The Music Tutor

A Summary Prepared for Vivace, Inc.

Roger B. Dannenberg
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
Phone: 412-268-3827
Fax: 412-268-6787

5 August 1992

This document describes the intellectual property of the Piano Tutor (Music Tutor), describes the effort needed to make a commercial Music Tutor product, and estimates the associated cost. The term *Piano Tutor* will refer to the specific research prototype constructed at CMU. We will use the term *Music Tutor* for general discussions of past, present, and future systems based on the Piano Tutor technology. Several documents are attached as appendices. Appendix I is a reprint of a technical paper on the Piano Tutor. This gives the most complete description of the Piano Tutor system. Appendix II is the final report on the Piano Tutor to the Markle Foundation. This gives the most up-to-date status of the system. Appendix III contains design notes on lesson selection, and Appendix IV has notes on the interface between Lisp and C in the Piano Tutor. Finally, Appendix V contains notes on the score file format.

1. Functional Description

- **Curriculum Analysis Software** is used to verify the consistency and completeness of the Piano Tutor curriculum, which consists of about 70 interrelated lessons.
- **Multimedia Presentations** are used by lessons to communicate to the student. There are about 100 of these presentations, which consists of the following media:
    - **Videodisc.** There are 30 minutes of video used by the current system. This is one side of one standard CAV laserdisc. Our video master is Betacam tape.
    - **MIDI files.** For each piece of music, there is one MIDI file giving a model performance. These are mostly two-channel piano performances, with the solo part on one channel, and a piano accompaniment on the other.
    - **Audio files.** There are a large number (~100) of voice messages. These are 8-bit audio files sampled at 11Khz.
    - **Music Notation files.** Music notation is stored as an ascii-coded display list. Note that it is not generated automatically from MIDI files, nor is MIDI

generated automatically from the notation files.

- **Scripts.** Multimedia presentations are organized and synchronized by scripts which sometimes include additional graphical animation commands. These scripts exist in two forms: a frame-based lisp representation that is editable and a "compiled" or stripped-down version for efficient playback only.

- **Curriculum.** The curriculum is a specification of all lessons, their prerequisites and objectives, their multimedia presentations and their instructional content. This results in several representations:
  - **Lesson Plans,** consisting of English descriptions, make up a high-level design specification.
  - **Music Compositions,** notated with Finale, exist for all music used in the Piano Tutor. These are original compositions by Drs. Sanchez and Joseph.
  - **Accompaniments for Music Compositions** exist as MIDI files (mentioned above).
  - **Theory Lesson Software.** A number of lessons involve special lesson-specific software, coded in Lisp.
  - **Automatic Lesson Selection Software.** The expert system that controls the Piano Tutor creates a student-specific curriculum during instruction. The expert system is written in Lisp.

- **Remediation.** The Piano Tutor expert system also generates remedial instruction in response to student errors. There are two main software components responsible for remediation.
  - **Performance Analysis Software.** After each student performance, performance information is analyzed to produce a list of low-level errors. These errors are further (re)classified at higher levels when possible. For example, a low-level pitch error may be reclassified as an octave error if it meets certain conditions.
  - **Remediation Selection Software.** Once errors have been classified, software is used to select what remedial message to give to the student.

- **Real-time Interaction** is a key element of the Piano Tutor, taking place during piano performance.
  - **Computer Accompaniment** software enables the Piano Tutor to follow a student's performance while synchronizing an accompaniment part. Computer accompaniment assumes the student's part and the accompaniment part are known in advance to the computer.
  - **Page Turning.** During performances, the Piano Tutor turns pages as follows: Music is displayed as two grand staves. At some point during the performance of the second grand staff, the first one is erased and redrawn with the the first grand staff of the next page. When the student begins again at the top of the display, the second grand staff is redrawn. At all times, the student can look ahead to the next measure.

2. Inventory

The previous section outlined the components of the Piano Tutor by function. Here we list the components by form. (The parenthesized topics refer to the function as described in the previous section.)

- Lisp Code (Curriculum Analysis Software, Theory Lesson Software, Automatic Lesson Selection Software, Performance Analysis Software, Remediation Selection Software, Multimedia Presentations)
- C Code (Real-time Interaction, Multimedia Presentations)
- Videodisc/Master Videotape (Multimedia Presentations)
- Audio files (Mutimedia Presentations)
- Lesson Plans and Scripts (Curriculum/Lesson Plans)
- MIDI Files (Multimedia Presentations, Curriculum/Music Compositions, Curriculum/Accompaniments)
- Music Notation Files (Multimedia Presentations)
- Multimedia Presentation Scripts (Multimedia Presentations)
- Music Manuscripts (Curriculum/Music Compositions)

3. Equipment Needed for Prototype System

- Mac II, 5MB RAM, 60MB disk, with Allegro Common Lisp and Think C.
- Videodisc Player: Pioneer LDV4200
- Electric Piano: e.g. Yamaha Clavinova CPL-350
- MIDI Synthesizer and Audio Mixer (Optional)

4. Capabilities: Prototype, Product, and Future

For an overview of the Piano Tutor, please see Appendix I. This section describes the required functionality for a Music Tutor product and what future products might do.

A Music Tutor product should have all of the basic capabilities of the Piano Tutor prototype; that is, it should incorporate a large curriculum, provide intelligent remediation using voice and music, and it should tailor instruction to the student's needs. In addition, the product should have a larger set of lessons and music examples. We found that the prototype covers introductory theory a little too fast and there are many times where alternate lessons would be helpful for slower learners.

At least two areas are open for future product versions. First, it might be interesting to explore non-piano instrument instruction. For most instruments, a MusicTutor would serve as a supplement to traditional instruction because the computer would not teach basic sound production. Another area is more advanced piano pedagogy. Additional analysis software could be written to help students play with even articulation (both timing and dynamics), steady tempo, and perhaps some amount of expression could be analyzed.

5. Relationships among MusicTutor, Computer Accompaniment, and PracticeMate

Computer Accompaniment is a basic technology used by both PracticeMate and MusicTutor. PracticeMate uses Computer Accompaniment to synchronize an accompaniment to a live performance. Providing accompaniment is the major function of PracticeMate. In the MusicTutor, synchronizing accompaniment is just one of the functions of the Computer Accompaniment technology, which is also used to follow and analyze student performances without accompaniment. The ability to follow a error-laden performance is a key enabling technology for the interactive instruction provided by MusicTutor. In short, MusicTutor augments PracticeMate with evaluation and interactive instruction. Most of the functions of PracticeMate are included in MusicTutor.

6. Time and Cost Estimates

It is very hard to estimate development costs without a clearer picture of the final product, but I will assume the following target: a Macintosh-based system with CD-ROM using the existing content and lessons of the CMU Piano Tutor. A CD-ROM would be large enough to hold all of the Piano Tutor video and audio in some QuickTime format as well as software, music, and other data.

The development and porting tasks are:

- Develop software video and audio drivers and interfaces (2 mos)
- Port presentation software from Lisp to C (3 mos)
- Port evaluation software from Lisp to C (6 mos)
- Convert presentation video, graphics, etc. (3 mos)
- Enhance and port user interface (3 mos)
- Write manual (2 mos)

Thus, the total effort to port and poductize the *existing* system is about 2 man years, and is probably doable by 2 people in one year.

Any significant changes will involve new design and may add significantly to the cost. For example, to enhance the content of the curriculum significantly would probably require shooting new video, recording new audio soundfiles, developing new notation, etc. This could easily occupy a content developer or two for one year, adding 50-100% to the development cost.

Similarly, the existing Piano Tutor has a simple but effective user interface. If this were elaborated into a sophisticated control panel (even as much as PracticeMate), the additional cost could be quite high. Consider that almost all of the software cost in Practice Mate is in supporting the interface features of the cue sheet, measure numbers, conducting modes, and rehearsal loops.

As a point of reference, the Miracle Piano Teaching System was developed by 20 people (including management and hardware) in one year. The major differences are:

- The Piano/Music Tutor software is much more sophisticated.

- The Piano/Music Tutor uses more multimedia.
- The Piano/Music Tutor has more types of animation.
- The Miracle System includes a hardware synthesizer.
- The Miracle System has two fairly elaborate video games.
- The Miracle System referred to here was written on an 8-bit Nintendo.

7. Other Considerations

Vivace should study the "Miracle Piano Teaching System" carefully. CMU has not obtained any information on the marketing of this product, but the sales figures and pricing for Mac, PC, and Nintendo versions should be illuminating. We considered a high-end Music Tutor based on the existing equipment ($5,000 - $10,000) and concluded that it was within the reach of schools and perhaps music stores (who could rent time on the Music Tutor), but not homes. The Miracle system seems to be selling in the $400 range to homes. A CD-ROM version of Music Tutor would fall somewhere in between these extremes.

I. A Computer-Based Multi-Media Tutor for Beginning Piano Students

II. Report to Markle Foundation

The Piano Tutor Project

Final Report 1991

Roger B. Dannenberg, Marta Sanchez, Annabelle Joseph,
Ronald Saul, Robert Joseph

November, 1991

8. Introduction

The Piano Tutor Project began as an ambitious project to integrate multimedia and intelligent tutoring technologies with a sound and extensive piano instruction curriculum. After three years of concentrated effort, the project produced a working prototype that has achieved international recognition as the state-of-the-art in computer-based music instruction systems. In a pilot study, the Piano Tutor has been used to teach piano to a group of 14 subjects. This study confirmed our belief that a computer-based instruction system could be very effective in teaching a substantial body of musical knowledge. Our subjects enjoyed using the Piano Tutor, they learned at a rapid pace, and they could transfer their new skills to sight-reading and performance tasks of music in standard piano instruction books.

There have been at least three other noteworthy achievements. The first is the application of score-following technology and advanced human computer interface concepts. The Piano Tutor is the only system to date that can follow a student's piano performance, turning pages automatically and synchronizing an accompaniment. This form of computer interface is not only fun for students, but it also challenges them to develop their own sense of rhythm. Automatic page turning is an important technique for presenting music to students via computer screen, and a by-product of accompaniment is a low-level analysis of student errors. In addition, the system uses an extensive multimedia presentation system to create a natural dialog with the student.

The second achievement is a contribution to Instructional Design theory. We applied Instructional Design concepts to develop the extensive Piano Tutor curriculum. We found Instructional Design to be helpful in providing general principles and guidelines, but we needed much stronger analytic tools to develop the Piano Tutor curriculum. We invented a formal representation for curricula, and we developed computer programs to analyze these curricula for a broad range of properties. This new approach to curriculum design has far-reaching implications for future computer-based instruction systems as well as for textbooks and hypermedia systems.

Our third achievement is the development of an expert system component for the Piano Tutor. The expert system analyzes student performances and gives pertinent feedback to guide the

student. Initially, the student is encouraged to correct mistakes by trying again, but if this fails, the system automatically proposes simpler tasks that help the student concentrate on the source of difficulty. If problems persist, the system assumes that the the student needs to strengthen previously learned skills. Again, the system identifies relevant tasks for the student.

The next section presents a more detailed list of our accomplishments. The following sections of this report present in greater detail what we feel are major achievements. The last section outlines the final status of the project, the outlook for commercial development, and other "spinoffs" of this research.

9. Accomplishments

The Piano Tutor was a large project with many components. Ultimately, the project may be remembered as a single integrated system, but we think it is appropriate to break this down into a number of individual accomplishments that contributed to the greater whole. One interesting aspect of the Piano Tutor is how well this complicated engine is hidden "under the hood" of the exterior interface.

The Piano Tutor curriculum includes one hundred (100) lessons, which are interrelated in a complex network of prerequisites and objectives. The design of the curriculum included an organization and enumeration of basic skills, music composition, script-writing, video production, graphics production, multimedia presentation development, programming, and testing.

We developed score-following software to turn pages and accompany the student during a performance (see Section 10.)

We developed software to analyze the curriculum early in the design phase (see Section 11.)

We developed music notation display software to be used in instruction and performance.

We developed a graphics toolkit for programming animations and graphical illustrations.

We developed a multimedia presentation editor that we use to construct (without further programming) animations, video, voice, graphic, text, and music presentations. This greatly expanded our ability to instruct with a limited amount of video play time.

We developed and tested a lesson selection strategy that adapts to the needs of students. (see Section 12.)

We developed performance analysis software that gives intelligent feedback to the student. (see Section 12.)

We produced and then revised a videodisc with material for all of our lessons.

We recorded and edited a substantial amount (about an hour continuous play time) of speech to supplement the videodisc.

We conducted a pilot study in which 14 students were taught piano by the Piano Tutor.

design a single curriculum, but to build an intelligent system that could design a new curriculum for each student. Our task became designing a set of lessons with explicit relationships to one another and to the underlying skills the Piano Tutor would attempt to teach.

We feel the result was a breakthrough in Instructional Design. We developed a formal representation for lessons in which every lesson has a list of prerequisites and objectives. The student's state of knowledge is modeled as a set of skills. Consistent with instructional design principles, a student is qualified to take a lesson when the prerequisites of the lesson are mastered by the student (contained in the student's skill set). If the student is successful in taking the lesson (lessons have behaviorally described evaluation criteria), then the objectives of the lesson are added to the student's set of skills.

The strength of this model lies in the fact that computers can simulate a student's progress through the curriculum. Computers can also analyze the network of lessons to determine whether certain desirable properties exist. For example, every skill should be taught by at least one lesson. Every lesson should teach something the student does not already know. There should be a way (via a sequence of lessons) to learn every skill. These maxims are extremely difficult to check for every skill and lesson. We developed techniques whereby computers can check for these and many other useful properties automatically in minutes. When errors are found, it is a relatively simple matter to revise the design since no time has been invested in scripting, media production, or programming. These tasks are performed only after a complete design has been validated by computer.

This formalized approach to curriculum design is not at all limited to the Piano Tutor, nor is it limited to intelligent teaching systems. We have discussed these ideas with other educators at Carnegie Mellon and found these ideas seem to formalize existing ideas of how computers might help teachers. One professor is interested in computer-assisted textbook authoring, and is hoping to apply our ideas. It also seems that hypermedia and hypertext systems, for which effective choices of links are an oft-cited problem, could benefit from the formal approach that we offer.

12. The Expert System

The Piano Tutor is by far the most sophisticated music-teaching computer system. To a great extent, the capabilities of the Piano Tutor are a result of its expert system component, which analyzes student performances and chooses how to proceed. In essence, this is where teaching knowledge is utilized. Some key aspects of the expert system are that is works in close conjunction with our curriculum design, and it operates at two levels: within lessons and outside of lessons.

An important design decision in the Piano Tutor was to factor the curriculum into two levels. At the global level, the problem is to select a lesson from a large set of possibilities. At the local level, the problem is to deliver the lesson, including specific suggestions in response to student performances. We found this two-level structure to work quite well. As noted above, the global level is more or less independent of the subject area, so much of our work here is relevant to other domains. Work at the global level focuses on curriculum design, and by dealing with fairly abstract lessons, we can more easily address curriculum planning issues.

Within lessons, interaction tends to be much more focussed. It has been well-known since the

design a single curriculum, but to build an intelligent system that could design a new curriculum for each student. Our task became designing a set of lessons with explicit relationships to one another and to the underlying skills the Piano Tutor would attempt to teach.

We feel the result was a breakthrough in Instructional Design. We developed a formal representation for lessons in which every lesson has a list of prerequisites and objectives. The student's state of knowledge is modeled as a set of skills. Consistent with instructional design principles, a student is qualified to take a lesson when the prerequisites of the lesson are mastered by the student (contained in the student's skill set). If the student is successful in taking the lesson (lessons have behaviorally described evaluation criteria), then the objectives of the lesson are added to the student's set of skills.

The strength of this model lies in the fact that computers can simulate a student's progress through the curriculum. Computers can also analyze the network of lessons to determine whether certain desirable properties exist. For example, every skill should be taught by at least one lesson. Every lesson should teach something the student does not already know. There should be a way (via a sequence of lessons) to learn every skill. These maxims are extremely difficult to check for every skill and lesson. We developed techniques whereby computers can check for these and many other useful properties automatically in minutes. When errors are found, it is a relatively simple matter to revise the design since no time has been invested in scripting, media production, or programming. These tasks are performed only after a complete design has been validated by computer.

This formalized approach to curriculum design is not at all limited to the Piano Tutor, nor is it limited to intelligent teaching systems. We have discussed these ideas with other educators at Carnegie Mellon and found these ideas seem to formalize existing ideas of how computers might help teachers. One professor is interested in computer-assisted textbook authoring, and is hoping to apply our ideas. It also seems that hypermedia and hypertext systems, for which effective choices of links are an oft-cited problem, could benefit from the formal approach that we offer.

12. The Expert System

The Piano Tutor is by far the most sophisticated music-teaching computer system. To a great extent, the capabilities of the Piano Tutor are a result of its expert system component, which analyzes student performances and chooses how to proceed. In essence, this is where teaching knowledge is utilized. Some key aspects of the expert system are that is works in close conjunction with our curriculum design, and it operates at two levels: within lessons and outside of lessons.

An important design decision in the Piano Tutor was to factor the curriculum into two levels. At the global level, the problem is to select a lesson from a large set of possibilities. At the local level, the problem is to deliver the lesson, including specific suggestions in response to student performances. We found this two-level structure to work quite well. As noted above, the global level is more or less independent of the subject area, so much of our work here is relevant to other domains. Work at the global level focuses on curriculum design, and by dealing with fairly abstract lessons, we can more easily address curriculum planning issues.

Within lessons, interaction tends to be much more focussed. It has been well-known since the

- "Yes. I had never learned Bass Clef before."
- "Very much! [and to the second question:] Not sure. It is teaching me to play the piano. (But the true judge would be a human teacher.)"
- "I enjoyed it very much but I'd like to learn more."
- "I think it is fun working with the Tutor. The hardest thing for me is to learn how to keep time. The Tutor gets me on that all the time. So, yes, I am learning what I want and *need* to learn."

Our survey shows that students enjoy the style of teaching embodied in the Piano Tutor and they feel is is a good way to learn.

Every student spent a small amount of time with human teachers to assess their progress and supplement the Piano Tutor by checking on posture, ease of movement, and other physical components of piano playing. In general, we found students were learning rapidly, and they were able to apply their knowledge and skills in sight-reading tasks outside the Piano Tutor curriculum. One of the surprises was that students seems to have better than normal ability to keep time and play in rhythm. We believe the constant feedback from the Piano Tutor promotes good practice habits, as illustrated by the last student comment listed above.

In summary, we set out to evaluate the Piano Tutor by testing it with a small pool of students. As expected, we found many small problems and we fixed many of them by augmenting the curriculum when it was found lacking. Overall, though, the testing went very smoothly. We found no serious problems with our approach, and we were gratified that students were so positive about the system. To a great extent, the numbers speak for themselves: most of our students stayed with the system to the end. Those that finished mastered what would normally be considered at least a one-year curriculum. The *average* time was 69 calendar days and less than 20 on-line hours.

14. The Future

There are now three operational Piano Tutor systems. We intend to keep these systems functional on the Carnegie Mellon campus for the foreseeable future. We expect the mere existence of these systems will inspire and inform a great deal of future work in music education and intelligent tutoring.

We have also begun to negotiate with a new company that is interested in developing a commercial version of the Piano Tutor. It appears to us that a product will require substantial new development, but we are encouraged by the discussions we have had so far.

There are more papers in progress. We plan to expand this report with additional technical material and publish in a computer science journal. Another article will explain our work on curriculum design and analysis in sufficient detail that others can begin using our techniques.

We have also had an inquiry about making the Piano Tutor available via ISDN so that students could "phone in" for piano lessons. This would be a pilot study initially, but it brings up a new marketing possibility.

15. Conclusion

The Piano Tutor Project established a new plateau of excellence in computer-based music instruction. Having received world-wide recognition, we feel confident that we have had an impact on the designers of future educational systems. We are pleased by the prospect of a commercial development of the Piano Tutor, but even if that does not take place, there are other areas with perhaps even greater potential that we have touched on. These include the ideas of active, intelligent, multimedia instruction, and formal support for curriculum design.

We are extremely indebted to the Markle Foundation for support, which made all of this possible. We are also pleased to acknowledge the contributions of Peter Capell, who participated in the design and implementation of the Piano Tutor.

III. Lesson Selection Design Notes

Ronald Saul
October 1991

An overview of the functionality in the curriculum or lesson selection portion of the Piano Tutor. The main curriculum process (accessed through the main menus) starts in the procedure guide-tutor-session in CurriculumEngine.lisp. The skill updating scheme when a student passes a lesson has been described many times (handled by the function pass-lesson and the related routines in lesson-structure.lisp).

Here I'll just describe the new features added in lesson-selection-process. These functions fire in the given order on the lesson pool. Only if a function does not select a lesson does control pass to the next function.

SELECT-PROBLEM-SKILL: When a student has been forced to withdraw by errors which related to a specific skill, that skill is added to a list of *problem-skills*. This function simply looks in the current pool for a lesson which has one of these skills as an objective and selects it. This implements a remedial backtracking scheme. With the current set of lessons, students are sometimes confused by this - It's not always obvious what skill is being addressed or the same remedial lesson keeps popping up ad nauseum.

SELECT-FOCUS-LESSON: In order to try to keep the lesson sequence somewhat directed, we choose a *focus-skill* from the objectives of a selected lesson. This skill remains the focus until it achieves the highest score. This function finds a lesson in the pool (if any) that teaches the current focus skill.

SELECT-ADVISORY: (Our current set of lessons don't include lessons appropriate for the 'advisory' label, so this function exists but was only tested with artificial set ups.) The idea is to have lessons labelled 'advisory which are appropriate to be reviewed at regular intervals after they have been learned. When a lesson counter has reached a set value, an 'advisory lesson is preferentially selected, either from the lesson pool or from a set of previously taught advisory lessons.

SELECT-HARDEST: Prefer the lesson in the pool with the most objectives. (This could easily be refined to "most potential point gain in objectives".) This simply serves to push students faster through the curriculum if they can handle it.

Overview of the lesson analysis procedure, primarily contained in the files "BeamSearch.lisp" "ErrorTypes.lisp" "DiagnosisFunctions.lisp" and called from the function analyze-performance in lesson-execution.lisp. After a student performance, the first step in error analysis is to run the performance and the target score through the function "beam-search" to extract the primitive pitch and rhythm errors. Beam-Search is essentially Roger Dannenberg's score matching algorithm recast onto a different data structure to make it easy for a lisp-AI kind of guy like me to grasp. It is also similar to the speech understanding system in Kai-Fu Lee's thesis. The code has many inefficiencies (such as floating point multiplies and search node creations) which could be improved upon, but the worst cases only seem to take a few seconds anyway. Note that this is a post process, so keeping up with each note as fast as it is played becomes a secondary consideration.

The basic idea is this: At any point in a performance we must be able to answer these questions about the student:

- Where is he?
- How fast is he going?
- What mistakes did he make while getting there?

Where? is expressed as the index into the score which corresponds to the current note of the performance. How fast? is measured by the ratio of the student's tempo to the tempo of the standard score. Two ratios are kept; The maximum believable ratio and the minimum believable ratio. I'll define "believable" a little later. Beware that internal tempi are measured in "Centiseconds per quarter-note" which is the inverse (times a constant) of standard metronome markings.

The student's pitch and rhythm mistakes are called "bugs" and are stored as list structures with the name of the mistake and pointers into the scores. e.g. (wrong 5 3) means the student's fifth note was a wrong pitch while we were expecting to hear the 3rd chord of the score. (Beware of another potential confusion - performance indices are measured in individual notes while indices into the score are simultaneous chords.) (skip 9 5 4) means that the student's ninth note, played when we were expecting the fifth chord of the score is actually in the fourth chord. i.e. the student's attention shifted backwards and he's repeating himself. (misplaced 4 4 .80) means the student's fourth note (= fourth chord of score) was played early - the preceding time interval was only 80% of what it should have been.

For convenience, here's a copy of the defstruct from BeamSearch:

```
(defstruct node
  (sind 0)          ;; The chord in the score that we are expecting
  (pind 0)          ;; The last performance note this node explains
  (new-chord)       ;; Flag - We are waiting for the first note in this
        chord
  (chord nil)       ;; The pitches in the current chord we haven't
        heard yet
  (bug nil)         ;; The last Bug needed to explain getting to this
        state
  (prob 1)          ;; The odds weight of this state (for comparison to
        others)
  (max-tempo most-positive-fixnum)
                    ;; The maximum (perf-tempo / score-tempo) we can
        believe
  (min-tempo 0)     ;; The minimum  " " " "
  (kids nil)        ;; Children nodes (alternative following bugs)
  (mom nil))        ;; Parent node.
```

The search processes each note of the performance in order, comparing it to what is expected in the score at this point. When a discrepancy is found, several alternative "bugs" must be hypothesized and these become children of the current node. Subjective probabilities are assigned to each hypothesized bug. The probabilities were determined intuitively and by experiment but their exact values probably don't matter very much.

When the current note matches expectations, the likelihood of the node is increased. A constant width "beam" of hypotheses are checked for each performance note and if necessary, new children created for each. The beam width is currently set to six, so the six most likely candidates are kept, the rest thrown away. When all the notes have been processed, the most likely node in the beam is assumed to be the true explanation of the performance, and all the

student's bugs can be retreived by tracing backwards from this node to the root. Note that if the student made no mistakes, the root node is the only one ever created and its probability is pegged to 1. Differing hypotheses may lead to the same prediction of where in the score the student is and these nodes cannot be distinguished further; therefore all but the most likely of these alternates are discarded at each iteration.

Pitch discrepancies are easy to define: The performed note is simply not one of the notes we are expecting in the current score chord.

Rhythmic judgments are more fun. There's probably a simpler way to explain this (and code it), but here's what I did. After experimenting on myself, I adopted this model of human rhythm: There is some ideal "true" tempo, expressed in the code as a ratio to the score's tempo. But human frailty adds some Gaussian noise to rhythmic judgments. A longer note is the sum of a sequence of infinitesimal rhythmic judgments so, applying the central limit theorem, the standard deviation from the "true" time will grow proportionally to the square root of the length of the note. The musical judgment of correct rhythm will therefore be anything within an interval proportional to the square root of the length of the note. (Of course, first I noticed the square root relation, then I made up this rationale.) The best we can do in Piano Tutor is keep estimates of the upper and lower bounds on what the "true" ratio might be, assuming the performer is trying to play in strict, steady rhythm. (Piano Tutor does not deal with rubato!) The lower bound starts at zero and the upper bound starts at +infinity. Each rhythmic interval the performer plays is a sample from which we can compute a range within which the "true" tempo will fall. In a correct performance, the ranges will all overlap and we can keep adjusting the Min upward and the Max downward to converge on an accurate estimate of the performers "true" tempo. A rhythmic error is signaled whenever the new Max estimate falls below the old Min or vice-versa. The new estimates of min and max ratios are given by the pseudo-lisp:

```
(let*   ((dur (length of the performed time interval))
         (beats (length of the corresponding interval in the score))
         (root-term (* *rhythmic-sensitivity* (sqrt dur)))
         (new-min (/ (- dur root-term) beats))
         (new-max (/ (+ dur root-term) beats))) .... )
```

rhythmic-sensitivity is a magic constant whose value has crept upward as Piano Tutor testing has progressed.

The only other kind of primitive error found in the search is the SKIP - when the performer's attention shifts to a different point in the score out of the expected sequence. When any pitch or rhythm mistake is noticed, one hypothesis must always be that the student skipped and is actually somewhere else in the score. The possible locations for the destination of a skip are found by searching forward and backward from the last known location for chords that contained the unexplained note that the student played. This search stops when enough candidates are found to fill up the beam. Each location is considered a separate hypothesis with probability inversely proportional to the distance of the alleged skip. I have no rationale for that, it just seems about right. To save some time and beam width, a single "generic-skip" node is first created which accrues likelihood as more unexplained notes come in. Only when this generic skip has become the most likely candidate in the beam is it expanded recursively by looking for the possible destination chords in the score. This recursion is limited to one level to avoid exponential (and redundant) expansion. The results of expanding the generic skip back to the current note index are then folded into the beam and search continues.

The Error Tree

Beam-Search passes back a list of primitive errors that the student made. This list is fed into the root of the tree of Error Types and a simple traversal of the tree is carried out. Each Error Type has an attached Diagnosis Function which examines a list of more general errors and extracts the subset which could be instances of this Error Type. Clearly, when the list becomes empty, further subtree traversal is cut off. Each diagnosis function is free to examine the original score, the recorded performance and any attached notation indications (stored as association lists in the variable *note-features* - see the file ScoreDescriptors.lisp for a set of primitives to describe features in a score.)

The diagnosis function at the top of the articulation subtree is special in that it generates new error types. This is enabled only when no pitch or rhythm errors have been found, since its not at all clear how to judge articulations if you can't assume correct rhythms. See the global variables at the top of DiagnosisFunctions.lisp for the relevant articulation parameters - e.g. How short a staccato must be, how much a legato can be separated etc.

Each error type has an associated severity - an arbitrary number meant to represent the importance of correcting this type of error. The error to be remediated is chosen by multiplying this severity by the number of instances of this error that were found and picking the highest number. However, before the remediation is carried out, a short list of rules (IF-statements) is executed to test for special remediation conditions:

- withdrawal of lesson on too many errors.
- passing the lesson anyway if the errors don't seem serious.
- picking out a phrase to work on where the errors seem concentrated.

See the file NewRemediation.rules for these.

IV. C Code Interface for Piano Tutor

The preliminary work for the foreign function interface was done by Blair H. Evans. It was then modified by Robert L. Joseph into what it is today. The objective was to allow a person to use C code written in Lightspeed C in the Allegro Lisp environment.

To use the code interface you should be familiar with:

- Low level System Interface (In Allegro Manual)
- Pascal Records (In Allegro Manual)
- Macintosh Memory Management (Inside Macintosh)

The code allows you to pass mac pointers in a record to C routines. The Lisp and C records are identically defined. The record has an integer number for desired function, a Pascal function definition used for printing to a the lisp listener window, then the arguments that a function might need. Currently we pass at most 3 arguments.

```
(defrecord (TestRec :pointer)
  (function integer)
  (pfun pointer )
  (arg1 pointer )
  (arg2 pointer)
  (arg3 pointer))
```

The actual C code is stored in a file as a code resource. When an external function is call via (ext-code <testrec> <file-name>) a code resource is loaded into memory from the <file-name> given. The code is locked so the system can not purge the code. Then the code resource is called with the <testrec> that was passed. The ext-code function will return a ptr back. It is up to the user to interpret the pointer information depending on what function was called. In the C routines a case statement on the function field determines the function being executed.

To program C routines you should know about:

- FF-Call Stuff (Supplement notes to Allegro User's Manual)
- Lightspeed C Code Resource Stuff (Lightspeed C user's Manual)

To write C code find externalfunction routine. This is the routine that sets up the print function and dispatches the args to the right functions specified by the function number. An example of external codes can be seen in projects Res EC Patch I and II. Currently the functions that they support are described below with a list of things that need to be done.

High Level Functions Done:

- (read-score <file-name>) -> returns mac pointer to score
- (display-score <adagio-pointer>) -> plays the adagio file
- (play-score <adagio-pointer>) -> plays the adagio score
- (record-score <accomp>) -> records an adagio score that a person has played
- (destroy-score <adagio-pointer>) -> destroys an adagio score
- (eval-score <perf-pointer> <score-pointer>) -> returns an evaluation of the performance vs. score. Returns a lisp data type should.

- (print-ev <lisp-eval-structure>) -> Prints out the lisp structure of the evaluation functions.

Functions To Do:

- (write-score <adagio-pointer> <file-name>) -> write the adagio score to a file
- (destroy-eval-pointer <eval-pointer>) -> goes in eval-score to destroy the ext-code for the C evalrec. Have to destroy Memory in eval-score from what is returned from the C stuff.
- (ac_play <accomp> <score> <record-flag>) -> Put this in to replace record/play

IV.1. Rule Interpreter

Rules are used for lesson selection, remediation, and analysis. They are represented by a frame in framekit. The rule has the form:

```
(rule1
        (if      (<if-clause1>
                 ...)
        (then   (<then-clause1>)
                 ...)
        (else   (<else-clause1>)
                 ...)
)
```

The else-clause is optional. The if then and else parts of a rule are lisp code that evaluates to something. A rules then clauses are executed when all the rules if-clauses return non nil. A rules else clauses are executed when at least one of a rules if-clauses returns nil. The if clause can have an expression of the form (Any identifier In list). When this form of expression is encountered, the remainder of the rule is evaluated with identifier bound to each value in list. This provides much power of predicates.

(run-rule <rule-name>) => Will run a rule and returns the last clause of the then or else or nil if it didn't fire.

(do-rule-list <list-of-rules>) => Is a function that runs through the list of rules in sequential order by default until no rule firers or a rule returns 'quit. Currently rule execution is defined by what the rules returns. The list and response is below: nil -> continues to the next rule continue -> continues to the next rule restart -> begins evaluating the rules from the beginning of the list. quit -> quits the rule execution

Rule-Trace => Variable that turns rule tracing on and off.

IV.2. Basic System Routines

This is the frame with the knowledge that the system teachers and the knowledge in which the student is expected to learn

(skill-instance (students-level (value students-mastery-level)) (used-by (value lesson-with-entry-as-prerequisite)) (taught-by (value lesson-with-entry-as-objective)))

(get-values 'skill 'instance-of) -> will return a list of all the skills

(get-skill-obj <skill>) -> will return all the objective lessons for that skill

(get-skill-pre <skill>) -> will return all the prerequisite lessons for that skill

(update-student-level <skill> <level>) - this will update the student skill level for the appropriate skill or flag wrong input error

highest-score is the max score the student can learn a skill

IV.3. Lesson Description

(lesson <lesson-name> { attribute-pairs}) the obj and pre slots are stored as (skill num) pairs: num is the level for the student to obtain to be considered satisfying the pre or it is the amount that the lesson hopes to teach.

the attribute-pairs for the pre and obj slots can be either single skill names or a pair of skill name and level achieve. if no number is given then the *highest-score* is put in the right place.

the lesson frame automatically puts the information for the skills in the correct place with the help of demons. No skill frames need to be explicitly created.

IV.4. Utilities

(print-skills) - prints out all the skills with pre and post lessons

(print-skill <skill>) - prints out <skill> along with pre and post lesson information

(print-lessons) - prints out all the lessons the system knows about

IV.5. Graphs

(piano-graph 'lesson &optional (shrink nil)) - creates a graph of the lesson/skill tree. Shrink allows you to put all information on one page

IV.6. Lesson Interface

Currently a lessons when a lesson is executed the presentation is done, then the task, then the remediation and then the analysis. The variable *evaluation* is set to how the student performed from the last task done in the current lesson.

The presentation is presenting a presentation frame to the user. This will need to be discussed more.

The task is a slot that request the student to do something. The student can be asked to just listen to a performance or it could be answer some question like a test. The task slot is a list of things describing what is required of the student. The following list is the beginning of what I could think of.

play nil Play and record performance. Evaluation is against self

play score Play and record performance. Evaluation is against score

play score accomp Play and record performance with accompaniment. evaluation is against score.

form Student fills a form of information in. I am not sure what the evaluation will be like

Remediation takes the rules in the remediation slot and runs them with do-rule-list.

IV.7. New Function

(repeat-task) -> repeats the previous task for the lesson and sets *evaluation* to the results.

IV.8. New Variables

performance -> The current performance of the student.

fired-rules -> list of all the rules that were fired in remediation and what they returned.

IV.9. Files and Their Uses

lesson structure.lisp => Basic frame structures for the piano system (lessons, sublessons, rules, skills, concepts)

lesson graphics.lisp => The graphics for the piano tutor system that uses framegraphics stuff.

lesson engine.lisp => The inference engine that runs one lessons.

fklessondb.lisp => The analysis routines that Roger did

rule engine.lisp => The rule inference engine

Adagio Ext Fun.lisp => The lisp code for the external code interface

CCL ext fun1.ffc- => The code resource that contains the functions from Res EC Patch I

CCL ext fun2.ffc- => The code resource that contains the functions from Res EC Patch II

IV.10. Directories

Framekit => Directory that has the framekit system in it.

Generic Graphics => Directory that has the framegraphics section in it.

Tree Graphics => The directory that has the tree drawing graphics of Joe Bates.

V. Score File Formats

The input files for the accompaniment program have a semi-rigid, semi-human-readable form. A score is represented by three files:

- Solo Score
- Accompaniment Score
- Graphical Score

This differs from JM's implementation in that there are separate files for each section of data, and there is no file for performance parameters, which are now held on the Lisp side of the world for convenient editing.

The solo and accompaniment score sections are in the Adagio music description language. The graphical score is essentially a block of signed numbers represented in ASCII but otherwise not easily decipherable. Graphical scores are produced using the module "ScoreConverter.c" along with a C subroutine which defines the score; the output of this process should be copied verbatim into the accompaniment score file. In other words, scores are produced by writing programs which output the encoded score. More details are given below (I'll also look for an example and more documentation to include in an appendix -RBD).

Just for the record, the performance parameters are listed below. The parameters define default values for tempo, volume levels for the solo and accompaniment, anticipation, acceleration, grace time, and the midi channel and program numbers for the solo and each accompaniment part. The parameters are as follows:

<acceleration (max percent rate change times 10)>
<anticipation (centiseconds)>
<grace time (seconds)>
<tempo (beats per minute)>
<solo volume (0-10)>
<accompaniment volume (0-10)>
<voice count>
  for each voice in voice count:
    <voice name (an ascii string on a line by itself)>
    <voice state (0 - off, 1 - Mac internal sound, 2 - midi)>
    <voice midi channel (1-16)>
    <voice midi program number (1-256)>
    <voice volume (0-10)>

Except for voice names, which are strings, these parameters are signed integers. A zero volume is equivalent to off. For example, the acceleration could be 150 (i.e. 15 percent of the current clock rate), the anticipation could be 8 centiseconds, the grace time 20 seconds, the tempo 76 beats per minute, and both the solo and accompaniment volumes could be 6. There might be three voices total, the solo voice and two accompaniment voices. The solo voice could be played at a volume of 8. The two accompaniment voices might be called "piano" and "flute" and are to be played on midi channels 2 and 3, respectively, using midi program numbers 1 and 17, respectively. The piano volume could be set to 5 and the flute volume set to 10 to get a good balance between the two instruments. (The voice volumes are used to achieve balance within the accompaniment while the overall solo and accompaniment volume control are used to bring out or back the solo with respect to the accompaniment.)

The solo and accompaniment files are Adagio descriptions of the the solo and accompaniment performances. The Adagio voice command should be used to make the solo part voice one and the accompaniment parts voices two through as many voices as there are separate accompaniment parts. The assignment of the separate melodic lines of the accompaniment to voices is a matter of judgement. Each voice may be assigned a different midi channel and program number and its dynamic balance with the other voices may be individually set. If one had both violin and 'cello parts in the accompaniment, it might be reasonable to assign these to the same voice (call it the "strings" part) but only if the same synthesizer program number could be used to produce both violin and 'cello sounds. There is a one-to-one correspondence between the Adagio voice numbers used in the solo and accompaniment descriptions and the voices named when parameters are specified parameters section. The parameters assign the MIDI channel, program numbers, and volume level for each part. (NOTE: The Adagio program change command, "z", is ignored.) Finally, both the solo and the accompaniment sections should have a !tempo Adagio directive. The specified tempos should match and should be identical to the tempo specified in the parameters section. The following example may clarify the previous discussion:

```
*SOLO
!tempo 76
* solo part (Adagio voice 1, set on following line)
t0 r n0 v1
r w2
c4 q; d; e; c
...


*ACCOMPANIMENT
!tempo 76
* accomp part 1 (Adagio voice 2)
t0 r n0 v2
c3 q; d; e; c
c3 q; d; e; c
...
* accomp part 2 (Adagio voice 3)
t0 r n0 v3
r w4
c5 q; d; e; c
c5 q; d; e; c
...
```

The graphics section defines the graphical objects which constitute the score display. This section is prepared by running a program and pasting the output of this program verbatim into the accompaniment score file.

The user describes the graphical score with a subroutine which calls operations in the ScoreConverter.h interface to add various graphical objects (staffs, clefs, notes, rests, barlines, etc.) to the score. After the entire score is described, the user's subroutine causes it to be dumped in text form into a file. (See any of the examples in the "Demo" directory to see how this is done.) The file containing this subroutine is added to the accompaniment project and a call to it

is placed in the "main" procedure (in AccompMain.c). The file ScoreConverter.c is also added to the project. When the project is run, the graphics score will be computed and dumped into a file as part of the startup process. The user can then "Quit" and proceed with the production of his new accompaniment score file. (And people still ask me to explain what I don't like about the Mac! -RBD).

V.1. Graphics File Format

This section was derived from the file `note.h`, which defines elements of graphical scores. The data is represented as a sequence of ascii decimal signed numbers separated by white space (blanks, tabs(?), or newlines).

Coordinates. All object locations are given as offsets from the bottom left corner of their staff. The x coordinate increases to the right and the y coordinate increases in the upward direction (backwards from the usual Macintosh convention).

```
          |--------------
          |--------------
          |--------------
          |--------------
          X--------------
   (0,0)  -> increasing x
          ^ increasing y
```

WARNING: There appears to be a design flaw in the way codes were allocated to simple symbols (character indexes in the music font) and more complex objects. In particular, the "string" object (which is unimplemented) has the same code as the BASECLEF object. These codes should be shuffled around to prevent such conflicts. However, doing so will invalidate previously produced score files so watch out! [Since this has no effect on the system as it stands (and I'm not to blame for the original design goof) I'm leaving it as it stands. -- John Maloney]

Simple Symbols.

Simple Symbol Code, x, y

x and y are the symbol location relative to the bottom left corner of the symbol's staff. Constants are used to index into the music font for the desired symbol. (See note.h). The following are more complex objects than the preceding simple symbols.

Strings.

String Code, x, y <string>

x and y are the string location relative to clef staff. Characters are packed two to a long. If first character is 'then the next character is a style and next is the font size. Otherwise, the string is printed in the default style.

NOT IMPLEMENTED!

Rectangle.

Rectangle Code, x, y, width, height

x and y are the bottom left corner of the rectangle. Width and height are positive in the positive direction.

Note Box.

Note Box Code, x, y, width, height

Defines the enclosing box for a compound event (all the symbols for a note or chord). x and y are the bottom left corner of the box. Width and height are positive in the positive direction.

End Note.

End Note Code

Marks the end of a note cluster.

IMPLEMENTED, BUT NOT NECESSARY!

End Staff.

End Staff Code

Marks the end of a staff.

Crescendo.

Cres Code, x, y, height, length

x and y are the top location of the crescendo or decrescendo. height and width specify the dimensions of a bounding box for the crescendo in positive direction.

WARNING: The next four elements are tagged with numbers which are unlikely to be the parameters to other commands in order to efficiently recognize them when we are reading in the score file.

THIS IS A MAJOR (AND POSSIBLY DANGEROUS) HACK!!!

Phrase.

Phrase Code, x1, y1, x2, y2

x1 and y1 are the start location of the phrase. x2 and y2 are the end location of the phrase. Uses the standard spline formula.

Tie. Tie Code, x1, y1, x2, y2

x1 and y1 are the start location of the tie. x2 and y2 are the end location of the tie. Uses the standard spline formula.

Array Representation. Here is an outline of the internal system array representation:

```
  start                                                                  start
  system                                                                 staff
  array                                                                  info
    V                                                                      V
-------------------------------------------------------------------------------
|  # of  | length | staff | staff |      | staff | staff | staff |      |  next
| staffs |  of    |   1   |   2   | ...  | rela- | rela- | info  | ...  | staff's
|   in   | system | loca- | loca- |      | tive  | tive  |       |      |  info
| system | braket | tion  | tion  |      | x val | y val |       |      |
-------------------------------------------------------------------------------
    ^        ^        ^
STAFFNUM     |     STAFFLOC
           BRACKETLENGTH
```

Table of Contents

1. Functional Description 1
2. Inventory 3
3. Equipment Needed for Prototype System 3
4. Capabilities: Prototype, Product, and Future 3
5. Relationships among MusicTutor, Computer Accompaniment, and PracticeMate 4
6. Time and Cost Estimates 4
7. Other Considerations 5

I. A Computer-Based Multi-Media Tutor for Beginning Piano Students 6

II. Report to Markle Foundation 1

8. Introduction 1
9. Accomplishments 2
10. The Human-Computer Interface 3
11. Instructional Design 3
12. The Expert System 4
13. Results 5
14. The Future 6
15. Conclusion 7

III. Lesson Selection Design Notes 8

IV. C Code Interface for Piano Tutor 12

- IV.1. Rule Interpreter 13
- IV.2. Basic System Routines 13
- IV.3. Lesson Description 14
- IV.4. Utilities 14
- IV.5. Graphs 14
- IV.6. Lesson Interface 14
- IV.7. New Function 15
- IV.8. New Variables 15
- IV.9. Files and Their Uses 15
- IV.10. Directories 15

V. Score File Formats 16

- V.1. Graphics File Format 18