3 - o) ',')

> toLPPitchClass :: PitchClass -> LPPitchClass
```

```
> toLPPitchClass pc'
>   = case pc' of
>       Cf -> "ces";  C -> "c"; Cs -> "cis"
>       Df -> "des";  D -> "d"; Ds -> "dis"
>       Ef -> "ees";  E -> "e"; Es -> "eis"
>       Ff -> "fes";  F -> "f"; Fs -> "fis"
>       Gf -> "ges";  G -> "g"; Gs -> "gis"
>       Af -> "aes";  A -> "a"; As -> "ais"
>       Bf -> "bes";  B -> "b"; Bs -> "bis"

> modeString :: Mode -> String
> modeString m = case m of
>   Major -> "major"
>   Minor -> "minor"
```

### 3.4 Writing a LilyPond Score to a File

Writing a `LilyPondScore` is a simple matter of mapping `LilyPondMark`s and `LilyPondSection`s to their string representations, and concatenating all of those strings together. The result is a valid LilyPond input script.

```
> renderLPScore :: LilyPondScore -> String
> renderLPScore sections = concat (map rs sections)
>  where rs s = (renderSection s) ++ "\n"
>        renderSection (LPComment c) = lpComment c
>        renderSection (LPHeader t)  = lpHeader t
>        renderSection (LPVersion v) = lpVersion v
>        renderSection Relative      = "\\relative"
>        renderSection (LPBlock marks)
>          = "{" ++
>             (concat (map (\x -> (renderLPMark x) ++ " ") marks))
>             ++ "}"

> renderLPMark :: LilyPondMark -> String
> renderLPMark m = case m of
>   (LPKey (Key pc mode))   -> "\\key " ++ (toLPPitchClass pc) ++ " "
>                              ++ "\\" ++ (modeString mode)
>   (MetronomeMark bl bpm) -> "\\tempo " ++ bl ++ "=" ++ (show bpm)
```

```
>   (LPNote p d)           -> p ++ d
>   Tie                    -> "~"
>   FinalBar               -> "\\bar \"|.\""

> lpComment text = "%{" ++ text ++ "%}"
> lpHeader title = lpSection "header" ("title = \"" ++ title ++ "\"")
> lpSection sectionName body = "\\" ++ sectionName ++ "{" ++ body ++ "}"
> lpVersion version = "\\version \"" ++ version ++ "\""
```

### 3.5 A Typsetting Example

The `ScoresExample` module includes a short score that demonstrates the typesetting subsystem. Data types from Haskore and the `ComposersWorkbench` module are used to encode the first three phrases from the Finale from Theme and variations, found on page 242 of Paul DeVille's Universal Method for the Saxophone. [DeV08]

```
> module ScoresExample where
> import Ratio
> import Basics
> import ComposersWorkbench
> import Scores

> umTandV :: Score
> umTandV = Score title (4%4) 108 (Key D Major) m
>   where title         = "Finale from Theme and variations, " ++
>                         "Universal Method, p.242"
>         m             = (mp phrase1) :+: (mp phrase1) :+: (mp phrase3)
>         phrase1       = makeMelody oct phrase1PCS phrase1Rhythm
>         phrase3       = makeMelody oct phrase3PCS phrase3Rhythm
>         phrase1PCS    = [0, 4, -1, 0, -5, 4, -1, 0, -5, 7, 5, 4, 2, 0,
>                          -1, -3, -5, -7, -8, -10, -12, -13, -10, -3, -5]
>         phrase1Rhythm = stn3 :&: stn :&: qnPStn :&: stn3 :&: qn :&:
>                         stn8 :&: stn8
>         phrase3PCS    = [-8, -12, -10, -8, -7, -5, -3, -2, 0, -2, -3,
>                          -2, 2, 0, -2] ++
>                         [-3, -4, -3, 1, 2, 0, -1, -2, -1, 3, 4, 2] ++
>                         [0, -1, 0, 4, 5, 4, 2, 4, 2, 1, 2, 6, 7, 5]
```

```
>         phrase3Rhythm = phrase31R :&: motive32R :&: motive32R :&:
>                         motive32R :&: stn8
>         phrase31R     = en :&: stn :&: stn :&: stn4 :&: stn8
>         motive32R     = den :&: stn :&: stn4
>         stn3          = stn :&: stn :&: stn
>         stn4          = stn3 :&: stn
>         stn8          = stn4 :&: stn4
>         stn           = NoteR (1 % 16)
>         qnPStn        = NoteR (5 % 16)
>         qn            = NoteR (1 % 4)
>         en            = NoteR (1 % 8)
>         den           = NoteR (3 % 16)
>         oct           = (12*5)+2
>         mp p          = (Phrase [] p)

> renderUMTandV = renderLPScore (score2LPScore umTandV)
```

The value of `renderUMTandV` is the following text, which LilyPond interprets as shown in Figure 1.

```
%{Generated from a Score by Scores.lhs.%}
\header{title = "Finale from Theme and variations, Universal Method, p.242"}
{\key d \major \tempo 4=108 d''16 fis''16 cis''16 d''16 a'4 ~ a'16 fis''16
cis''16 d''16 a'4 a''16 g''16 fis''16 e''16 d''16 cis''16 b'16 a'16 g'16
fis'16 e'16 d'16 cis'16 e'16 b'16 a'16 d''16 fis''16 cis''16 d''16 a'4 ~
a'16 fis''16 cis''16 d''16 a'4 a''16 g''16 fis''16 e''16 d''16 cis''16 b'16
a'16 g'16 fis'16 e'16 d'16 cis'16 e'16 b'16 a'16 fis'8 d'16 e'16 fis'16
g'16 a'16 b'16 c''16 d''16 c''16 b'16 c''16 e''16 d''16 c''16 b'8. ais'16
b'16 dis''16 e''16 d''16 cis''8. c''16 cis''16 f''16 fis''16 e''16 d''8.
cis''16 d''16 fis''16 g''16 fis''16 e''16 fis''16 e''16 dis''16 e''16
gis''16 a''16 g''16 \bar "|." }
\version "2.6.0"
```

**Finale from Theme and variations, Universal Method, p.242**

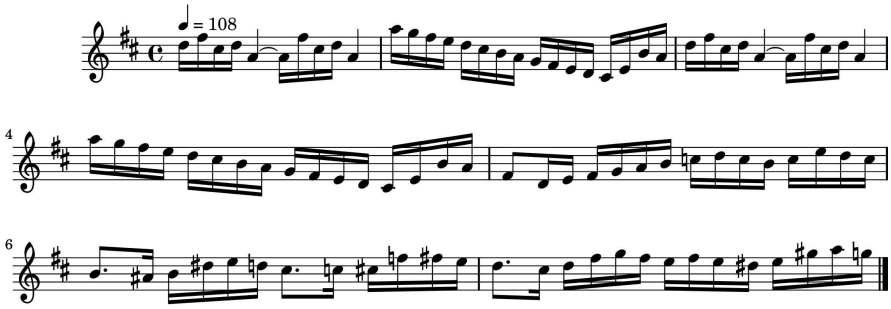


Figure 1: The final result of typesetting the example score.

# 4 Fingering Selection

A saxophone fingering is a set of keys that are depressed on the saxophone to produce a pitch. Each fingering produces only one pitch, but several fingerings may produce a common pitch. In common practice, fingering selection is left up to the performer, and it is expected that he will choose the best sequence of fingerings for any given phrase. For a performer some fingering sequences will be much more difficult than others, so the difficulty of a piece depends both on the sequences of fingerings that are selected and the performer's ability to execute those sequences at a given tempo.

The `SaxophoneFingerings` module serves as a means to generate a "correct" fingering sequence for a given musical phrase. The "correct" fingering sequence is here defined as the easiest sequence for an intermediate to advanced saxophonist to execute, according to the author's experience as a saxophonist. The design of the module is sufficiently general, however, that it could be modified to fit the tastes of other saxophonists, or even to be dynamically adjusted to match the abilities of a given perforer as their technique improves.

```
> module SaxophoneFingerings where
> import Ix
> import Ratio
> import Graph
> import Basics
> import ComposersWorkbench
```

### 4.1 Fingering Representation

The basis for this representation of saxophone fingerings is the key layout of the Selmer Mark VI Alto Saxophone. Each fingering is given a unique number to simplify identification and indexing. Fingerings are represented with this number, a list of keys that are simultaneously depressed, and the (concert) pitch that will be produced as a result.

```
> type SaxophoneFingeringNumber = Integer

> data SaxophoneFingering
>   = SaxFingering SaxophoneFingeringNumber
>                  [LeftHandKey] [RightHandKey] Pitch
>     deriving (Read, Show, Eq)

Abbreviations:

Symbol Meaning
------ -------
LH     Left Home
LP     Left Pinkey (used to play low notes)
LS     Left Side   (used to play high Eb, D, and F in the key of Eb)
T      Thumb
u      Upper

Examples:

Symbol  Used to play this pitch, in Eb
------  ------------------------------
LH1     B
LH1l    Bb
LP1     G#
LP2     low B
LP3     low C#
LP4     low Bb
LS1     high D
LS2     high Eb
```

```
LS3     high F

> data LeftHandKey = LH1u | LH1 | LH1l | LH2 | LH3
>                  | LP1 | LP2 | LP3 | LP4 | LS1 | LS2 | LS3 | T
>     deriving (Read, Show, Eq)

Abbreviations:

Symbol Meaning
------ -------
RH     Right Home
RP     Right Pinkey (used to play low notes)
RS     Right Side   (used to play middle Bb, C, etc.)
u      Upper

Examples:

Symbol  Used to play this pitch, in Eb
------  ------------------------------
RH1     F
RH3u    F#
RP1     D#
RP2     low C
RS1     high E
RS2     middle C
RS3     middle Bb

> data RightHandKey
>   = RH1 | RH2 | RH3 | RH3u | RP1 | RP2 | RS1 | RS2 | RS3
>     deriving (Read, Show, Eq)
```

A weighted graph of `SaxophoneFingering`s is used in the selection algorithm. Since the graph data type used requires its elements to be indexable and ordered, it is necessary to make `SaxophoneFingering` an instance of these classes.

```
> instance Ord SaxophoneFingering where
>   compare (SaxFingering n _ _ _) (SaxFingering m _ _ _) = compare n m
```

```
> instance Ix SaxophoneFingering where
>   range (SaxFingering n _ _ _, SaxFingering m _ _ _)
>     = map lookupFingering (range (n,m))
>   index (SaxFingering n _ _ _, SaxFingering m _ _ _)
>         (SaxFingering x _ _ _) = index (n,m) x
>   inRange (SaxFingering n _ _ _, SaxFingering m _ _ _)
>           (SaxFingering x _ _ _) = inRange (n,m) x
```

Notably, this representation is not sufficient to express extended techniques like half-hole fingerings and multiphonics. Likewise it is possible to use this representation to make nonsense fingerings. Nonetheless, it is a realtively simple and intuitive representation that is sufficient to express the most common fingerings on the saxophone. Additionally, while not all saxophones have the same key chart, the chart used here is one of the most common.

Given a suitable representation, the next step is to encode all of the saxophone fingerings in that representation. The variable names used indicate the register and pitch produced, in the key of Eb, by each fingering. For example the fingering which produces the concert pitch C one octave above middle C is given the name highAF. This corresponds to the name saxophonists commonly use for this fingering – "the fingering for high A".

```
> lowBbF   = sf 1  (lh123 ++ [LP4]) rhLowC           (Cs,3)
> lowBF    = sf 2  (lh123 ++ [LP2]) rhLowC           (D, 3)
> lowCF    = sf 3  lh123            rhLowC           (Ds,3)
> lowCsF   = sf 4  (lh123 ++ [LP3]) rhLowC           (E, 3)
> lowDF    = sf 5  lh123            rh123            (F, 3)
> lowDsF   = sf 6  lh123            (rh123 ++ [RP1]) (Fs,3)
> lowEF    = sf 7  lh123            rh12             (G, 3)
> lowFF    = sf 8  lh123            [RH1]            (Gs,3)
> -- low F# middle finger
> lowFs1F  = sf 9  lh123            [RH2]            (A, 3)
> -- low F# trill key
> lowFs2F  = sf 10 lh123            [RH1, RH3u]      (A, 3)
> midGF    = sf 11 lh123            []               (As,3)
> midGsF   = sf 12 (lh123 ++ [LP1]) []               (B, 3)
> midAF    = sf 13 lh12             []               (C, 4)
```

```
> -- middle A# sidekey
> midAs1F  = sf 14 lh12             [RS3]           (Cs,4)
> -- middle A# index finders
> midAs2F  = sf 15 [LH1]            [RH1]           (Cs,4)
> -- middle A# middle finger
> midAs3F  = sf 16 [LH1]            [RH2]           (Cs,4)
> -- middle A# two keys under left index
> midAs4F  = sf 17 [LH1, LH1l]      []              (Cs,4)
> midBF    = sf 18 [LH1]            []              (D, 4)
> -- middle C middle finger
> midC1F   = sf 19 [LH2]            []              (Ds,4)
> -- middle C side key
> midC2F   = sf 20 [LH1]            [RS2]           (Ds,4)
> -- middle C overtone
> midC3F   = addT 21 lowCF
> -- middle C# open
> midCs1F  = sf 22 []               []              (E, 4)
> -- middle C# overtone
> midCs2F  = addT 23 lowCsF
> midDF    = addT 24 lowDF
> midDsF   = addT 25 lowDsF
> midEF    = addT 26 lowEF
> midFF    = addT 27 lowFF
> midFs1F  = addT 28 lowFs1F
> midFs2F  = addT 29 lowFs2F
> highGF   = addT 30 midGF
> highGsF  = addT 31 midGsF
> highAF   = addT 32 midAF
> highAs1F = addT 33 midAs1F
> highAs2F = addT 34 midAs2F
> highAs3F = addT 35 midAs3F
> highAs4F = addT 36 midAs4F
> highBF   = addT 37 midBF
> highC1F  = addT 38 midC1F
> highC2F  = addT 39 midC2F
> highCsF  = addT 40 midCs1F
> highDF   = sf 41 [T, LS1]            []          (F, 5)
> highDsF  = sf 42 [T, LS1, LS2]       []          (Fs,5)
> highE1F  = sf 43 [T, LS1, LS2]       [RS1]       (G, 5)
> highE2F  = sf 44 [T, LH1u, LH2, LH3] []          (G, 5)
```

```
> highE3F  = sf 45 [T, LS3]            []          (G, 5)
> highE4F  = sf 46 [T, LS1, LH1u]      []          (G, 5)
> highF1F  = sf 47 [T, LS1, LS2, LS3]  [RS1]       (Gs,5)
> highF2F  = sf 48 [T, LH1u, LH2]      []          (Gs,5)

> sf = SaxFingering
> rhLowC = rh123 ++ [RP2] -- right hand of low C
> rh123  = rh12 ++ [RH3]  -- right hand home 123
> rh12   = [RH1, RH2]     -- right hand home 12
> lh123  = lh12 ++ [LH3]  -- left hand home 123
> lh12   = [LH1, LH2]     -- left hand home 12
> addT n' (SaxFingering n lh rh (pc,o))
>   = (SaxFingering n' (T:lh) rh (pc,o+1))
```

For this representation, it is straightforward to deine functions that map from pitches to fingerings and from fingerings to pitches.

```
> saxophoneFingerings :: Pitch -> [SaxophoneFingering]
> saxophoneFingerings p = saxF p'
>   where p' :: Pitch
>         p' = trans (-3) p -- alto sax Eb = concert c

> saxF :: Pitch -> [SaxophoneFingering]
> saxF p@(pc, o) = case (pc, o) of
>   -- enharmonics
>   (Bf, _) -> saxF (As, o);    (Cf, _) -> saxF (B, (o-1))
>   (Bs, _) -> saxF (C, (o+1)); (Df, _) -> saxF (Cs, o)
>   (Ef, _) -> saxF (Ds, o);    (Ff, _) -> saxF (E, o)
>   (Es, _) -> saxF (F, o);     (Gf, _) -> saxF (Fs, o)
>   (Af, _) -> saxF (Gs, o)
>   -- low range
>   (As, 2) -> [lowBbF]; (B,  2) -> [lowBF]; (C,  3) -> [lowCF]
>   (Cs, 3) -> [lowCsF]; (D,  3) -> [lowDF]; (Ds, 3) -> [lowDsF]
>   (E,  3) -> [lowEF];  (F,  3) -> [lowFF];
>   (Fs, 3) -> [lowFs1F, lowFs2F]
>   -- middle range
>   (G,  3) -> [midGF];  (Gs, 3) -> [midGsF]; (A,  3) -> [midAF]
>   (As, 3) -> [midAs1F, midAs2F, midAs3F, midAs4F]
```

```
>   (B,  3) -> [midBF];
>   (C,  4) -> [midC1F, midC2F, midC3F]
>   (Cs, 4) -> [midCs1F, midCs2F]
>   (D,  4) -> [midDF];  (Ds, 4) -> [midDsF]; (E,  4) -> [midEF]
>   (F,  4) -> [midFF];
>   (Fs, 4) -> [midFs1F, midFs2F]
>   -- high range
>   (G,  4) -> [highGF]; (Gs, 4) -> [highGsF]; (A,  4) -> [highAF]
>   (As, 4) -> [highAs1F, highAs2F, highAs3F, highAs4F]
>   (B,  4) -> [highBF]
>   (C,  5) -> [highC1F, highC2F]
>   (Cs, 5) -> [highCsF]; (D,  5) -> [highDF]; (Ds, 5) -> [highDsF]
>   (E,  5) -> [highE1F, highE2F, highE3F, highE4F]
>   (F,  5) -> [highF1F, highF2F]
>   -- altissimo (none yet)
>   _      -> error ("saxF: no fingerings for this pitch (in Eb): "
>                    ++ (show p))

> fingeringToPitch :: SaxophoneFingering -> Pitch
> fingeringToPitch (SaxFingering _ _ _ p) = p
```

For indexing, it is also useful to have a set of functions for mapping fingerings to their unique identifiers and back.

```
> lookupFingering :: SaxophoneFingeringNumber -> SaxophoneFingering
> lookupFingering n = case n of
>  1  -> lowBbF;   2  -> lowBF;    3  -> lowCF;    4  -> lowCsF
>  5  -> lowDF;    6  -> lowDsF;   7  -> lowEF;    8  -> lowFF
>  9  -> lowFs1F;  10 -> lowFs2F;  11 -> midGF;    12 -> midGsF
>  13 -> midAF;    14 -> midAs1F;  15 -> midAs2F;  16 -> midAs3F
>  17 -> midAs4F;  18 -> midBF;    19 -> midC1F;   20 -> midC2F
>  21 -> midC3F;   22 -> midCs1F;  23 -> midCs2F;  24 -> midDF
>  25 -> midDsF;   26 -> midEF;    27 -> midFF;    28 -> midFs1F
>  29 -> midFs2F;  30 -> highGF;   31 -> highGsF;  32 -> highAF
>  33 -> highAs1F; 34 -> highAs2F; 35 -> highAs3F; 36 -> highAs4F
>  37 -> highBF;   38 -> highC1F;  39 -> highC2F;  40 -> highCsF
>  41 -> highDF;   42 -> highDsF;  43 -> highE1F;  44 -> highE2F
>  45 -> highE3F;  46 -> highE4F;  47 -> highF1F;  48 -> highF2F
```

```
> fingeringNumber :: SaxophoneFingering -> SaxophoneFingeringNumber
> fingeringNumber (SaxFingering n _ _ _) = n
```

Finally, a function is provided to determine if two fingerings sound the same.

```
> soundsSame :: SaxophoneFingering -> SaxophoneFingering -> Bool
> (SaxFingering n _ _ _) `soundsSame` (SaxFingering n1 _ _ _)
>   = if n == n1 then True else b
>   where g = [[9,10], [14..17], [19..21], [22,23], [28,29], [33..36],
>              [38,39], [43..46], [47,48]]
>         b = or (map (\x -> (elem n x) && (elem n1 x)) g)
```

To demonstrate this representation, the following program prints to the console the entire saxophone fingering chart.

```
> printSaxFingeringChart :: IO ()
> printSaxFingeringChart = putStr saxFingeringChart

> saxFingeringChart :: String
> saxFingeringChart = "\nSaxophone Fingering Chart\n\n" ++ diagrams
>   where diagrams       = concat (map showDiagrams allSaxPitches)
>         showDiagrams x = (heading x) ++ (fingeringDiagrams x)
>                          ++ separator
>         heading      x = (show (trans (-3) x)) ++ ":"
>         separator      = "\n-----------------------" ++
>                          "----------------------\n"
>         -- all pitches in the (basic) saxophone range concert c#3 - Ab5
>         allSaxPitches :: [Pitch]
>         allSaxPitches = map pitch [(1+12*3)..(8+12*5)]

> fingeringDiagrams :: Pitch -> String
> fingeringDiagrams p
>   = concat (map (\d -> "\n" ++ (fingeringDiagram d)) ds)
>   where ds = saxophoneFingerings p
```

```
> fingeringDiagram :: SaxophoneFingering -> String
> fingeringDiagram (SaxFingering n lh rh _)  = show lh ++ " " ++ show rh
```

### 4.2 A Weighted Graph of Saxophone Fingerings

The algorithm for selecting a fingering sequence given a piece of music uses a weighted graph of saxophone fingering transitions, where the weight is an estimate of the difficulty of the transition. The higher the weight, the easier it is to play, or the more rapidly it may be played with ease. Given a sequence of pitches $< p_0, ..., p_n >$ the graph is used to find the sequence of fingerings $< f_0, ..., f_n >$ with the greatest weight $w$ where $w$ is the sum of all pairs $(f_x, f_{x+1})$.

To construct this graph, the author charted the maximum tempos at which he could play all of the transitions for which there are multiple transitions that produce the same pitch sequence. A default weight was used for all intervals for which there is only one fingering combination. Although somewhat time-consuming, this approach is comprehensive and produces very satisfactory results. Other techniques for populating this graph might include a rule-based approach, or the use of physical models to simulate difficulty in technique. For example one might collect a body of rules like "jumping octaves is costly", or "never use the F# fork key unless the adjoining pitch is F". Alternatively, one might encode the number and positions of fingers employed in each fingering, and then give a higher weight to transitions that cause the performer's fingers to move the least.

The type synonym `FingeringTempo` is used to encode the graph weights. It is a measure of the rapidity with which one can play a given fingering transition. Specifically, it represents the tempo (quarter-note beats per minute in 4/4) at which the author can play a given fingering sequence in 16th-notes. Unfortunately, it was not possible to display the weighted graph neatly when printing this paper. It is included here in part, and the source code is availble upon request.

```
> type FingeringTempo = Integer

> saxFGraph :: Graph SaxophoneFingering FingeringTempo
> saxFGraph = mkGraph False (lowBbF, highF2F)
>             ((mw lowBbF [(2, 50), (3, 112), (4, 54), (5, 44), (6, 26), (7, 52),
```

```
>              (mw lowBF ((ns 3 8)  ++ (f948 48  44  44 56  58  60  56  66 42 66
>              (mw lowCF  ((ns 4 8) ++ (f948 60  38  76 84  84  92  84  69 66 76
>              (mw lowCsF ((ns 5 8) ++ (f948 112 48  80 72  63  84  50  46 30 48
>              (mw lowDF  ((ns 6 8) ++ (f948 120 48  72 84  72  88  72  60 50 40
>              (mw lowDsF ((ns 7 8) ++ (f948 138 48  66 92  92  100 66  54 52 50
>              (mw lowEF  ((8, n):     (f948 144 84  76 116 104 126 76  54 46 100
>              (mw lowFF               (f948 108 168 80 138 46  130 100 70 56 100
>              (mw lowFs1F (zip [11..48] [168,160,168,96,72,168,140,108,126,70,100
>              (mw lowFs2F (zip [11..48] [144,140,140,50,126,40,116,100,126,66,30,
>              (mw midGF  ((ns 12 13) ++ (f1448 96  50 48 144 116 80 100 144 90  8
>              (mw midGsF ((13, n):      (f1448 96  50 48 136 112 76 80  140 100 8
>              (mw midAF                 (f1448 192 60 58 168 160 92 84  160 80  1
>              (mw midAs1F (zip [18..48] [144,116,30, 44, 100,50,52,52,52,52,66, 6
>              (mw midAs2F (zip [18..48] [168,60, 100,100,94, 66,76,76,76,76,60, 1
>              (mw midAs3F (zip [18..48] [160,58, 80, 90, 90, 68,74,74,74,50,120,6
>              (mw midAs4F (zip [18..48] [50, 63, 24, 104,144,70,68,68,68,68,110,1
>              (mw midBF   (f1948 50 150 54 160 60 100 90 120 170 160 60 46 100 60
>              (mw midC1F   (zip [22..48] [160,60,72,72,72,72,120,110,126,126,126,
>              (mw midC2F   (zip [22..48] [120,56,50,50,50,50,80, 70, 90, 90, 90,
>              (mw midC3F  (zip [22..48] [60,130,64,60,64,64,70, 50, 60, 60, 60, 3
>              (mw midCs1F  (zip [24..48]        [76,76,80,80,100,96, 126,126,126,
>              (mw midCs2F (zip [24..48]        [74,60,74,74,80, 70, 60, 60, 60, 7
>              (mw midDF  ((ns 25 27) ++ (f2848 100 30  52 76 74 68 72 50 50 30 40
>              (mw midDsF ((ns 26 27) ++ (f2848 100 30  52 76 74 68 72 50 50 30 40
>              (mw midEF  ((27, n):      (f2848 110 30  52 76 74 68 72 50 50 30 40
>              (mw midFF                 (f2848 50  110 52 76 74 68 72 50 50 30 40
>              (mw midFs1F (zip [30..48] [160,160,132,80,70,152,148,132,100,70,120
>              (mw midFs2F (zip [30..48] [100,100,100,76,140,60,120,100,90, 60,100
>              (mw highGF ((ns 31 32) ++ (f3348 120 90 90 140 130 94 50 30 40 28 9
>              (mw highGsF ((32, n):     (f3348 120 90 90 140 130 94 50 30 40 28 9
>              (mw highAF                (f3348 160 60 60 140 130 94 50 30 40 28 9
>              (mw highAs1F (zip [37..48] [130,104,50,110,90, 90, 30,20,28,20,30,2
>              (mw highAs2F (zip [37..48] [160,60, 64,100,80, 80, 50,20,40,20,48,2
>              (mw highAs3F (zip [37..48] [156,58, 60,100,80, 80, 48,20,38,20,46,2
>              (mw highAs4F (zip [37..48] [80, 64, 40,152,100,100,46,20,36,20,50,2
>              (mw highBF   (f3848 80 140 50 20 46 20 50 40)) ++
>              (mw highC1F (zip [40..48] [152,100,90,60,80,40,40,60,80])) ++
>              (mw highC2F (zip [40..48] [110,80, 70,50,30,42,30,50,30])) ++
>              (mw highCsF ((ns 41 42) ++ (f4348 110 100 120 90 100 90))) ++
>              (mw highDF  ((42, n) :     (f4348 108 70  80  70 90  70))) ++
```

```
>              (mw highDsF                (f4348 110 70  90  70 90  70)) ++
>              (mw highE1F                (f4748                138 52)) ++
>              (mw highE2F                (f4748                50  120)) ++
>              (mw highE3F                (f4748                100 40)) ++
>              (mw highE4F                (f4748                50  40)))
>  where n = 10 -- default weight
>        ns l h = [(x,n) | x <- [l..h]]
>        mw fingering weights = [(fingering, lookupFingering f, w) | (f,w) <- weig
>        f4748 w47 w48 = [(47, w47), (48, w48)]
>        f4348 w43 w44 w45 w46 w47 w48
>          = [(43, w43), (44, w44), (45, w45), (46, w46)] ++ (f4748 w47 w48)
>        f3848 w38 w39 w43 w44 w45 w46 w47 w48
>          = [(38, w38), (39, w39)] ++ (ns 40 42) ++ (f4348 w43 w44 w45 w46 w47 w4
>        f3348 w33 w34 w35 w36 w38 w39 w43 w44 w45 w46 w47 w48
>          = [(33, w33), (34, w34), (35, w35), (36, w36), (37, n)] ++
>            (f3848 w38 w39 w43 w44 w45 w46 w47 w48)
>        f2848 w28 w29 w33 w34 w35 w36 w38 w39 w43 w44 w45 w46 w47 w48
>          = [(28, w28), (29, w29)] ++ (ns 30 32) ++
>            (f3348 w33 w34 w35 w36 w38 w39 w43 w44 w45 w46 w47 w48)
>        f1948 w19 w20 w21 w22 w23 w28 w29 w33 w34 w35 w36 w38 w39 w43 w44 w45 w46
>          = [(19, w19), (20, w20), (21, w21), (22, w22), (23, w23)] ++
>            (ns 24 27) ++ (f2848 w28 w29 w33 w34 w35 w36 w38 w39 w43 w44 w45 w46
>        f1448 w14 w15 w16 w17 w19 w20 w21 w22 w23 w28 w29 w33 w34 w35 w36 w38 w39
>          = [(14, w14), (15, w15), (16, w16), (17, w17), (18, n)] ++
>            (f1948 w19 w20 w21 w22 w23 w28 w29 w33 w34 w35 w36 w38 w39 w43 w44 w4
>        f948 w9 w10 w14 w15 w16 w17 w19 w20 w21 w22 w23 w28 w29 w33 w34 w35 w36 w
>          = [(9, w9), (10, w10)] ++ (ns 11 13) ++
>            (f1448 w14 w15 w16 w17 w19 w20 w21 w22 w23 w28 w29 w33 w34 w35 w36 w3
```

The current implementation of the search algorithm is the trivial one – all of the possible fingering sequences are enumerated, and the highest value path is selected. This limits the usefulness of this function to short phrases. An algorithm that prunes the graph iteratively would increase the performance of this function.

```
> findFingeringSequence :: Music -> [SaxophoneFingering]
> findFingeringSequence m = findFingeringSeq (extractPitches m)
```

```
> findFingeringSeq :: [Pitch] -> [SaxophoneFingering]
> findFingeringSeq pitches
>   = findHWPath saxFValue (map (\p -> (saxophoneFingerings p)) pitches)

> findAllFingerings :: Music -> [[SaxophoneFingering]]
> findAllFingerings m = ffs (extractPitches m) []
>   where extendS f s' = map (\s -> s++[f]) s'
>         ffs :: [Pitch] -> [[SaxophoneFingering]] ->
>                [[SaxophoneFingering]]
>         ffs []     s  = s
>         ffs (p:ps) []
>           = ffs ps (map (\f -> [f]) (saxophoneFingerings p))
>         ffs (p:ps) s'
>           = ffs ps (concat (map (\f -> extendS f s')
>                                 (saxophoneFingerings p)))

> saxFValue :: [SaxophoneFingering] -> Integer
> saxFValue []       = 0
> saxFValue (x:[])   = 0
> saxFValue (a:b:xs) = if (a == b)
>                      then restC
>                      else if (a `soundsSame` b)
>                           then -100
>                           else ((weight a b saxFGraph) + restC)
>   where restC = saxFValue (b:xs)
```

# 5 The Performer Model

The `Performer` module implements a model of a performer's technical abilities and a persistence mechanism for that data. It also includes a function for computing a difficulty score for a given piece of music, and a given performer.

```
> module Performer where
> import Ratio
> import IO
> import Array
> import Basics
```

```
> import ComposersWorkbench
> import Miscellany
> import SaxophoneFingerings
```

### 5.1 Representation

A `Performer` has a name and description of abilities. Currently, this discription is limited to the area of technical speed, but this representation could be expanded later to accommmodate areas such as intonation, atriculation, and improvisation. The performer's technical speed is represented by a map of fingering transitions to `FingeringTempo`s where the mapped tempo is the maximum tempo at which the performer has been observed to perform the transition successfully. Each transition is also mapped to a confidence level which indicates the degree of confidence in the mapping between the transition and the tempo. As more observations of a performer's performance of a given transition are input into the system, this confidence level is increased.

```
> type Name = String
> type ConfidenceLevel = Integer -- from 1 to 10
> type SaxFCombination
>   = (SaxophoneFingeringNumber, SaxophoneFingeringNumber)
> type TechniqueMap
>   = Array SaxophoneFingeringNumber
>           (Array SaxophoneFingeringNumber
>                  (FingeringTempo, ConfidenceLevel))

> data Abilities = Abilities TechniqueMap
>  deriving (Read, Show, Eq)

> data Performer = Performer Name Abilities
>  deriving (Read, Show, Eq)

> maximumTempo :: TechniqueMap -> SaxFCombination -> FingeringTempo
> maximumTempo t s = fst ((t ! (minp s)) ! (maxp s))

> confidence :: TechniqueMap -> SaxFCombination -> ConfidenceLevel
> confidence t s = snd ((t ! (minp s)) ! (maxp s))
```

```
> updateRecord :: TechniqueMap -> SaxFCombination -> FingeringTempo ->
>                 ConfidenceLevel -> TechniqueMap
> updateRecord t s f c = t//[(minp s, innerArray)]
>  where innerArray = (t ! (minp s))//[(maxp s,(f,c))]

> maxConfidence = 10
> lowConfidence = 5
> minConfidence = 1

> -- Max and Min for pairs
> maxp (a, b) = max a b
> minp (a, b) = min a b
```

### 5.2 Persistence

Since the model of the performer will be updated iteratively, through interaction with the performer over long periods of time, it is necessary to persist the data to disk. An additional data type and accompanying functions are provided for this purpose. Haskell's `Read` and `Show` classes are used to implement this functionality, by reading and writing the data as text.

```
> data PerformerDB = PerformerDB FilePath Performer
>  deriving (Read, Show, Eq)

> -- An "initial" performer database with the "default" set of abilities
> makePDB :: FilePath -> Name -> PerformerDB
> makePDB f n = PerformerDB f (makePerformer n)

> makePerformer :: Name -> Performer
> makePerformer n = Performer n (Abilities t)
>  where t     = array (1, 48) [(n, innerArray n) | n <- [1..48]]
>        innerArray x
>          = array (x, 48) [(y, (tInit, cInit)) | y <- [x..48]]
>        tInit = 15
>        cInit = 2

> -- I/O operations
> loadDB :: FilePath -> IO PerformerDB
```

```
> loadDB f = do h <- openFile f ReadMode
>               s <- hGetContents h
>               hClose h
>               return $ r s
>  where r :: String -> PerformerDB
>        r = read

> storeDB :: PerformerDB -> IO ()
> storeDB pdb@(PerformerDB f _)
>  = do h <- openFile f WriteMode
>       hPutStr h (show pdb)
>       hClose h
```

### 5.3 Difficulty Estimation

One use of the performer database is to estimate, for any musical phrase, the difficulty of that phrase. In the following, an algorithm is introduced for the computation of a "difficulty measure" for a given phrase and performer. This estimate is also given a (composite) confidence level, or degree of accuracy.

This difficulty measure is a weighted average where fingering transitions in the phrase that are "out of reach" for a performer are given a high weight compared to transitions that are "difficult but doable" which recieve a lower weight, and transitions that are "easy", which are given no weight. Thus, a phrase whose transitions all are at speeds at or beneath the associated speeds in the `TechniqueMap` will have a difficulty score of zero, while phrases that have some intervals whose speeds are within a small amount of the speeds associated in the `TechniqueMap` will have a small difficulty measure, and phrases that include even a few transitions whose speeds are greater than a small amount more than the associated speeds in the `TechniqueMap` will have a high difficulty measure.

Where $s_p$ is the speed of a given transition in the phrase, $s_d$ is a speed of a given transition in the database, $b$ is a small number, $c$ is the number of "close" transitions in the phrase, for which $s_p < s_d + b \wedge s_p > s_d$, $f$ is the number of "far" transitions, for which $s_p >= s_d + b$, and $e$ is the number of "easy" transitions, for which $s_p <= s_d$, then the difficulty measure is

$$\frac{c + 10 * f}{10 * (e + c + f)}.$$

```
> type Difficulty = Double -- bounded from 0 to 1
> type DegreeOfConfidence = Double -- average from 1 to 10

> -- Takes a performer, a music, beats per minute, beats per measure,
> --   and returns a difficulty rating.
> estimateDifficulty :: Performer -> Music -> Integer -> Integer ->
>                       (Difficulty, DegreeOfConfidence)
> estimateDifficulty p@(Performer _ (Abilities t)) m bpm bpms = (d, c)
>   where c = (fromInteger $ sum $ map (confidence t) s) /
>             (fromInt $ length s)
>         d :: Difficulty
>         d = (fromInt ((nct + (10 * nft)))) /
>             (fromInt (10 * (net + nct + nft)))
>         s :: [SaxFCombination]
>         s = fingeringCombinations m
>         nct :: Int
>         nct = tmc (\(t',t'') -> (t' < (t'' + buffer)) && t' > t'')
>         nft = tmc (\(t',t'') -> t' >= (t'' + buffer))
>         net = tmc (\(t',t'') -> t' <= t'')
>         tmc :: ((FingeringTempo, FingeringTempo) -> Bool) -> Int
>         tmc p = length $ filter p tm
>         -- tm: tempo map (a,b) where a is T_Phrase and b is T_KB
>         tm :: [(FingeringTempo, FingeringTempo)]
>         tm = zip (fingeringTempos m bpm bpms) $ map (maximumTempo t) s
>         buffer = 10 -- the amount by which we want the performer to "reach"

> fingeringCombinations :: Music -> [SaxFCombination]
> fingeringCombinations m
>   = zipSeq $ map fingeringNumber $ findFingeringSequence m

> -- Takes music, bpm, and beats per measure
> fingeringTempos :: Music -> Integer -> Integer -> [FingeringTempo]
> fingeringTempos m bpm bpms
>   = map (durToFT bpm bpms) $ take ((length pd) - 1) pd
>   where pd = extractDurations m -- durations of all notes in the phrase

> -- durToFT normalizes durations in the KB to fingering tempos. It
> --   takes the bpm and bpMeas of a phrase and the duration of a note
> --   and returns a tempo. It also rounds down the derived tempo to
> --   the nearest integer.
```

```
> durToFT :: Integer -> Integer -> Dur -> FingeringTempo
> durToFT bpMin bpMeasure dur
>   = floor $ (fromInt (numerator r)) / (fromInt (denominator r))
>   where r = (fromInteger bpMin) /
>             ((fromInteger 4) * dur * (fromInteger bpMeasure))
```

# 6 Performance Evaluation

To build a database of information about a performer's abilities one must provide a means to input this information in a machine processible format. While it would be ideal to extract this information from the recorded audio and video of a set of performances, such a task is beyond the scope of this research project. On the other extreme, requiring the performer to measure in detail his abilities and enter them into a database is impractical. The `PerformanceEvaluation` module attempts to strike a practical balance between ease of use and ease of implementation in this regard.

```
> module PerformanceEvaluation where
> import Basics
> import SaxophoneFingerings
> import Performer
> import Scores
```

The `PerformanceEvaluation` data type represents an evaluation of a given performance. This evaluation is of a particular phrase in a larger work, and it indicates whether or not the performer played that phrase successfully. `PerformanceEvaluation`s are linked to the phrases they describe by a phrase identifier, which is a score identifier (the title of the score) combined with an integer which indicates the phrase number (0 for the first phrase, 1 for the second, etc.) of that score. If the performer did not play the phrase successfully, a reason is provided.

```
> data PerformanceEvaluation
>  = CanPlay PhraseID | CannotPlay PhraseID Reason
>     deriving (Read, Show, Eq)
```

```
> data PhraseID = PhraseNum Integer ScoreID
>     deriving (Read, Show, Eq)

> type ScoreID = String

> data Reason = TooHigh | TooLow | TrickyRhythm
>             | TooLong | TrickyFingering
>     deriving (Read, Show, Eq)
```

This representation is modelled after the common practice of marking up a score with annotations. For example, a music teacher might read the score while the student plays, and use a pencil to circle and annotate phrases that were played incorrectly. By encoding those annotations in a machine processible form, a foundation is created for building a rough measure of a performer's technical facility, programmatically. The system may then develop further etudes that will help "hone in" on the performer's weaknesses in technique.

## 6.1 Updating the Performer Database

Performance evaluations include useful information about the performer's technical facility that one would like to extract. For each performance, it would be helpful to find the phrases that were successful or failed due to difficulty of technique, and assimilate that information into the performer database. The `updateWithEvals` function takes a list of scores, a list of performance evaluations, and a performer database, and it updates that database according to information derived from the performance evaluations.

```
> updateWithEvals :: [Score] -> [PerformanceEvaluation] ->
>                    TechniqueMap -> TechniqueMap
> updateWithEvals s [] t = t
> updateWithEvals s (p:ps) t
>   = updateWithEvals s ps (updateWithEval s p t)

> updateWithEval :: [Score] -> PerformanceEvaluation ->
>                   TechniqueMap -> TechniqueMap
> updateWithEval s (CanPlay p) t
```

```
>   = updateWithEvalBy updateWithSuccess s p t
> updateWithEval s (CannotPlay p TrickyFingering) t
>   = updateWithEvalBy updateWithFailure s p t
> -- no assimilations yet for other types of failure
> updateWithEval _ (CannotPlay _ _) t = t

> updateWithEvalBy :: (Music -> Integer -> Integer ->
>                       TechniqueMap -> TechniqueMap) ->
>                      [Score] -> PhraseID ->
>                      TechniqueMap -> TechniqueMap
> updateWithEvalBy f scores (PhraseNum i title) t = f m bpm bpMeas t
>   where m :: Music
>         m      = phraseNum i s
>         bpMeas = timeSigToBPMeas timeS
>         s@(Score _ timeS bpm _ _) = scoreWithTitle title scores
```

This function is implemented with two update functions – one which is applied for successful phrases, and another that is applied for phrases that were performed unsuccessfully. In the case of a phrase that was successfully performed, the database is updated to reflect the performer's ability to play each of the fingering transitions that occurred within the phrase. The `updateWithSuccess` function takes a phrase, a tempo in beats per minute, and a number of beats per measure, and returns a function which updates the performer database accordingly.

For each transition in the phrase, if the speed of that transition in the phrase is higher than the maximum speed of that transition in the database, then the database is updated with the new speed. If the speeds are equal, then the confidence in that transition is incremented. Otherwise the confidence in that interval is incremented by a small amount, but not more than a small amount over the "low confidence" level.

```
> updateWithSuccess :: Music -> Integer -> Integer ->
>                       TechniqueMap -> TechniqueMap
> updateWithSuccess = updateBy updateTSucc

> updateTSucc :: (SaxFCombination, FingeringTempo) ->
>                TechniqueMap -> TechniqueMap
> updateTSucc (d,pt) t' = updateRecord t' d newT newC
```

```
>   where kbt  = maximumTempo t' d
>         kbc  = confidence t' d
>         newT = if kbt < pt then pt else kbt
>         newC = if kbt == pt
>                then min (kbc + 1) maxConfidence
>                else (if kbc < (lowConfidence + 1) then kbc + 1 else kbc)
```

For phrases that were unsuccessfully performed due to technique problems, it is not certain which of the transitions within the phrase are the source of difficulty, so every transitions' confidence level is decreased. In the etude generation step, these "low confidence" transitions will be selected for testing so as to "hone in" on the particularly difficult transitions for that performer. Like `updateWithSuccess`, `updateWithFailure` takes a phrase, a tempo in beats per minute, and a number of beats per measure, and returns a function which updates the performer database accordingly.

```
> updateWithFailure :: Music -> Integer -> Integer ->
>                      TechniqueMap -> TechniqueMap
> updateWithFailure = updateBy updateTFail

> updateTFail :: (SaxFCombination, FingeringTempo) ->
>                TechniqueMap -> TechniqueMap
> updateTFail (d,pt) t' = updateRecord t' d kbt newC
>   where kbt  = maximumTempo t' d
>         kbc  = confidence t' d
>         newC = if kbc > lowConfidence
>                then lowConfidence
>                else max minConfidence (kbc - 1)
```

A general function is used to update the database in both cases, based on the phrase and specific update function provided.

```
> updateBy :: ((SaxFCombination, FingeringTempo) ->
>               TechniqueMap -> TechniqueMap) ->
>             Music -> Integer -> Integer -> TechniqueMap -> TechniqueMap
> updateBy g m bpm bpMeas t = f (zip s ft) t
```

```
>   where s :: [SaxFCombination]
>         s  = fingeringCombinations m
>         ft = fingeringTempos m bpm bpMeas
>         f :: [(SaxFCombination, FingeringTempo)] ->
>              TechniqueMap -> TechniqueMap
>         f []     t' = t'
>         f (x:s') t' = f s' (g x t')
```

# 7 Etude Generation

One use of the performer model is in the generation of simplistic yet useful etudes, customized to match the abilities of a given performer. The `EtudeGeneration` module includes a set of functions for generating such etudes.

```
> module EtudeGeneration where
> import Ratio
> import List
> import Basics
> import ComposersWorkbench
> import Miscellany
> import Scores
> import SaxophoneFingerings
> import Performer
```

## 7.1 Selecting Transitions to Test

The etudes that are generated will be simple sequences of two- or three-note melodies, each of which includes a transition which it would be beneficial for the performer to exercise. The tempo of the etude and the rhythms of its melodies are chosen such that each transition is played at or near the performer's maximum speed.

```
>  -- Fingering Transition
> type FingeringTrans = (SaxophoneFingering, SaxophoneFingering)
> type TempoRange = (Integer, Integer)
```

```
> -- Transition with Tempo Ranges
> type TransitionWTR = (FingeringTrans, TempoRange, [(TempoRange, Dur)])
```

To determine the transitions that will appear in the etude, a function is provided which takes a musical phrase and finds all of the fingering transitions in the performer database that are either at a lower speed in the database than they are in the phrase, or are of a low confidence in the database. Since each transition will become a two- or three-note phrase in the etude, selecting low confidence transitions will allow these transitions to be tested in isolation. The selection of transitions of lower speed in the database than in the phrase is perhaps obvious, since these are precisely the transitions that the performer needs to exercise to be able to play the phrase correctly.

```
> transitionsToTest :: TechniqueMap -> Music -> Integer -> Integer ->
>                      [FingeringTrans]
> transitionsToTest t m bpm bpMeas = map f (filter isTestable fctm)
>   where f :: (SaxFCombination, FingeringTempo) -> FingeringTrans
>         f ((a, b), _) = (lookupFingering a, lookupFingering b)
>         isTestable :: (SaxFCombination, FingeringTempo) -> Bool
>         isTestable (fc, mt)
>           = (confidence t fc) <= lowConfidence ||
>             mt > (maximumTempo t fc)
>         ft :: [FingeringTempo]
>         ft = fingeringTempos m bpm bpMeas
>         fctm :: [(SaxFCombination, FingeringTempo)]
>         fctm = zip (fingeringCombinations m) ft
```

The `generateEtude` function takes this list of transitions and then generates an etude. This is a multi-step process that is described in depth in the remainder of this section.

```
> generateEtude :: TechniqueMap -> [FingeringTrans] -> Title -> Score
> generateEtude tm ts title = Score title (4%4) t (Key C Major) m
>   where t = selectTempo trs -- tempo
>         (trs,twtrs') = findTempoRI twtrs -- the origional twtrs
```

```
>         twtrs = (genRanges.sortTrans.(trainingRanges tm)) ts
>         m = sequencePhrases mels -- music
>         -- melody fragments, derived from each transition
>         mels :: [Music]
>         mels = map (\(ft, tr, tdm) -> altMel t ft tdm) twtrs'
>         altMel :: Integer -> FingeringTrans -> [(TempoRange, Dur)] ->
>                   Music -- alternatig Melody fragment generator
>         altMel tempo ft tdm
>           = alternatingMelody (findPitchSeq ft) (selectDur tempo tdm)
```

### 7.2 Generating a Tempo and Note Durations

First, the the maximum tempo for each transition is retrieved from the performer database, and a tempo range is created to indicate the range of preferred speeds a which each transition should be tested, in the etude. These transitions and associated tempo ranges are then ordered from the lowest to the highest range. This ordering is used in future steps to assure that the best mapping from tempo ranges to durations of notes is used.

```
> trainingRanges :: TechniqueMap -> [FingeringTrans] ->
>                   [(FingeringTrans, TempoRange)]
> trainingRanges tm ts = map f ts
>   where f t = let a = (maximumTempo tm (transToCombo t))
>               in (t, (a, a+10))

> transToCombo :: FingeringTrans -> SaxFCombination
> transToCombo (a,b) = (fingeringNumber a, fingeringNumber b)

> sortTrans :: [(FingeringTrans, TempoRange)] ->
>              [(FingeringTrans, TempoRange)]
> sortTrans = sortBy (\(a,b) (c,d) -> compare b d)
```

Next, a set of tempo ranges for various note duration types is generated. Since, for example, a sixteenth note at 40 bpm has the same real duration as a quarter note at 160 bpm, it is useful to generate, for each transition, a tempo range for each note type that might be used in the etude. Later, a particular set of note durations will be chosen from these sets of ranges

which will allow one tempo to be used for all of the different parts of the etude.

```
> genRanges :: [(FingeringTrans, TempoRange)] -> [TransitionWTR]
> genRanges = map genTWTR
>   where genTWTR :: (FingeringTrans, TempoRange) -> TransitionWTR
>         genTWTR (s, t@(t1, t2)) = (s, t, tempoRangeMap t)

> tempoRangeMap :: TempoRange -> [(TempoRange, Dur)]
> tempoRangeMap (t0,t1) = filter isModerate allTs
>   where daf :: [(Dur, Ratio Int)]  -- Durs and Factors
>         daf = [(1%32, 1%2), (1%24, 2%3), (1%16, 1), (1%12, 4%3),
>                (1%8, 2), (1%6, 8%3), (1%4, 4)]
>         allTs :: [(TempoRange, Dur)]
>         allTs = map f daf
>         f :: (Dur, Ratio Int) -> (TempoRange, Dur)
>         f (a,b) = ((g t0 b, g t1 b), a)
>         g :: Integer -> Ratio Int -> Integer
>         g x y = (iRatioFloor.ratioIntToRational) ((fromInteger x) * y)
>         isModerate ((a,b),c) = (a >= 30) && (b <= 170)
```

The next step is to search for a tempo to use for the etude by finding the interstection of all of the tempo ranges generated by the previous step. As required, the lower bound of a given tempo range may be lowered to make it match the others, but the upper bound will never be raised. This assures that the difficulty of the piece will never be too great for the performer.

```
> -- Find Tempo Range Intersection
> --   The input must be sorted for this function to work properly
> findTempoRI :: [TransitionWTR] -> ([TempoRange], [TransitionWTR])
> findTempoRI [] = error "findTempoRI: list cannot be empty"
> findTempoRI ((t@(_,_,tr)):ts) = findTempoRIAcc ts ((map fst tr),[t])

> findTempoRIAcc :: [TransitionWTR] -> ([TempoRange], [TransitionWTR]) ->
>                   ([TempoRange], [TransitionWTR])
> findTempoRIAcc [] x = x
> findTempoRIAcc ((t@((s1,s2),(tempo0,tempo1),trd)):ts) a@(tr',t')
```

```
>   = if (null x)
>     then findTempoRIAcc (((s1,s2),y,z):ts) a
>     else findTempoRIAcc ts (x, t'++[t])
>     where x = intersectTR (map fst trd) tr'
>           y = (tempo0 - 5,tempo1)
>           z :: [(TempoRange, Dur)]
>           z = tempoRangeMap y

> intersectTR :: [TempoRange] -> [TempoRange] -> [TempoRange]
> intersectTR [] _ = []
> intersectTR _ [] = []
> intersectTR (r:rs) rs1 = (limitTR r rs1) ++ (intersectTR rs rs1)

> limitTR :: TempoRange -> [TempoRange] -> [TempoRange]
> limitTR _        [] = []
> limitTR (t0,t1) ((t2,t3):rs)
>   = if t2 <= t1 && t3 >= t0 && t1 >= t0
>     then (a,b):(limitTR (t0,t1) rs)
>     else limitTR (t0,t1) rs
>     where a = if t0 <= t2 then t2 else t0
>           b = if t3 >= t1 then t1 else t3
```

A tempo is then selected that is in the middle of the range of acceptable tempos determined previously. For each transition, the largest note duration whose tempo range includes the selected tempo for the etude is selected for use in one of the two- or three-note melodies in the etude.

```
> selectTempo :: [TempoRange] -> Integer
> selectTempo rs = floor $ (fromInteger (mr0 + mr1)) / 2
>   where (mr0,mr1) = median rs

> median :: (Ord a) => [a] -> a
> median [] = error "can't find the median of an empty set"
> median x  = x' !! (floor ((fromInt (length x')) / 2))
>   where x' = sort x

> selectDur :: Integer -> [(TempoRange, Dur)] -> Dur
> selectDur t m = maximum ds
```

```
>   where ds = [d | (_,d) <- filter (inRange t) m]
>         inRange t' ((t0,t1),_) = t' >= t0 && t' <= t1
```

### 7.3 Generating Pitches and Assembling the Etude

Once the durations have been selected for each two- or three-note melody, the pitch content for each melody is determined. This is a little trickier than it might appear at first, since one would like to select a sequence of pitches for which the "correct" fingering sequence includes the transition to be tested. In most cases, it is adequate to select a pitch sequence by simply mapping the fingerings in the sequence to the pitches they produce. In the case where the derived melody does not include the desired transition, however, one must modify that melody so that the transition will occur naturally. The easiest way to do this is to replace all occurrences of the pitch $x$, whose fingering ($f_{x1}$) is not the desired fingering ($f_{x2}$) with a sequence of pitches $< x, y, x >$ where the transition $< f_{x2}, f_y >$ has the highest value of all transitions from $f_{x2}$ to some $f_y$.

Consider, for example, the fingering sequence $< f_{fork}, f_a >$ where $f_{fork}$ is the fingering for the awkward, but sometimes useful "fork key" fingering for F# in the key of Eb, and $f_a$ is the fingering for A in the key of Eb. The correct fingering for the melody $< F\#, A >$ is $< f_m, f_a >$, where $f_m$ is the "middle finger" fingering for F#. Clearly, the trivial mapping of fingerings to pitches is not adequate in this case. The correct fingering for $< F\#, F, F\#, A >$ , however, is $< f_{fork}, f_f, f_{fork}, f_a >$, where $f_f$ is the fingering for F-natural. This melody does include the desired transition, $< f_{fork}, f_a >$. In effect, the insertion of the pitch F, "forces" the "relatively awkward" fingering sequence $< f_{fork}, f_a >$, for which we are trying to generate a melody.

```
> findPitchSeq :: FingeringTrans -> [Pitch]
> findPitchSeq (s1,s2) = (f s1 s1p) ++ (f s2 s2p)
>   where initSeq = findFingeringSeq [s1p,s2p]
>         s1p     = fingeringToPitch s1
>         s2p     = fingeringToPitch s2
>         f a b   = if (a `elem` initSeq)
>                   then [b]
>                   else [b,lowestCostP a,b]
```

```
> lowestCostP :: SaxophoneFingering -> Pitch
> lowestCostP s = fingeringToPitch s'
>   where s' = maximumBy f allFingerings
>         allFingerings = map lookupFingering [1..48]
>         f :: SaxophoneFingering -> SaxophoneFingering -> Ordering
>         f s0 s1 = compare (g s0) (g s1)
>         g s' = saxFValue [s,s']
```

Now, given a set of melodic fragments that we would like to appear in the etude, all that remains is to arrange them together into one piece of music. Perhaps the simplest way to do this is to make one phrase for each fragment, ending each with a held tone. The `alternatingMelody` function is used to assemble the pitches and durations for each transition into one of these phrases. Among other things, the function assures that the generated phrase will begin and end on the downbeat.

```
> alternatingMelody :: [Pitch] -> Dur -> Music
> alternatingMelody p d = makeMusic p' dSeq
>   where rp  = concat (repeat p)
>         isTriplet = (rd' `mod` 3) == 0
>         rd' = if (isWholeNum rd)
>               then iRatioFloor rd
>               else error "input duration is not a supported note length"
>         rd  = recip d
>         np' = if isTriplet then 13 else 9
>         p'  = take np' rp
>         na  = (length p') - 1
>         minLastD  = (fromInt na) * d
>         minTotalD = 2 * minLastD
>         lastD :: Dur
>         lastD = if isWholeNum minTotalD
>                 then minLastD
>                 else (fromInt (ceiling (fromIRatio minTotalD)))
>                      - minLastD
>         dSeq  = (take na (repeat d)) ++ [lastD]
>         makeMusic :: [Pitch] -> [Dur] -> Music
>         makeMusic ps ds = foldl1 (:+:) (zipWith (\p d -> (Note p d [])) ps ds)
```

This procedure for generating an etude produces functional but aesthetically uninteresting etudes. Similar techniques to those shown here might be employed to rearrange the melodic frangments to make the result more interesting, melodically and rhythmically.

### 7.4 An Example of Etude Generation

Given a very simple etude (shown in figure 2), the initial (empty) performer database, and a set of performance evaluations, the etude generation functions can be used to create the etude shown in Figure 3.

```
> module EtudeGenerationExample where
> import Ratio
> import Basics
> import ComposersWorkbench
> import Scores
> import Performer
> import PerformanceEvaluation
> import EtudeGeneration

> simpleEtude :: Score
> simpleEtude = Score "A very simple etude" (4%4) 108 (Key C Major) m
>   where m  = sequencePhrases [p1, p2]
>         p1 = makeMelody oct p1PCS p1Rhythm
>         p2 = makeMelody oct p2PCS p2Rhythm
>         p1PCS = [4,2,0,2,4,0,0,4]
>         p2PCS = [5,4,2,4,5,2,0]
>         p1Rhythm = en4qn2 :&: hn :&: hn
>         p2Rhythm = en4qn2 :&: (NoteR 1)
>         en4qn2 = en :&: en :&: en :&: en :&: qn :&: qn
>         qn             = NoteR (1 % 4)
>         en             = NoteR (1 % 8)
>         hn             = NoteR (1 % 2)
>         oct = (12*5)

> rttEtude2 :: Score
> rttEtude2 = generateEtude tm ts title
>   where title = "Round Trip Test: Etude Based on the Simple Etude"
>         ts :: [FingeringTrans]
```

```
>         ts = transitionsToTest tm (phraseNum 1 simpleEtude) bpm
>                                   (timeSigToBPMeas timeS)
>         (Score t timeS bpm _ _) = simpleEtude
>         tm :: TechniqueMap -- updated Technique Map
>         tm = updateWithEvals [simpleEtude] pes tm0
>         pes :: [PerformanceEvaluation]
>         pes = [CanPlay (PhraseNum 0 t),
>                CannotPlay (PhraseNum 1 t) TrickyFingering]
>         (Performer _ (Abilities tm0)) = defPerformer

> defPerformer = makePerformer "Sample Performer with Default values"
```

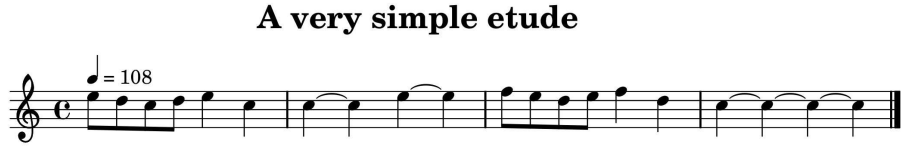


Figure 2: A simple etude.

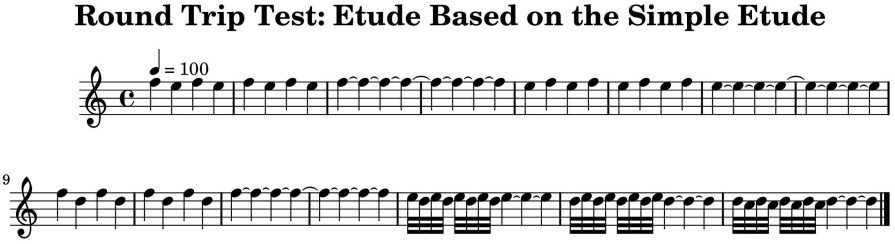


Figure 3: An etude generated automatically, based on the second phrase of the etude shown in Figure 2.

# 8 Other Modules

The system described in this paper includes a few modules that have not yet been mentioned, but are available on request. Many test functions, sample scores, and supporting functions for tasks like writing text to disk can be found in the `Test` module. A modified version of the `Graph` module

from Rabhi and Lapalme's collection of functional programming algorithms [RL99] is also used. This module has been modified by the author to include a measure of sparsity (as described by Rabhi and Lapalme) and an algorithm for finding paths through a graph based on a set of alternatives. The latter is an essential part of the fingering selection algorithm described previously in this paper. Also developed for this project is an implementation of the Recursive Best-First Search algorithm from Russell and Norvig's AI text. [RN03] The `BestFirstSearch` module was originally intended to serve as a mechanism for fingering selection, but it was eventually replaced by a simpler graph-based approach.

The work presented in this paper builds in part on an earlier project by the author to develop a module of functions for algorithmic composition and analysis, using Haskore (`ComposersWorkbench`). Some of the functions in the `ComposersWorkbench` module are also used in this project to simplify the construction and interpretation of Haskore data. As needed some minor functions for the manipulation of Haskore data were added to this module to support the work described in this paper. The `Testing` and `QuickCheck` modules are dependencies of `ComposersWorkbench` and so they are also required for compilation, although they are not used in the system.

Also, some generic functions for List manipulation and number conversion are included in the `Miscellany` module.

# 9 Futher Research

Perhaps the most interesting direction for further research would be the development of algorithms to modify the etudes generated by this system, to make more esthetically pleasing etudes, without increasing their difficulty. Another useful step would be to put this system into regular use as part of a musician's practice routine. The feedback from this experience would no doubt provide a useful test of the efficacy of this approach. Thirdly, it would be valueable to add another dimension of musicality to the performer model. Perhaps breath control or intonation might be good candidates for this extension of the model. Breath control is a well-defined aspect of musicianship, making imlementation in the performer database straightforward. Exercises for intonation might include an ear training component, for which music programming systems are well-suited. This might be a good way to make the system more attractive for adoption by musicians.

## 10 Acknowledgements

This paper was written under the direction of Paul Hudak as part of a one-semester independent research project. Thanks also go to the members of the LilyPond and Haskell email lists for sharing their knowledge of the LilyPond input format and Haskell persistence mechanisms.

## References

[DeV08] Paul DeVille. *Universal method for the Saxophone*. Carl Fischer, 1908.

[Hud00] Paul Hudak. *Haskore Music Tutorial*. Yale University, Department of Computer Science, New Haven, CT 06520, third edition, February 2000. Available online at http://haskell.org/haskore.

[Hud03] Paul Hudak. An algebraic theory of polymorphic temporal media. Technical Report YALEU/DCS/RR-1259, Yale University, Department of Computer Science, July 2003.

[HWN05] et. al. Han-Wen Nienhuys, Jan Nieuwenhuizen. *GNU LilyPond The music typesetter*, 2005. Available online at http://haskell.org/haskore.

[Jon03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[RL99] Fethi Rabhi and Guy Lapalme. *Algorithms: A Functional Programming Approach*. Addison-Wesley, 1999.

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 2003.

[Thi04] Henning Thielemann. Audio processing using haskell. *Proceedings of the 7th International Conference on Digital Audio Effects (DAFx '04)*, 2004.