1992

Computer improvisation of jazz solos

Daniel Chen

Rochester Institute of Technology
School of Computer Science and Technology

Computer Improvisation of Jazz Solos

by
Daniel C Chen

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

_____
John A. Biles

_____
Warren Carithers

_____
Steve Kurtz

July 13, 1992

Title of Thesis: Computer Improvisation of Jazz Solos

I Daniel C Chen hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Daniel C. Chen 7/13/92

This thesis discusses the possibilities of using a computer to create Jazz solos. Various implementations using stochastic and rule based approaches were created and applied to analyze the original melody as well as the chord progressions. Based on the melodies generated, a rule-based approach that considered the original melody and the chord progression was found to produce the most "musical" improvisations.

CONTENTS

1. Introduction & Overview

The original idea to write a thesis to deal with music originated with a class assignment dealing with the role of a knowledge base engineer. The job of the knowledge base engineer is to query an expert for information to gather enough basic rules and facts to create an expert knowledge base. The knowledge base in this case dealt with "improvising" jazz music. During the querying process the expert gave the opinion that he felt it was very unlikely a computer could ever produce any kind of music. This opinion on computer composed music became the basis of this thesis, an attempt to program a computer to "improvise" music.

Richard Kram has an interesting belief that "Musicians make good Programmers" [Kram85]. He goes on to hypothesize that this is due to the fact that both music and computer programs are normally linear processes. A misplaced note could ruin a musical phrase, and a misplaced computer instruction can make a block of code useless. Both composers and programmers must deal with measurements subjectively. When does a piece of music become boring or redundant? When does a computer program become unstructured or inefficient? If both composers and programmers must find and correct errors in their works, can a programmer build a program to write code?

Many interesting questions and issues arise when dealing with computer composed music. A group of notes organized randomly may be considered noisy and chaotic, not musical. A group of notes organized in a simple repetitive pattern may be considered too boring and also not musical. Would something between a random and a constant pattern be considered music? A computer is well suited for reproducing patterns, and pseudo-random behavior can be simulated with a computer, too. But can a computer compose good music? The quality of music is based on opinion, but what determines the difference between whether or not a

series of notes qualifies as music? This thesis will not attempt to answer this problem, but it will make an attempt at producing music that listeners can judge for themselves.

This is a thesis that deals with the creation of Jazz solos through the use of a computer. The software for the thesis was written in Prolog with a knowledge base containing elements of basic music theory with rules to develop a solo. Part of the background section will describe other systems that have created music through the use of computers, and the next section describes in more detail several systems that used computers to compose music. The final background section will describe basic assumptions that the thesis makes. Section 3, on the implementation, describes in more detail the structures of the code and it's interactions. Section 4 will contain the results. The result section contains a comparison of several improvised songs. A step by step example of one song is also contained in the results section. The final section, section 5, is the conclusions. In this section, a description of the overall successfulness of the thesis, and how it could be improved upon.

2.0 Background - Approaches to Computer Composed Music.

This document was written with the assumption that the reader has some basic knowledge of music theory and the Prolog computer language. A brief description of music theory is presented in Appendix A, and a more in-depth view of music theory can be found in a standard music theory book [Benw83].

Section 2.1 will briefly describe several historical approaches to computer composed music. A more detailed description of systems that use a stochastic approach is provided in Section 2.2. An interesting method of using binary trees to create a simple melody is described in Section 2.3. Section 2.4 describes a thesis similar to this one, which attempted to use a computer for jazz improvisation. The final section 2.5 is the conclusion for this background.

2.1 Historical Approaches

This section will briefly describe several previous attempts at using computers to create a melody. It is included to give the reader a "flavor" of what has been accomplished. Many music scholars have tried to use computers as an aid to music theory. This research has been dominated by statistical techniques. Some count the number of B flats or the number of times a major third occurs in order to figure out a musical style [Road85]. Such a statistical profile, combined with common patterns in other pieces, can be used as a rather limited method for constructing music.

Ebcioglu [Ebci84] developed a knowledge-based expert system to generate chorales in the style of J. S. Bach. The chorale program is written in BSL (Backtracking Specification Language), and is based on a direct compilation of a formula taken from first order predicate calculus. Currently the program only harmonizes an existing chorale melody. The knowledge base contains 190 rules and heuristics, which were found mostly by empirical observation of the chorales and by

personal intuitions. The system generates chorales from left to right, backtracking until a solution that satisfies all the constraints is found.

Another early attempt at music composition was described by Olson-Belar [Hill70]. Their composing machine used signal generators to generate random numbers, which were fed into devices that assigned weighted probabilities to control the pitch and rhythm of the composition being generated. Inputs for the weights were taken from 11 Stephen Foster tunes. These tunes were transposed to D major so that the probability sample of pitches covered a greater range of pitches. First, second, and third order frequency counts of pitches were gathered and applied to the weighting. Olson and Belar actually reproduced one of the original eleven Stephen Foster tunes with this device, which demonstrates that the basic music structure can be generated based upon probability.

Brooks, Hopkins, Neumann, and Wright [Hill70] provided a more substantial work on the applicability of stochastic models in both the analysis and synthesis of simple music. The authors statistically analyzed 37 hymn tunes up to an eighth order approximation. The tunes were constrained such that they all had to be in C major and in a common meter. The various transition or interval frequencies were used in tables for transition probabilities for creating new tunes. The probability of a transition depended on the transition before the current transition. Random integers were generated and screened depending on how they fit against the probabilities. A "try-again routine" was used to rewrite unacceptable passages. Their results showed that if the order of synthesis is too low, the note sequences were not typical of the sample analyzed. If the order of synthesis was too high, duplication of the original sample would occur. A point in between the two extremes of synthesis resulted in tunes that were recognizable members of the class from which the sample was drawn from. The best level of synthesis seemed to be around sixth order, which agreed with the work carried out by Baker [Bake63].

Baker analyzed selected musical passages from Haydn, Mozart, and Beethoven, and found that lower orders of analysis and synthesis may be sufficient if a simple enough stochastic model is used.

A survey reported by Hiller describes a generative approach to music[Hill70]. He describes three goals that a program must have to generate music: a scientific verification of a music theory (simulating a known style); producing an object of aesthetic interest (original composition); and recreational value (colloquially referred to as fun). Style has been approached by gathering statistical data on a specific composer. Original composition has been attained through the use of random number tables. I feel the recreational value is attained through completing the task and demonstrating it to others. Most of the work in computer composed music seems to be based on a gathering data on a specific style and applying a stochastic approximation of different orders of approximation. The methods described in the next sections also use the stochastic process to some degree, but also include variations of different structures or theories.

2.2 Stochastic Approach

This section will describe in more detail several systems that use a stochastic approach. The first method described will be an approach using stochastic approximation. Following that, will be several methods that use structures to further constrain the stochastic approach.

As stated earlier, music may be created by randomly generating a list of notes. Mozart supposedly did this using dice[Bate80]. One proposed method described by Bateman [Bate80] first generates random numbers from -12 to 12, which represented intervals between successive notes. A series of notes is then generated from the intervals. A table was created that weighted the desirability of different intervals, where short and consonant intervals were given a heavier weight than longer intervals. The melodies generated with this process were not be very interesting to most people.

Instead of weighting intervals on how short or long they are, one could base the weighting on a more desirable interval having a heavier weighting, and a less

Interval Number	Relationship	Probability of Selection	Interval Number	Relationship	Probability of Selection
-12	- Octave	3%	0	Unison	4%
-11	- Major seventh	1%	+1	+ Half step	7%
-10	- Minor seventh	2%	+2	+ Whole step	6%
-9	- Major sixth	3%	+3	+ Minor third	5%
-8	- Minor sixth	3%	+4	+ Major third	4%
-7	- Perfect fifth	7%	+5	+ Perfect fourth	6%
-6	- Tritone	1%	+6	+ Tritone	1%
-5	- Perfect fourth	6%	+7	+ Perfect fifth	7%
-4	- Major third	4%	+8	+ Minor sixth	3%
-3	- Minor third	5%	+9	+ Major sixth	3%
-2	- Whole step	6%	+10	+ Minor seventh	2%
-1	- Half step	7%	+11	+ Major seventh	1%
0	Unison	4%	+12	+ Octave	3%

Figure 1 Example of probability distribution function

desirable interval having less weight. Figure 1 shows one such possible interval weighting. Intervals of a minor second and fifth are more desirable so they are given

a greater weight than a less desirable interval of a tritone or a major seventh. This method of constraining the note selection may produce a better melody, but it still needs more structure.

Selecting notes by this method is termed "first order" because the note selected is dependent on only one parameter (the preceding note.) A second order system can be used to create more structure and further constrain a random note pattern. There are many ways a second order system can be constructed. The note could be selected by the preceding two notes; or as Bateman suggests, the interval determines a set of 10 possible functions that are used to determine the next note. For example, a tritone in traditional music is usually followed by a stepwise interval. The stepwise interval then will be given a higher rating for that next note. This process of selecting notes can be considered a stochastic process and can be continued to a third order, fourth order, etc. process. To make it more interesting, exceptions could be incorporated. For example, no more than three consecutive leaps or no more than four consecutive alternating ascending and descending intervals could occur in a row. Another possibility is to incorporate a small routine that would select the next set of intervals.

Another method of viewing this would be to use a Markov Chain as described by Jones [Jone81]. Jones defines a stochastic process as a collection of random variable quantities distributed in space or time. Computer music systems usually require specific information of all the parameters relating to a sound. This may be much more information than what is in a musical score. Stochastic techniques offer a useful means of data reduction. Defining parameter limits on the actual values that are generated stochastically can significantly reduce the amount of work required. Composer usually rely on a performers' interpretations for the fine control of parameters like intonation, duration, timbre, and intensity. Stochastic techniques also produce unanticipated possibilities that music theory may restrict, which allows

one to break the limits of imagination. Jones used a random decaying function to generate random numbers. In composing Firelake, a random integer generator was used to call itself recursively in the form of RAND(RAND(N)). The function RAND(N) returned a random integer from 1 to N, so when the function is called recursively, the whole function will produce numbers weighted toward 1. When these numbers are applied to events, event e1 will occur most often, while, event e2 will occur less frequently. If the events were pitches, it allows the pitches generated to be built around a quasi-tonal center. Using this method of generating numbers, Jones combined this using Markov chains as acontrolling mechanism. This is similar to the method that Bateman suggests, but has more of a structure than the first order note selection described in the previous section.

A Markov chain takes into account the context of an event in a sequence and makes the probability of its occurrence depended on the event that preceded it. Basic properties of Markov chain events are commutative, reflexive, symmetric, transitive, recurrent, and transient. A recurrent event is one that may happen again after it has occurred. A transient event is one that will not recur. Figure 2 shows a stochastic matrix of a Markov chain of order eight. Event e2 has a 100% chance of occurring if the previous event was e1, but if the previous event was e2, there is only a 50% chance of event e2 occurring.

		Next Events							
		e1	**e2**	**e3**	**e4**	**e5**	**e6**	**e7**	**e8**
Current	**e1**	0	1.0	0	0	0	0	0	0
Events	**e2**	0.5	0.5	0	0	0	0	0	0
	e3	0.1	0.1	0.4	0.4	0	0	0	0
	e4	0	0.2	0	0	0.8	0	0	0
	e5	0	0	0.5	0.5	0	0	0	0
	e6	0.1	0	0.1	0	0.1	0.7	0	0
	e7	0	0	0	0	0.4	0	0.3	0.3
	e8	0	0.2	0	0.2	0	0	0.6	0

Figure 2
Markov chain events

An event relation diagram of the matrix in Figure 2 is shown in Figure 3 to

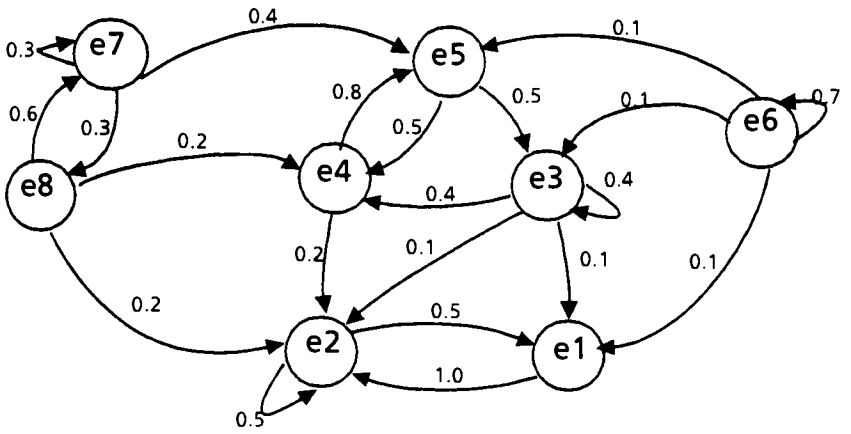


Figure 3 Event Relation Diagram

help visualize the structure. In this diagram, event e2 is shown as having two arrows exiting from event e2. The arrow to e1 has a 50% chance of occurring and the arrow back to itself also has a 50% chance of occurring. Each event is then associated with a note sequence. Figure 4 shows three possible event sequences that might be generated from the matrix. Sequence S1 starts with event e6. Event e6 shows the possibility of events e1, e3, e5, or e6 of occurring. In this case event e6 occurred first, from the second event e6, event e3 occurred. From event e3, events e1, e2, e3, or e4

S1 = e6 e6 e3 e3 e4 e5 e5 e4 e2 e2 e1 e2 e1 e2 e1 e2 e2 e2
S2 = e6 e6 e6 e6 e1 e2 e1 e2 e1 e2 e2 e1 e1 e2 e1 e2 e1 e2 e2
S3 = e8 e4 e2 e2 e1 e2 e1 e2 e2 e2 e2 e1 e2 e2 e1 e2 e1 e2

Figure 4 Several Possible Event Sequences

could occur. The Figure 5 shows how an event may be associated with a note sequence. Figure 6 then shows how the first few events in sequence S1 would look like as an actual note sequence. A Markov chain structure may be used upon another to increase the order of the Markov system. So the events e1-e8 could be sub events of an even bigger Markov chain big event En. Big event En would progress to another big event Em after some number of sub events occurred within

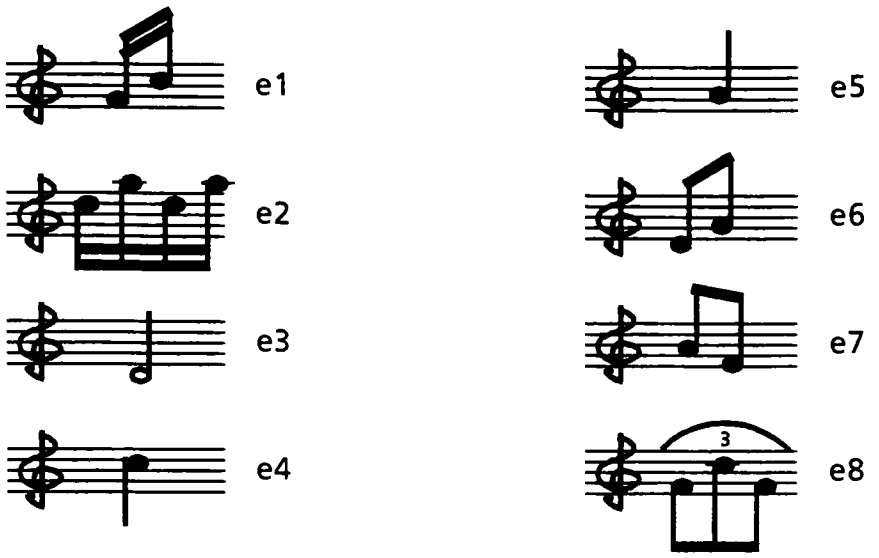


Figure 5 Possible Note Sequences for Events

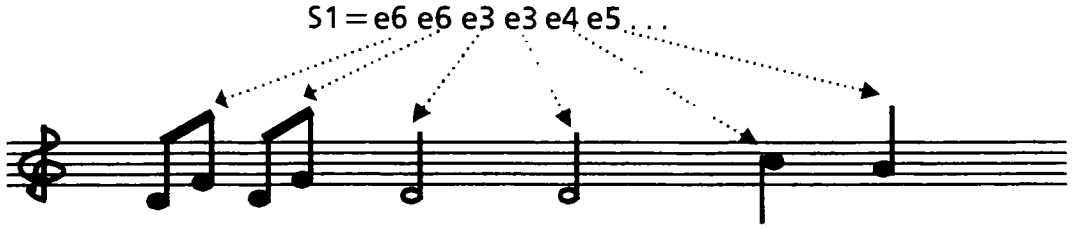


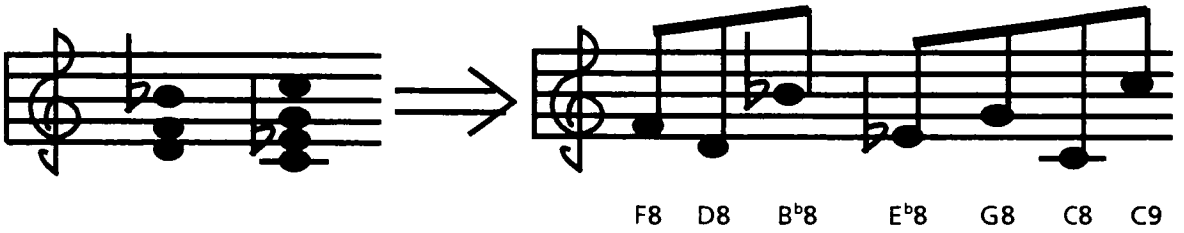
Figure 6 Partial expansion of sequence S1

big event En, or a sub event e9 could be created which proceeds to some big event that follows big event En. This would in turn change events from scalar values to vectors and eventually to a finite state grammar. Markov chains maintain a rigid structure, but still use a stochastic approach. The next section will describe another method of note pattern generation that is structured, but is in a sense less random.

2.3 An approach using a binary tree

Richard Kram [Kram85] describes an interesting method which uses binary trees. Kram's method uses a set of triads to generate a melodic pattern. First, the method of storing will be described; then a process of creating a melody will be discussed.

In Kram's example, two alternating triads a step apart are used (Fig 7a). The notes of both triads are strung out in a linear representation (Fig 7b). The labels

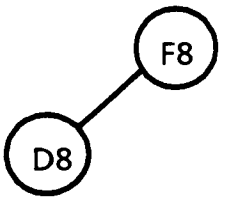


Two triads a step apart
Figure 7a

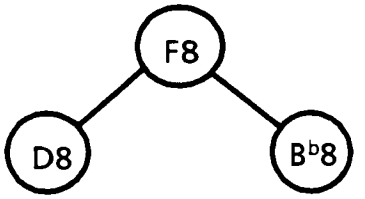
Triads separated into a sequence of notes
Figure 7b

below each note in figure 7b represent the note name and the octave, where an octave change occurs from the note B to the note C. The notes are then selected and placed into a binary tree, such that the left child is of lower pitch than the root of the

Root of tree
Figure 8a



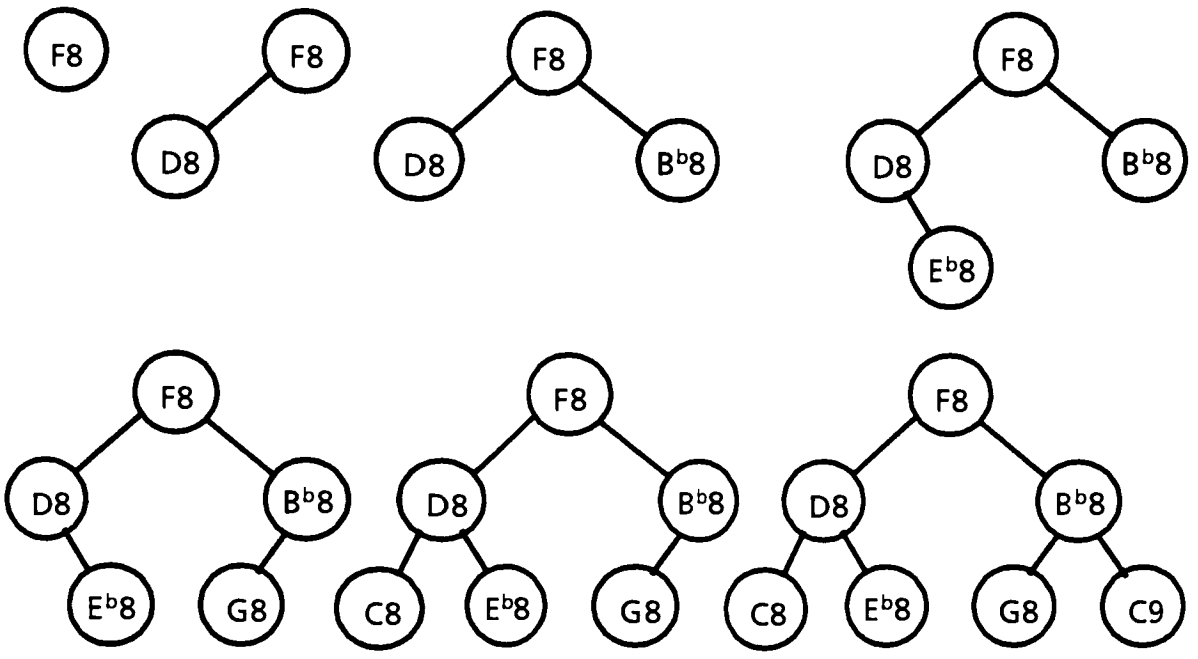
D8 Lower pitch than F8
Figure 8b



B♭8 Higher pitch than F8
Figure 8c

tree and the right child is a higher pitch than the root. In the example given, F8 the first note is used as the root of the tree (Figure 8a). The next note D8 is of lower pitch so it is placed in the left child of the tree (Figure 8b). The next note B♭8 is higher pitched so it is placed to the right of the root of F8 (Figure 8c). The complete

sequence and final tree is shown in Figure 9 There are limitations in this binary



Triads placed in a binary tree
Figure 9

tree representation of storage. One limitation mentioned by Kram is the lack of ability to store two notes of the same pitch. Another is that the original note ordering is lost.

The process of creating the melody is dependent on how one selects the tree traversal. The three most obvious methods would be either an inorder: left, node, right; (Fig 10a), preorder: node, left, right; (Fig 10b) or postorder: left, right, node traversal (Fig 10c) of the binary tree. Other methods of tree traversal may produce interesting patterns, too. Another way to traverse a tree is to unwind it. It is much easier to visualize this as first selecting the leaves of the tree and continuing up to the root as the leaves for that depth of the tree are used (Fig 11). A separate tree with a set of transposed notes could be used to create a harmonic variation. Since this thesis does not deal with harmony, other than what is present in the chords, the

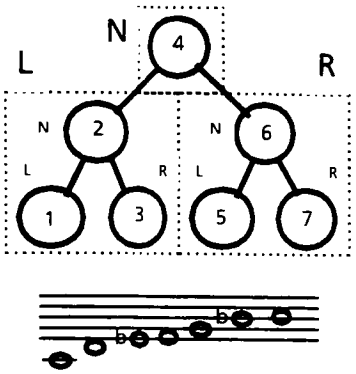


Fig 10a Inorder

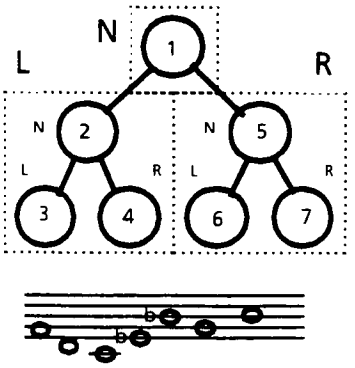


Fig 10b Preorder

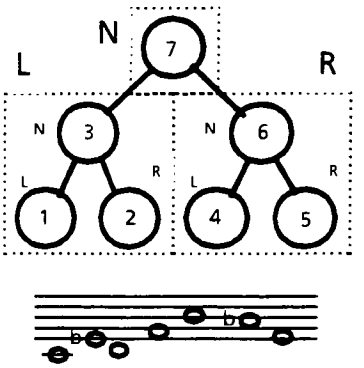


Fig 10c Postorder

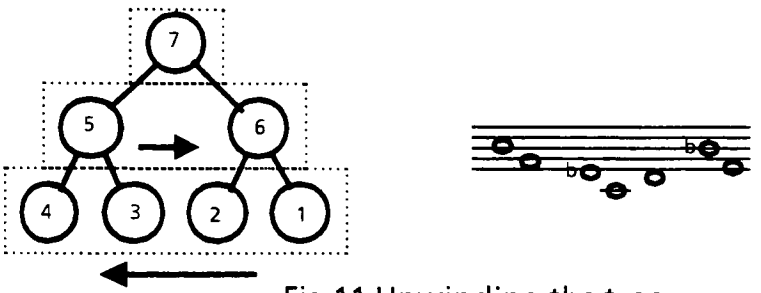


Fig 11 Unwinding the tree

harmonic variation will not be discussed. Another method that may reduce the modal sound would use notes that were not in the first tree to form a second tree. The notes would be selected alternately between both trees to generate the melodic pattern. The resultant set of notes may create an odd combination of both chromaticism and modality. More patterns may be created by channeling the output of the current tree into a new tree and traversing that tree. There are many additional ways one could use the basic tree structure to generate notes.

2.4 Computer Jazz Improvisation

It would be much more useful if the computer could make connections within the music, in a sense understand music structure and build on it. I feel David Levitt came very close to doing this. Levitt's masters thesis[Levi81] deals with Jazz Improvisation. The program can be broken down into two parts. The first is a simple analysis of the chord progression, and the second is the melody.

The chord progression is analyzed to determine a progression of modes which are used to determine consonance against which the melody is analyzed. The melody is divided into phrases and ranked on how well it can be used for improvisation material. The chords provide a framework to analyze the melody and produce improvisation. Each chord is analyzed to determine into which modes it fits. When a chord changes, a minimal motion criterion is applied. The minimal motion is defined such that the previous mode is retained unless the chord gives evidence that the mode has changed. This is noticed when just one of the tones in the chord is dissonant with the previous mode. This new tone is thought of as the disambiguator for the new mode and it will direct a transition to the nearest mode consonant with that chord. Figure 12 shows how a change in one note affects the current mode. If two or more tones differ from the previous mode or if the previously mentioned algorithm fails a new algorithm is used. The program detects this as a "discontinuous modulation" and the program uses a "typical progression" heuristic. A discontinuous modulation to an asymmetric chord is assumed to be a progression forming a new "well known chord progression". This would be known as something like the "2 5 1" chord progression.

After the chords are analyzed for their mode, the melody must be broken up into phrases. Levitt simply breaks up the melody at two measure boundaries for simplicity. Two-measure phrases were determined to be long enough to manifest

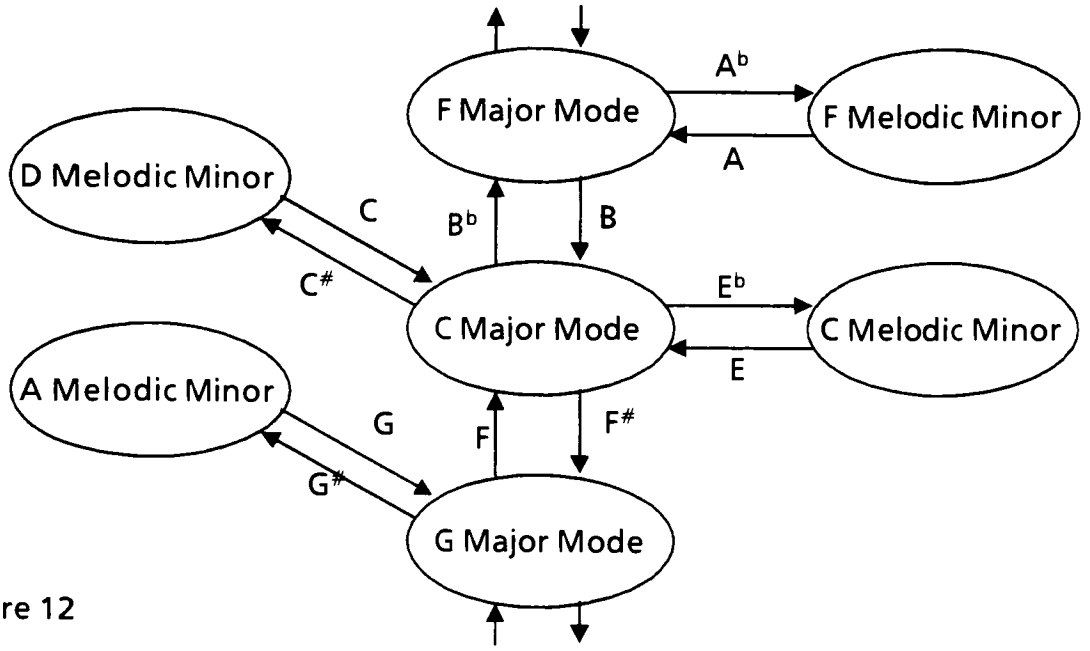


Figure 12

patterns that can be repeated. The last two measures of the initial melody are used to start the solo. After this phrase, a new melodic material will be generated. The variation the program applies to the theme is based on an algorithm that looks for a feature and then produces a given pattern. An outline of Levitt's variation-making scheme is shown in figure 13.

FEATURE	INHERITED PATTERN
Repeated Pitch	Contour
Leap > Fifth	Interval-size
Scalewise	chord-Degree(all Modal)
Unresolved Dissonance	Mode and Chord consonance patterns
Uniform features	Repeat uniform features
Else	Contour

Figure 13

The first feature can be thought of as if a repeated pitch is recognized, the repeated pitch pattern is inherited too. The second feature duplicates the interval, but not necessarily the direction of the interval. So if the interval is a minor seventh up, the inherited pattern can be either a minor seventh up or down. The scalewise

feature is described as a interval of a major second or less. In this case, the inherited pattern is a parallel motion with respect to the chord root. If an unresolved dissonance is present, the patterns of mode and chord consonance are inherited. Dissonances in music are apparent to listeners and provide a strong parallel framework. If all the previous features fail, the program seeks for general features whose imitation might keep the solo coherent. What the program looks for is pitches that are all chord consonant, intervals that are the same size or within a minor third, and a uniform ascending or descending interval over the whole phrase rather than a particular note. If one of these features is recognized the variation is constrained to inherit the same feature. If none of these are recognized, the original rhythm and general direction of the pitches is still inherited. The recognizers do not have to be found as the next section will further constrain the solution.

The previously described algorithm may not fully constrain a pitch. If it does not, another set of rules is applied to the note. The program next applies the following rules in the order shown (Fig 14). If a rule is applied and no options result,

Leaps larger than a Fifth (with respect to the neighbor)
Pitches close to the edge of the instrument range
The neighbor pitch itself (i.e. avoid repeated pitches)
Leaps larger than a Major 3rd
Pitches dissonant with the mode
The neighbors other neighbor, if known
Leaps larger than a Major 2nd
Pitches dissonant with the chord
Leaps larger than a minor 2nd.

Figure 14

that rule is ignored and the next is tried. When the option restriction has been performed, there still may be more than one pitch which satisfies the constraints. The program will then choose the pitch which is closest in pitch with the neighboring notes. If two pitches are just as close the lower one is chosen. Levitt mentions that

there is no justification for this. Levitt also mentions that this algorithm is not very complicated, and lacks noise sources, but it is motivated by theory.

2.5 Background conclusion

My thesis will deal with a program that will be based on one form of contemporary Western music, Jazz. The note relations will be based on a well-tempered 12-tone scale. Time will be in common (4/4) time, and a maximum of 2 chord changes per measure can occur. They will be on the 1st and 3rd beat of the measure. Harmony will not be dealt with, but provisions may be made to allow rhythm. The notes produced will all be of constant duration.

Inputs to the knowledge base such as chord progressions, desired note patterns, etc. will be required. A set of note patterns will be extracted from at least two of the following tunes: Now's The Time, Yardbird Suite, Don't Get Around Much Any More, Lady Bird, All the Things You Are, Four, Blue Bossa, A Night in Tunisia, and Green Dolphin Street. The music then will be analyzed to form the details of the implementation. The solo will not be generated in real time. The knowledge base will not generate any kind of rhythm. Song length will be either twelve or thirty-two measures in length, which can be repeated as many times as necessary. The harmonic progression, which forms an outline of the composition, already exists as the chord progression taken from a standard fake book. The chords are limited to five chord types: major, minor, dominant, half diminished and diminished. The final output will be a list of notes. If time permits an attempt will be made to have the program output the notes so one can listen to the output.

3.0 Implementation

A general layout of the software is depicted in the following diagram (figure 16). It is broken down into 4 different levels: Level 1 being the lowest, and Level 4

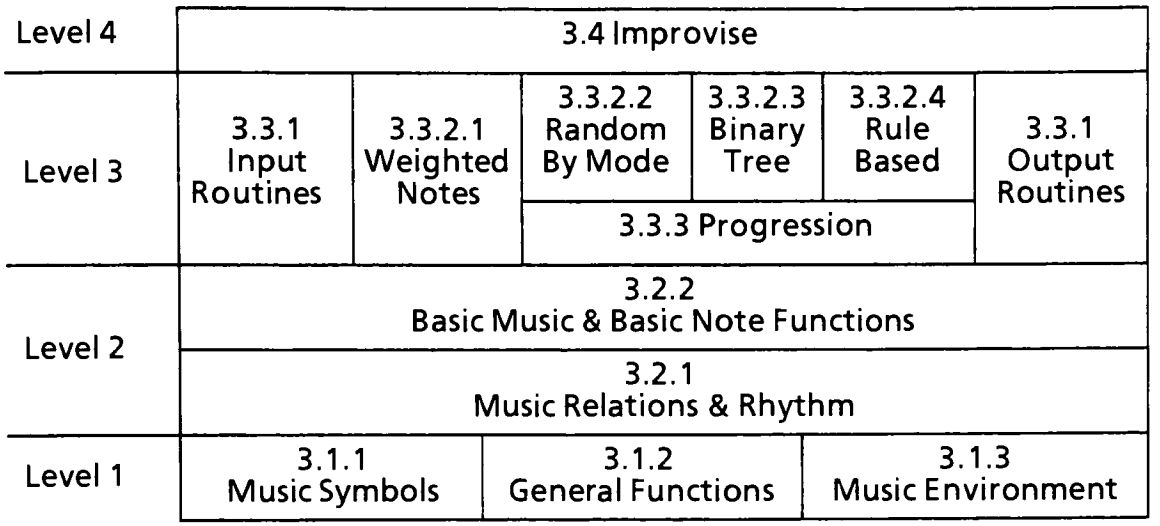


Figure 15 software layout

being the highest level. Within each level are one or more blocks, each of these blocks corresponds to one or more software modules. This chapter is divided into sections corresponding to these levels and blocks. In addition to these sections is a section which describes the user interactions. Each block depicted in the diagram is dependent on the block below it. So the blocks labeled Random By Mode, Binary Tree, and Rule Based are all dependent on the block labeled Progression, but the block labeled Output Routines does not depend on Progression. The numbers indicated in each block represent the section that describes that block. Terminology and verbiage used in this section is described in the glossary. In Prolog, symbols must begin with a lower case letter. Examples in the following section will follow this rule. Words beginning in uppercase represent variables, as this is how they are represented in Prolog.

3.1 Level 1 Low Level Functions

Level 1 corresponds to the lowest level functions. These are usually symbol definitions, generic prolog functions, constants, and other data that may effect the output, but does not affect the process. Level 1 is broken down into three sections, titled Music Symbols, General Functions, and Music Environment.

3.1.1 Music Symbols

The music symbols module determines how some of the data atoms are represented internally and how they are inputted. Listed below are a few of the symbols and a description.

note_symbols(*ExtNoteName*, *IntNoteName*, *NextIntNoteName*, *HSDist*)

Converts the input note name to the symbol used internally. Currently, each input note maps to the same internal symbol. *ExtNoteName* is the symbol the user would type from the keyboard. *IntNoteName* is the symbol that is represented internally. *NextIntNoteName* is the symbol that follows the *IntNoteName* in ascending or higher pitched order. *HSDist* is the distance in halfsteps between *IntNoteName* and *NextIntNoteName*. Adding a new type of note is as simple as adding another **note_symbols** rule. The order is dependent in the database. The first rule listed under **note_symbols** is assumed to be the start of a new octave.

rest_symbol(*ExternalSymbol*, *InternalSymbol*).
sharp_symbol(*ExternalSymbol*, *InternalSymbol*).
flat_symbol(*ExternalSymbol*, *InternalSymbol*).

Converts the external rest, sharp or flat symbol from what the user would type in on the keyboard to how it is represented internally. Currently the internal and external representations are the same.

number_of_note_names(*Num*)

Counts the number of defined note_symbols. Used to determine the number of notes in an octave.

convert(*ExternalChordSymbol, InternalChordSymbol*)

Converts *ExternalChordSymbol* or what the user typed in to an internal representation and returns it in *InternalChordSymbol*.

3.1.2 General Functions

This section will list rules that were created to support the music routines or were used to aid in the I/O.

add_item_to_each_elem(*List1, Instance, List2*)

Adds an *Instance* to each item in the *list1*. Backtracking this clause will fail.
Implementation file: GeneralFunctions.p
Example:
>add_item_to_each_elem([a,b,c],[1,2],NewList).
NewList = [[a, [1,2]], [b, [1,2]], [c, [1,2]]];
no

assign(*Item, Item*)

Is used to 'assign' a list or other data structure to an un-instantiated or unbound variable. Backtracking this clause will fail.
Implementation file:GeneralFunctions.p
Example:
?- assign([g,[f],1],GFlat).
GFlat = [g, [f],1];
no

generate_list_of(*Numeric, Atom, List*)

Generates a list *Atoms Numeric* times. Backtracking this clause will fail.
Example:
?- generate_list_of(3, f, List).
List = [f,f,f];
no

get_next(*List, Atom*)

Is used to return an item on a list. If backtracked to, the next item in the list will be returned. This will continue until there are no more items in the list.

Implementation file:GeneralFunctions.p

Example:

```
?- get_next([a,b,c],Next).
  Next = a;
  Next = b;
  Next = c;
  no
```

lengthen_list(*List1*, *Numeric*, *List2*)

Duplicates *List1 Numeric* times.

Implementation file: GeneralFunctions.p

Example:

```
?- lengthen_list([a,b,c],2, BigList).
    BigList = [a,b,c,a,b,c];
    no
```

make_random_list_from(*List1*, *Numeric*, *List2*).

Returns a *list2* of by picking items from *list1 Numeric* times.

Implementation file: GeneralFunctions.p

Example:

```
?- make_random_list_from([a,b,c], 6, RandomList).
    RandomList = [c, a, a, c, b, a];
    no
```

pop(*List1*, *Atom*, *List2*)

Returns the first *Atom* in the list and the resulting list without the first atom.

Implementation file: GeneralFunctions.p

Example:

```
?- pop([a,b,c],Item, List).
    Item = a, List = [b,c];
    no
```

pop_bottom(*List1*, *Atom*, *List2*)

Returns the last atom in the list and the resulting list without the last atom. *List1* must contain at least two atom.

Implementation file: GeneralFunctions.p

Example:

```
?- pop_bottom([a,b,c], Item, List).
   Item = c, List = [a,b];
   no
```

positive_random(*Numeric*)

Returns a positive random number.

Implementation file: GeneralFunctions.p

Example:

```
?- positive_random(Number).
 Number = 13242;
no
```

random(*Numeric1*, *Numeric2*, *Numeric3*)

Returns a random number between *Numeric1* and *Numeric2* inclusive.

Implementation file: GeneralFunctions.p

Example:

```
?- random(3, 5, Num).
   Num = 3;
   no
```

rotate_list(*List1*, *Numeric*, *List2*)

Rotates *List1 Numeric* times.

Implementation file: GeneralFunctions.p

Example:

```
?- rotate_list([a,b,c], 2, RotatedList).
   RotatedList = [c, a, b];
   no
```

split_list(*List1*, *Numeric*, *List2*, *List3*)

Splits *List1* at position *Numeric*.

Implementation file: GeneralFunctions.p

Example:

```
?- split_list([a,b,c,d,e], 2, Front, Back)
    Front = [a, b], Back = [c, d, e];
    no
```

traverse_tree(*TreeList*, *TreePath*, *Order*)

Traverses tree *TreeList* and returns each leaf visited in a list in *TreePath*. *TreeList* is a list of three items. The first is an atom. The remaining two must be lists. The traversal method is determined by *Order*. *Order* can be one of the following: postorder, revpostorder, inorder, revinorder, preorder, revpreorder

Implementation file: GeneralFunctions.p

Example:

```
?- traverse_tree([a,[b, nil, nil], [c, nil, nil]], List, inorder).
    List = [b, a, c];
    no
```

ungroup_list(*List1*, *List2*)

Removes a level of lists.

Implementation file: GeneralFunctions.p

Example:

```
?- ungroup_list([[a,b],[c,d]], Ungrouped).
    Ungrouped = [a,b,c,d];
    no
```

3.1.3 Music Environment

This module is used to define basic parameters and conversions of the system. This module also defines the external interface with a program written in C to play the actual pitch. A changeable configuration file is consulted within this module to allow changing of parameters like median pitch and unit of duration. Listed below are the rules created to convert the internal note format to one where the C resources may use.

convert_duration_to_internal(*Duration*, *NumericDuration*)

Converts the Internal representation of duration from a fraction to an absolute number. This conversion is dependent on rules beat_duration, and duration_to_beat which can be changed online by loading a new configuration file.

Example:

```
?- convert_duration_to_internal([1,2], Duration).
A = 600;
no
```

convert_note_to_pitch(*Note*, *NumericPitch*)

Converts the internal representation of a note into an absolute number. This conversion is dependent on pitch_offset which defines the starting point or lowest possible pitch. The output of this may also be changed online by loading a new configuration file.

Example:

```
?- convert_note_to_pitch([a, [], 5], Pitch).
P = 70;
no
```

play_note_internal(*Volume*, *Pitch*, *Duration*)

Plays the a pitch based on the input parameters. This is actually the interface to a resource written in C.

3.2 Level 2-Music Theory Functions

Level 2 modules are a collection of procedures which relate to music theory. This section is broken down into two parts. The first part deals with the simpler music theory and the second part deals with the more advanced parts of music theory.

3.2.1 Music Relations and Rhythm

Music Relations and Music Rhythm are the simpler rules used by the program. Most of MusicRelations rules are conversions or translations, while the rhythm file is just fraction arithmetic handling of duration.

The Music Relations module contains some basic associations between modes, intervals, scales, and chord types. The following is a partial list of these relationships.

interval(*NoteName1*, *NoteName2*, *HS*)

Returns the interval in halfsteps between the two note names.

intervals(*Interval*, *Hs*, *Degree*)

Gives the relationship between an interval name, the number of halfsteps, and the typical degree associated with that interval. The following table defines what is currently supported by the rule intervals.

modes(*Mode*)

Mode is a symbol which represents one of the possible modes defined in the database.

Example:

```
?- modes(Mode).
    Mode = ionian;
    Mode = dorian;
```

interval	halfsteps	degree
unison	0	0
mi2	1	1
ma2	2	1
mi3	3	2
ma3	4	2
p4	5	3
a4	6	3
d5	6	4
p5	7	4
mi6	8	5
ma6	9	5
mi7	10	6
ma7	11	6
p8, octave	12	7

Mode = locrianS2;
Mode = superLocrian;
no

mode_scale(*Mode*, *OffsetList*)

Returns a list of numbers representing the number of halfsteps from a root key defined by Mode.

Example:

?- mode_scale(ionian, OffsetList).
OffsetList = [0, 2, 4, 5, 7, 9, 11, 12]

mode_interval(*Mode*, *IntervalList*)

Returns a list of intervals in *IntervalList* representing the intervals between each of the notes in the scale defined by *Mode*.

Example:

?- mode_interval(ionian, IntervalList).
IntervalList = [ma2, ma2, mi2, ma2, ma2, ma2, mi2]

base_intervals_in_scale(*IntervalList*).
base_hm_intervals_in_scale(*IntervalList*).
base_mm_intervals_in_scale(*IntervalList*)

Defines the intervals for each of the base modes. Currently three based modes are defined. They are major, harmonic minor, and melodic minor. The *IntervalList* returned is a list of the intervals between the notes for a scale in that base mode.

chord_type_intervals(*ChordType*, *IntervalList*)

Returns the list of intervals in *IntervalList* that are used to compose the type of chord specified in *ChordType*. Listed below are the

Seven Chord Name	Internal Symbol	Intervals
Major 7	maseven	ma3, mi3, ma3
Dominate 7	seven	ma3, mi3, mi3
Minor 7	miseven	mi3, ma3, mi3
half diminished	hdim	mi3, mi3, ma3
diminished seven	dim	mi3, mi3, mi3
minor seven sharp 5	misevens5	mi3, ma3, ma3
minor seven sharp 7	misevens7	ma3, ma3, mi3

relationships that are defined for the seven chord types.

interval_sum(*Interval1*, *Interval2*, *HS*)
interval_sub(*Interval1*, *Interval2*, *HS*)

Returns the sum or the difference between the *Interval1* and *Interval2* in halfsteps in *HS*.

interval_total(*IntervalList*, *Total)*

Sums the intervals in *IntervalList* and returns the total number of halfsteps in *Total*.

Example:

?- interval_total([mi2, ma2], Total).

```
Total = 3;
no
```

The MusicRhythm.p module allows various calculations on the rhythm. Basically the rhythm is implemented as a duration. The duration being a fraction of a whole. Listed below are a few of the rules used. Since, duration is represented as fractions, the rules end up being mathematical operations on fractions.

frac_add(*Duration1*, *Duration2*, *TotalDuration*)

Adds the fraction in *Duration1* with the fraction in *Duration2* and returns the result in *TotalDuration*. An attempt is made to reduce the fraction, and if the result is negative, the numerator has the negative sign.

Example:

```
?- frac_add([1,2],[1,2],TotalDuration).
   TotalDuration = [1,1];
   no
```

frac_sub(*Duration1*, *Duration2*, *Difference*)

Subtracts *Duration2* from *Duration1* and returns the result in *Difference*. An attempt is made to reduce the fraction, and if the result is negative, the numerator has the negative sign.

Example:

```
?- frac_sub([1,2],[1,4], Difference).
   Difference = [1,4];
   no
```

frac_equal(*Duration1*, *Duration2*)

Checks for equality between *Duration1* and *Duration2*.

3.2.2 Basic Music and Note Functions

This section will describe the general music rules implemented in modules BasicMusic.p and BasicNoteFunctions.p. The rules will be listed in alphabetical order. Exceptions to this are rules that are dependent on a parent rule such as recursive functions.

chord_to_mode(*Chord*, *Mode*)

For a given chord, returns which *Mode* the specified *Chord* exists in.
Backtracking will return other modes.
Implementation file: BasicMusic.p
Example:

```
?- chord_to_mode([a, [], maseven], Mode).
  Mode = ionian ;
  Mode = lydian ;
  Mode = sixthHM ;
  Mode = major ;
  no
?- chord_to_mode([a, [], maseven], Mode).
  Mode = ionian ;
  Mode = lydian ;
  Mode = sixthHM ;
  Mode = major ;
  no
```

find_root(*Chord*, *Mode*, *RootNote*, *Degree*)

Returns a possible *RootNote* and *Degree* for a given *Chord* in a given *Mode*.
Implementation file: BasicMusic.p
Example:

```
?- find_root([[a, [], maseven], [1,1]], ionian, Root, D).
  Root = [a,[]], D = 1 ;
  Root = [e,[]], D = 4 ;
  no
```

get_notes_from_interval_list(*Note*, *IntervalList*, *NoteList*)

Converts an interval list into a sequence of notes. The sequence of notes will use the degree based on the interval. An interval of mi2 or ma2 will result in the note being one degree apart. An interval of mi6 or ma6 will result in the notes being 5 degrees apart.

Implementation file: BasicMusic.p

Example:

```
?- get_notes_from_interval_list([c,[],2],[ma2, mi2, ma2],N).
  N = [[c,[],2],[d,[],2],[e,[f],2],[f,[],2]] ;
  no
```

get_root_name(*NoteName1*, *NoteName2*, *Degree*)

Returns the number of degree's between two note names.

Implementation file: BasicMusic.p

Example:

```
?- get_root_name(a, c, P).
  P = 6 ;
  no
?- get_root_name(a, G, 5).
  G = d ;
  no
?- get_root_name(a, N,D).
  N = a, D = 1 ;
  N = g, D = 2 ;
  N = f, D = 3 ;
  N = e, D = 4 ;
  N = d, D = 5 ;
  N = c, D = 6 ;
  N = b, D = 7 ;
  no
```

in_consonance(*Mode*, *Note*, *NoteList*)

Determines if a given note list is in consonance relative to the *Mode* and the *Note* specified.

Implementation file: BasicMusic.p

Example:

```
?- in_consonance(ionian, [c,[],1], [[c,[],1],[d,[],1],[e,[],1]]).
```

```
    yes
?- in_consonance(M, [c,[],1], [[c,[],1],[d,[],1],[e,[],1]]).
    M = ionian ;
    M = lydian ;
    M = mixolydian ;
    M = thirdHM ;
    M = lydianAug ;
    M = lydianF7 ;
    M = mixolydianF6 ;
    M = major ;
    no
```

match_chord_mode(*mode, degree, chord type*)

Matches the *chord type* for a *mode* and the given *degree*. Degree must be an instantiated variable. Backtracking will attempt to find other solutions.

Implementation file: BasicMusic.p

Example:

```
?- match_chord_mode(ionian, 3, ChordType).
    ChordType = miseven;
    no
?- match_chord_mode(Mode, 1, ChordType).
    Mode = ionian, ChordType = maseven;
    Mode = dorian, ChordType = miseven;
                :
    Mode = firstHM, ChordType = misevens7
                :
    Mode = melodicMinor, ChordType = misevens7
                :
    Mode = superLocrian, ChordType = hdim;
    no
```

next_note_name(*note name1, note name2, numeric*)

Determines the note name adjacent to a given note name. The numeric field will be 1 if a wrap-around occurs, otherwise it will be zero. Backtracking will fail.

Implementation file: BasicNoteFunctions.p

Example:

```
?- next_note_name(a, NextNote, WrapAround).
  NextNote = b, WrapAround = 0;
  no
?- next_note_name(PreviousNote, a, WrapAround).
  PreviousNote = g, WrapAround = 1;
  no
```

note_distance(*Numeric*, *Note1*, *Note2*)

Determines the number of halfsteps between two notes. Distance is measured relative to *Note2*. Note1 and Note2 must be symbols or instantiated variables. If the a desired note some number of halfsteps away from another note is desired, shift_note_interval should be used.

Implementation file: BasicNoteFunctions.p

Example:

```
?- note_distance(HS, [a,[],1],[c,[f],1]).
  HS = -2 ;
  no
?- note_distance(-2, [a,[],1],N).
  no
```

notes_of_chord(*Chord*, *NoteList*)

Returns the list of notes for a given chord.

Implementation file: BasicMusic.p

Example:

```
?- notes_of_chord([[c, [], maseven], [1,1]], N).
  N = [[[c,[],5],[1,4]],[[e,[],5],[1,4]],[[g,[],5],[1,4]], [[b,[],6],[1,4]]] ;
  no
```

reverse_sharps_flats(*SFList1*, *SFList2*)

Reverses a list of sharps to a list of flats or a list of flats to a list of sharps. No error checking is done on the sharp or flat list. If either of the input lists does not contain a valid symbol, the result is also undeterminable. Backtracking will fail.

Implementation file: BasicNoteFunctions.p

Example:

```
?- reverse_sharps_flats([f,f],N).
  N = [s,s]
?- reverse_sharps_flats([s,s],N).
  N = [f,f]
?- reverse_sharps_flats(N,[f]).
  N = [s] ;
  no
```

sf_value(*sf list*, *numeric*)

Allows the conversion of an *sf list* to a halfstep value (*numeric*) or the reverse, the conversion of a half step value to a list of sharps or flats. Backtracking will fail.

Implementation file: BasicNoteFunctions.p

Example:

```
?- sf_value([f,f,f], SFValue).
  SFValue = -3;
  no
?- sf_value(SFList, 2).
  SFList = [s,s];
  no
```

shift_note(*note1*, *degree1*, *mode*, *degree2*, *note2*)

Returns the note that is a given number of degrees from another note. *degree1* is the degree that *note1* belongs to for the *mode* scale. *degree2* is the degree *note1* is from *note2*. Backtracking will fail.

Implementation file: BasicNoteFunctions.p

Example:

```
?- shift_note([g, [], 1], ionian, 6, Note2).
  Note2 = [f, [s], 2];
  no
?- shift_note([a, [], 1], 3, ionian, 1, Note2).
  Note2 = [b, [f], 1];
  no
```

shift_note_interval(Note1, Numeric, Degree, Note2)

Shifts Note1 the number of halfsteps given in Numeric. Degree will specify the note name of note 2.

Implementation file: BasicNoteFunctions.p

Example:

```
?- shift_note_interval([a,[],1],2,2,N).
   Degree = [c,[f],1] ;
   no
?- shift_note_interval([a,[],1],HalfSteps,2,[c,[f],1]).
   HalfSteps = 2 ;
   no
?- shift_note_interval([a,[],1],2,Degree,[c,[f],1]).
   Degree = 2 ;
   no
```

3.3 Level 3- High Level Functions

Level 3 contains the higher level functions. This section is broken up into 3 sections, the input and output routines, the note generation methods, and the chord analysis or progression section.

3.3.1 Input and Output Routines

The Input and Output modules contain all the input and output rules. The Input module is dependent on the Music Symbols module to convert input symbols into their corresponding internal representation. The rules in the Input module allow general keyboard input, menu prompts, and processing of input. General keyboard input is used to enter input or output file names. Window like menus are displayed for boolean queries. The menu rules will display the menu, prompt for input, and then return the chosen selection. Processing of input is required when input data is required. The input data is parsed and translated into an internal format. The following is a partial list of the rules in the Input module.

select_option(*Options*, *OptionSelected*)

Options is a list of two items. The first item is a string which is displayed to the user. The second item is the corresponding atom that is returned for which ever option the user selected. *OptionSelected* is instantiated with this item. This rule will display the options proceeding each item with a number. The user will be prompted for a number corresponding to the option they desire.

Example:

```
?- select_option([[messageX, first], [messageY, second]],Option).

1) messageX
2) messageY
Enter Choice:2

    Option = 2
```

read_file(*FileStream*, *NoteList*, *ChordList*)
Reads from the *FileStream* the *NoteList* and the *ChordList*.

get_file_name(*Prompt*, *FileName*, *FileType*)
Prompts user for a file name. If the user specified file does not exist or is in a different search path, the standard Macintosh Finder dialog display will appear. The *Prompt* is a string that is used for the first query to the user. The *FileName* is a string containing the file name that the user has inputted. The *FileType* is only partially implemented. It may be used in future additions to restrict to only entering file names of a specific file type. Currently three file types are defined. They are config, input, and, output, which are all mapped to the Macintosh Finder file type of ['TEXT']. Any other type is mapped to [].

get_notes_from_stream(*Stream*, *NoteList*)
Reads note data from *Stream*. Data inputted from the stream is filtered and the note information is extracted, and added to *NoteList*.

remove_extras(*Input*, *Line*)
This rule is used to extract any extraneous input data from *Input*. Input is a list of characters. Unnecessary data includes spaces or tabs. Comments are also removed here. A comment is considered to be two consecutive dashes and any data following the two consecutive dashes. The resultant is returned in *Line*.
Example:

```
?- explodec("dmi7 -- dminseven", CharList),
   remove_extras(CharList, Line).
   CharList = ['d','m','i','7',' ',' ','-','-',' ','d','m','i','n','s','e','v','e','n'],
   Line = ['d','m','i','7'];
   no
```

ask(*Question*, *Option1*, *Option2*, *Result*)

Displays a window with *Question*, and two buttons labeled *Option1* and *Option2*. The user may then click on the desired option. The option selected is then returned in *Result*. *Question* is a string, and *Option1*, and *Option2* are atoms.

The Output module depends on the Music Environment module. This module contains rules to convert internal data structures to either a readable, audible, or electronically accessible form. Readable form changes internal symbols to some set of text and displays them to the screen. Audible form output will convert a set of internal notes into sounds played through the speaker of the computer. Electronically accessible form allows improvised data to be written to a file for later use. The following is a list of some of the rules in the Output module.

print_note_name(*NoteName*)

Displays the atom corresponding to the *NoteName*. If the *NoteName* is a rest, the text string 'rest' is displayed.

print_sf(*SFList*)

Displays sharps and flats. If there are two sharps, then the text string 'sharp x 2' is displayed. If there are no sharps or flats, nothing is displayed. If there is only one sharp or flat, then a '#' or 'f' is displayed.

Example:

print_sf([f,f]).
flat × 2
yes

play(*ChordList*, *NoteList*)

The current implementation can only output one pitch at any one time so the *ChordList* field is ignored. *NoteList* is passed to rule play_note_list to convert and play the notes.

play_note_list(*List*)

Calls rule convert_whole_play_list to convert the List into an internal note format. The values are then asserted into the database and then passed to play_converted_list to play the note list.

play_saved_output

Prompts user for a file which is then reconsulted into the prolog database. A check is made to determine if the conversion parameters are different than those originally used. If they are different, the user is prompted on whether or not the output should reflect the new parameters. The notes are then played.

save_output

This rule allows the user to save the output of an improvisation. The user is queried for an output file name. The data saved is retrieved from the prolog database. The rules are input, results, pitch_offset, beat_duration, duration_to_beat, and the internal representation of the converted solo.

3.3.2 Note Generation

This section describes the currently implemented note generation methods. The first section describes the stochastic approach of selecting notes. The next section describes the use of random chord notes. The third section describes using chords in conjunction with a binary tree. The last section is a rule based note selection. In most cases the input rhythm is retained by using the same rhythm as the input. Rests that occurred were also retained when the input rhythm was used.

3.3.2.1 Stochastic Note Generation

This method of selecting notes uses weighted intervals selected by random. The interval table described in section 2.2 was used. The table is implemented in module IntervalWeightings.p as rule interval_weights(Interval, WeightUp, WeightDown). Interval represents the internal symbol for a specific music interval. WeightUp and WeightDown are numbers which represent the weights of that interval either up or down. The higher the number the heavier the weight. The first note of the melody is used as the starting note of the improvised solo. The next note is determined by an interval weight table. The interval weight table consists of two numbers which represents a range. If a randomly selected number falls within this range, that interval is selected. The greater the range, the higher weight of that interval. A check is made to keep the interval from going out of the actual high and low note range of the original melody. The rules implemented for the stochastic note generation can be found in module WeightedNoteSelection.p. Some of the rules are as follows.

get_weighted_notes(*Melody*, *WeightedNotes*)

The main weighted note rule. It will initialize the weight database by calling rule calc_weights. The highest and lowest notes in the

original melody are determined by calling get_lo_hi_notes. The *WeightedNotes* are then determined by calling get_next_weighted_note. The first or seed note used is the first note of the *Melody*.

calc_weights(*MaxWeight*)

Determines what the maximum weight can be and stores the range of valid ranges for each interval in the database. The maximum weight is calculated, and a rule weight is asserted for each interval up and down. This rule will be of the form weight(HalfSteps, Degree, RangeLo, RangeHi). Halfsteps is the number of halfsteps up or down. This implementation allows an interval up and an interval down to be represented as separate rules. Degree specifies the degree of the note. RangeLo and RangeHi specify the range for which the random number must fall between for the interval to be selected.

Example:

```
?- calc_weights(MaxWeight).
   MaxWeight = 100;
   no
```

get_weighted_deg_hs(*RDegree, HS*)

Returns a randomly selected degree and halfstep. This rule generates a random number, and then searches through the database for an interval where the random number falls between the weighted range.

Example:

```
?- get_weighted_deg_hs(RDeg, HS).
   RDeg = 5, HS = 8;
   RDeg = -2, HS = -3;
   RDeg = -1, HS = -1;
   :
   :
```

get_next_weighted_note(*Melody, SeedNote, Lo, Hi, WeightedNoteList*)

Generates a list of weighted notes using the rhythm from the *Melody*. Any rests in the *Melody* are retained. *SeedNote* is the

starting note. It is used to determine the next note. The interval that is chosen is relative to this note. *Lo* and *Hi* are the minimum and maximum notes allowed. *WeightedNoteList* is the returned list of notes.

3.3.2.2 Random Chord Note Generation

Random chord note generation will generate the list of notes by randomly selecting notes that make up the chords in the original melody. There are two groupings for this. The first will group all the chord notes into a pool and randomly select from this pool. Since it is possible that a key change could have occurred, this method may result in a note being played over a measure where it doesn't belong. The second grouping eliminates this by grouping the pool of notes by the key. The chord progression is initially analyzed and then broken down into several groups of notes. Notes for each key are picked from the pool that belong with that key. The rhythm for the first method described is strictly a quarter of the duration of the chord. If the chord duration was a whole note, each note that was derived from that chord would be a quarter note. Rhythm for the second method is matched with the input melody. The following is a description of one of the rules used for chord note generation.

get_random_mode_notes(*ChordGroupList*, *NoteGroupList*)

A list of chord groups grouped by mode is transformed into a list of notes grouped within the same group. *ChordGroupList* is a list of a list of chords grouped by mode. The notes of each of these individual lists is then placed into another list of lists. A random list is generated for each of these lists, where the number of notes chosen is twice as many as in the original list. This is because randomly selected notes, may end up picking only the shorter duration notes. Any extra notes are then trimmed off. *NoteGroupList* is the resulting list of a list of notes.

3.3.2.3 Binary Tree

Binary tree selection is an offshoot of the random chord note generation method. Again this method will use notes of the chord, and only the notes of the chord that are in the same key. The difference here is that the notes are not picked in random. Each note of a chord is sorted into a binary tree. The note selection is determined by how the tree is traversed. The current implementation allows six different tree traversals. They are post order, reverse post order, in order, reverse inorder, preorder, and reverse preorder. The rhythm is matched one for one with the input melody. The main part is implemented in the module titled BinaryTree. One of the rules is as follows.

get_binary_tree_notes(*ChordGroupList*, *NoteGroupList*, *Traversal*)

> ChordGroupList is a list of a list of chords grouped by mode. NoteGroupList is the resulting list of a list of notes grouped by mode. Traversal method is an atom representing the traversal method used. Since tree traversal can be thought of as a generic function, the tree traversal methods are not listed here, but in the GeneralFunctions module. get_binary_tree_notes first turns each chord list into a tree of notes, by calling make_tree. The resulting tree is then traversed. This process continues with each group of list of chords.

3.3.2.4 Rule Base

The rule based note selection uses the input notes to determine what notes should be used. Some rules were derived from Levitt's thesis, and some were chosen based upon personal judgment. Most of the rules depend on the previous note and the current note of the input melody. The first note of the note selection is the first note of the melody. Again, the rhythm is identical to that of the input melody. Some of the rules in the RuleBase module are as follows.

get_rule_notes(*SoloList*)

This rule uses data stored in the prolog database in the form of rule_db. Currently three different rules are stored. rule_db(melody, Melody), where Melody contains the original note melody, rule_db(chordList, ChordList), where ChordList is the original set of chords, and rule_db(progList, ProgList), where ProgList is the chord progression generated by get_progression. The results of the generated solo are returned in *SoloList*.

get_rule_notes_rec(*NextNoteCount*, *PrevOutNote*, *NoteList*).

This rule is actually a set of several rules used to constrain the output notes. *NextNoteCount* is the numeric position of the next note to be processed. It is used as a place holder. *PrevOutNote* is the last note that was selected. *NoteList* is the resulting list of notes that is returned. A break down of each of these rules and the order of selection is shown below.

Rule 1: If the previous note is the same as the current note, then repeat the previous note.

Rule 2: If the previous note is a fifth above or below, the next is either a fifth above or below. The fifth up or down is randomly selected.

Rule 3: If the previous note is a major 2nd or less from the current note, then pick a note from the chord. The chord note selected is randomly selected. The octave of the note is selected with the same octave as the previous note. This may result in an interval of major seventh.

Rule 4: If the current note does not belong to the current mode, then use that same note. This rule is to retain any dissonance that the original melody might have placed for a unique sound.

Rule 5: Currently this rule will just repeat the corresponding melody note.

3.3.3 Progression

Progression analysis involves determining which mode a specific chord belongs to. A chord belongs to the mode if the notes used to construct the chord exist in that mode. For example, the notes of a dominant seven chord have the intervals ma3, mi3, mi3. By comparing these intervals with the base major scale, it can be seen that these intervals exist starting with the fifth degree. This is the only location where the dominant seven chord can exists in the major scale. If the dominant seven chord was a G7, the root of the scale can be determined as C. This establishes a possible key for that chord. Analyzing several consecutive chords that result in the same key defines a progression in that key. Analysis of the chord progression is accomplished in Progression.p. Initially, the mode is selected to be either ionian, harmonic minor, or melodic minor. These are the most common modes, and by selecting one of those modes in that order increases the speed by increasing the chance of success. Other modes may also succeed, but these modes may result in odd or unfamiliar progression to the western trained ear. Next, a degree is selected. The degree list was also created in a specific order to increase processing speed. Most progression will progress from one specific degree to another specific degree. There are other cases where any degree will work, but the one listed first is actually the most desirable. Once the degree and the mode have been selected, a root chord or key is selected. This root is used against the next chord. If the next chord is contained in the root key, the chord after it is checked against the root. This process will continue if all the chords belong to the key until the whole chord list is complete. If a key change occurs, the chord may not fit or belong to the root key. When a chord is determined not to fit in a root key, a key change has occurred. The most often occurring key change is from a I to a V key, so that is the first attempt at a key change. This change will only allow the root note to

change. The mode will remain the same. If that change fails, a I to II key change could occur. The next check allows any kind of key change to occur. If that also fails to produce any kind of progression, a mode change is allowed. The mode change is restricted at the point where the previous chord must be a 4 chord, and the next can not be a 5 chord. The next rule will attempt to modify the chord type, so the progression could fit in a minor key. The last rule allows a root change if the preceding note was a II chord. This will essentially allow the cycle of fifths progression to occur. A summary of the chord progression rules are shown in the

	Chord Progression Rule
1	If a new Chord is contained in the root key, then keep key.
2	If the previous chord was a I chord, then allow next chord to be a V chord in a new key.
3	If the previous chord was a I chord, then allow next chord to be a II chord in a new key.
4	If the previous chord was a V chord, then allow next chord to be a II chord in a new key.
5	If the previous chord was a IV chord then force a mode change.
6	If current chord is a minor seven chord, then try to retain key. Change the chord into a 1 chord of that key. The chord type is modified to be a minor seven sharp seven
7	If the previous chord was a II chord, then force a new root for the next chord.
8	Catch all allows any chord or mode change.

Chord Progression Rule table above. A summary of the process just described would be as follows:

Select initial key

a) Select mode (ionian, melodicMinor, and harmonicMinor modes selected first).

b) Select degree (select degree based on following order 2,5,1,4, 7,6,3)

c) Verify chord matches degree for that mode to determine a key. If the chord does exist in that mode, proceed to next chord (rule 4). If the chord does not exist, backtrack to next possible degree. If this fails, backtrack to new modes.

Progress to next chord

1) Using the key determined in step c, check if chord exists in that key. If it does, save the degree that the chord exists in and repeat step 1 with next chord. If the chord does not belong to the current key, a key or mode change is occurring.

2) Check if the preceding chord is a I (one) chord. If it is, then allow current chord to be a V (five) chord. If the current chord can be a V chord of some key, that key is used, and step 1 is used to process the next chord. If the current chord can not exist as a V chord in any key, a different key change shift is used.

3) If the preceding chord was a I chord, then try and let the next chord be a II chord. If the current chord can be a II chord of some key, that key is used, and step 3 is used to process the next chord. If the current chord can not exist as a II chord in any key, another type of mode or key change is needed.

4) If the preceding chord was a V chord, then try and let the next chord be a I chord. If the current chord can be a II chord of some key, that key is used, and step 4 is used to process the next chord. If the current chord can not exist as a II chord in any key, another type of mode or key change is needed.

5) This rule allows a forced mode change. It will allow a mode change providing that previous chord was a IV chord. This will prevent wild chord and mode changes to occur.

6) Since minor key progression cannot always be analyzed the same as major keys, this rule was created to modify the root chord so it would fall under a minor key. If the chord is a miseven key, then this rule will attempt to keep the key and mode and attempt to force the chord as a I chord. If this succeeds, the chord type is change by sharping the seventh note.

7) The circle of fifths progression falls under this rule. This rule allows a new root if the preceding degree was a II chord, but forces the next chord to be a V chord.

8) The case here will just allow any mode or chord that will fit in the progression.

The rules created to analyze the progression are listed in Progression.p. The call to determine a set of chords progression is as follows:

get_progression([*chord*...],[*progression*...])

Converts a list of chords to a progression. Backtracking will attempt to find a new progression.
Implementation file:Progression.p
Example:
?- get_progression([[[a, [], miseven], [1,1]], [[d, [], seven], [1, 1]], [[g, [], maseven], [1, 1]]], P).
P = [[[[g, []], ionian, 2, miseven], [1, 1]], [[[g, []], ionian, 5, seven], [1, 1]], [[[g, []], ionian, 1,maseven], [1, 1]]]

The following describes the process in more details using the first 8 measures of Autumn Leaves as an example. The first 8 measures of Autumn Leaves uses the following chords:

A-7, D7, Gma7, Cma7, F#-7b5, B7, E-7, E-7

The chord progression for Autumn Leaves would progress as follows:

A-7 is the first chord, Ionian mode is selected and a degree of 2 is selected. The A-7 is then recorded as the 2 chord of the G ionian mode. (default).

D7 is the next chord, which is the V degree of g major so the process continues to the next chord (rule 1). This is recorded as the 5 chord of the G ionian mode.

Gma7 is the next chord which is the I degree of g major (rule 1). This is recorded as the 1 chord of the G ionian mode.

Cma7 also belongs in G major, but as the IV chord (rule 1). This is recorded as the 4 chord of G ionian.

F#-7b5 is first converted to a half diminished seven chord. This chord can not be constructed from the ionian mode of G, so rule 1 fails. Rule 2 and 3 checks if the previous chord is a I chord. The previous chord was a IV chord so rule 4 also fails. Rule 5 checks if the previous chord is a IV chord which it is. Rule 5 requires a mode change so an attempt is made to check if the previous chord is a I chord. Since it isn't we will try the default which is to use the II degree of an ionian scale. This switches us into a new key of e in the first harmonic minor mode. This is recorded as the 2 chord of the E first harmonic minor.

B7 is the next chord, which is the V degree of E first harmonic minor (rule 1). This is recorded as the 5 chord of the E first harmonic minor.

E-7 this fails rules 1-5, but rule 6 will try and retain the key by convert the E-7 to a one chord by sharping the seventh. This results in the notes E, G, B, D (#). By sharping the D, it allows the E-7#7 to become the 1 chord of E first harmonic minor. Since this works, rule 6

succeeds. This is recorded as the 1 chord of the E first harmonic minor.

E-7 this also fails rules 1-5, but rule 6 will try and retain the key by convert the E-7 to a one chord by sharping the seventh. Since this works, rule 6 succeeds. This is recorded as the 1 chord of the E first harmonic minor.

The progression for the first 8 measures of Autumn Leaves results in a 2 5 1 4 in G ionian, and 2 5 1 1 in E first harmonic minor.

3.4 Level 4-Improvise

The Improvise module combines all the major functions of the software. It controls the individual tasks in level 3 by setting up the menus, and calling the appropriate rules necessary for the item selected in the menu. Some menu selections require several layers of input, while other menu selections require several rules to be processed. More details about the menu input is described in section 3.7 User Interactions. On the initial consult of Improvise, rules are asserted which allow the loading of each individual modules. Most of the modules are consulted on the initialization by the Improvise module. Some of the more individualistic modules like the actually improvisation method are consulted only when needed. The main menu will be called just after initialization, when the rule Start is called. A partial list of the rules in Improvise are as follows.

load(*File*)

Used to consult or reconsult other files. *File* is a string specifying the name of the file to be consulted. This is in Improvise due to the fact that the Improvise module is always the first module loaded. This will assert the rule **loaded**(*File*) into the database to be referenced at a later time. This prevents multiple loads of the same file.

unload(*File*)

Used to retract rules in the specified file. *File* is a string specifying the name of the file to be retracted. The rule **remove**(*File*) must be a rule in the file. **unload** will call this rule to retract the rules contained in file. It will then retract **loaded**(*File*) from the database. Rules **load** and **unload** were added due to memory constraints.

start

Used to initialize and start up the code. This is called just after all the required modules have been loaded in. This will cause the display of

the main menu, and then execute the rule corresponding to the chosen selection.

change_config

Allows user to reconsult a configuration file. User will be prompted for a file name, and the specified file will be reconsulted.

play_saved_output

Allows the user to hear the output of a previously saved improvisation. User will be prompted for a file name. The specified file will then be read in and played for the user.

seed_random_number

Allows the user to change the random number table seed. User will be prompted for a number.

improvise

Displays improvisation methods, calls method selected, outputs audio melody, and then queries to save data.

binary_tree(*ChordList*, *SoloList*)

Queries user for method of binary tree traversal, and input file name. Input file is read in and chord input is then analyzed for a progression. Module NotesFromBinaryTree.p is loaded into memory. The progression is then stored into a binary tree, and the binary tree traversal method inputted earlier is applied to the binary tree. Module NotesFromBinaryTree.p is unloaded and the results are asserted in rules **input**(*Melody*, *ChordList*), and **results**(*ProgList*, *SoloList*).

weighted_note_selection(*ChordLst*, *NoteList*)

Queries user for input file. Input file is read in and module WeightedNoteSelection.p is loaded. Rule to generate list of weighted notes is applied, module WeightedNoteSelection.p is

unloaded, and results are asserted into rules **input**(Melody, *ChordLst*), and **results**([], *NoteList*).

match_notes_randomly_mode(*ChordLst*, *NoteList*)

Queries user for input file. Reads input file and loads module RandomNotesFromChord. Chord input is then analyzed for a progression. The progression list is then grouped by keys using rule **get_key_groupings**. Another rule is applied to randomize the notes within each group. The rhythm is extracted from the original melody and reapplied to the notes. RandomNotesFromChord is unloaded and the results are asserted in rules **input**(*Melody*, *ChordList*), and **results**(ProgList, *SoloList*).

get_notes_with_rules(*ProgList*, *SoloList*)

Queries user for input file. Reads in the file and chord input is then analyzed for a progression. RuleNoteSelection module is loaded. Information needed by this module is asserted into database. Rule is called to generate improvised notes. Information required by module is retracted from database. RuleNoteSelection is unloaded and the results are asserted in rules **input**(*Melody*, *ChordList*), and **results**(*ProgList*, *SoloList*).

3.5 User Interactions

The user interface will be a mixture of a menu driven system, along with keyboard and sometimes mouse selected input. Driving the user input with exclusive mouse input is beyond the scope of this thesis. There are two main menus that are not direct rules. These are consulted on start up. Each of these selections has an associated rule. These rules are described more thoroughly in section 3.4 Improvise.

The first menu that appears is the main menu shown in figure 16. In this menu, there are 5 options. The first option improvise, allows the user to input an

```
1) Improvise
2) Change Config
3) Play previously saved version
4) Seed Random Number Table
5) Exit to finder
Enter Choice:_
```

Figure 16 Main Menu

improvisation method, and then a file where the input data is stored. Upon completion, the user will be prompted on whether or not to save the output data.

The next option Change Configuration, currently allows three parameters that effect the output to be changed. These parameters are the pitch offset, the beat duration, and the duration to beat. Figure 17 shows the contents of the default configuration file. The Change Configuration option will prompt the user for a configuration file. The file entered will then be reconsulted into the prolog database. The pitch offset is a pitch adjustment value that the user may change. The default value is 10. Changing this value will change the pitch that is heard by the user when a pitch is played. Increasing this value will increase the pitch of the played notes. This value has no effect on how the notes are generated. The beat duration and duration to beat allow the user to change the tempo of the audio output. The

```
/*-------------------------------------------------------------------------
    pitch_offset

Change this value to 'tune' your notes */
pitch_offset(10).

/*-------------------------------------------------------------------------
    beat_duration

Change this to lengthen or shorten durations. Higher values will result in
longer durations, lower values will result in shorter durations */
beat_duration(300).

/*-------------------------------------------------------------------------
    duration_to_beat

This is the unit used against the beat_duration. */
duration_to_beat([1,4]).
```

Figure 17 Sample Config file.

default for **beat_duration** of 300 and **duration_to_beat** of [1,4] represents the fact that a note with the duration of [1,4] (a quarter note) will have a duration of 300. Increasing the beat duration value will increase the duration of the corresponding duration to beat value.

The third option allows the user to play a previously saved version. The previously saved version would be one that was saved after selecting option one (Improvisation). If the configuration was changed, the user will be prompted as to whether or not the file should be converted with the new configuration.

Option four allows the user to reseed the random number table. The default seed is 10. The last option exits to the finder.

The menu that appears after selecting Improvisation in the main menu is the improvisation menu. Currently this menu has four different methods of note selection. These options are more thoroughly described in section 3.3.2 Note Generation Methods.

1) Weighted Notes
2) Random Chords Grouped by Mode
3) Binary Tree
4) Rule Base
5) Return to Main Menu
Enter Choice:_

Figure 18 Improvise Menu

With many of the Improvisation options, an input file is needed. The file is a text file with note and chord information of the original melody. The melody is listed first followed by the chord list. The text file may contain any number of extra spaces without causing any problems. A comment sequence is also permitted. Comments are recognized when two consecutive dashes are encountered in the text file. The comment sequence along with any characters on that line up to a new line (carriage return character) are ignored. The delimiter between the melody and the chord list is a blank line. Comments are allowed on this blank line, but anything preceding the comments in this line may be interpreted as additional note input. Each note requires two lines. The first line contains the note name, sharps or flats, and an octave. The actual symbol used is defined in MusicSymbols.p. The corresponding description is described in section 3.1.1 Music Symbols. The examples used here will use the default symbols of {a,b,c,d,e,f,g} for the note names, {f,s} for the flats and {r} to represent a rest. The second line represents the duration. The duration is represented by a whole number or a fraction. If the duration is represented by a whole number, this whole number is interpreted as the denominator of a fraction where the numerator is a 1. The whole number representation assumes the 1/ prefix. A whole note would require the number 1. A quarter note would require the number 4. A fraction would be represented by two numbers separated by a slash. A sample of three possible input notes is shown in figure 19. The first note is a whole note c at octave 5. The second note is a half note

```
c5      -- C in octave 5
1       -- a whole note
cs5     -- C sharp  in octave 5
1/2     -- a half note
ff4     -- F flat in octave 4
4       -- a quarter note
```

Figure 19 Sample Input Notes

c sharp at octave 5. The duration is represented as 1/2, but may also be represented as 2. The third note is a quarter note f flat at octave 4.

Chord input also requires two lines. The second line which represent duration is the same as the note input. The first line which describes the chord is similar to the note, but instead of the octave, a chord type is needed. Section 3.1.1 shows a table of the allowed input chord types. A sample input of three chords is shown in Figure 20. The first chord is an a minor seven chord that lasts one measure. The next chord

```
ami7  -- A minor seven chord
1     -- Whole note
df7   -- D flat seven chord
2     -- Half note
ama7  --A major seven chord
2     -- Half note
```

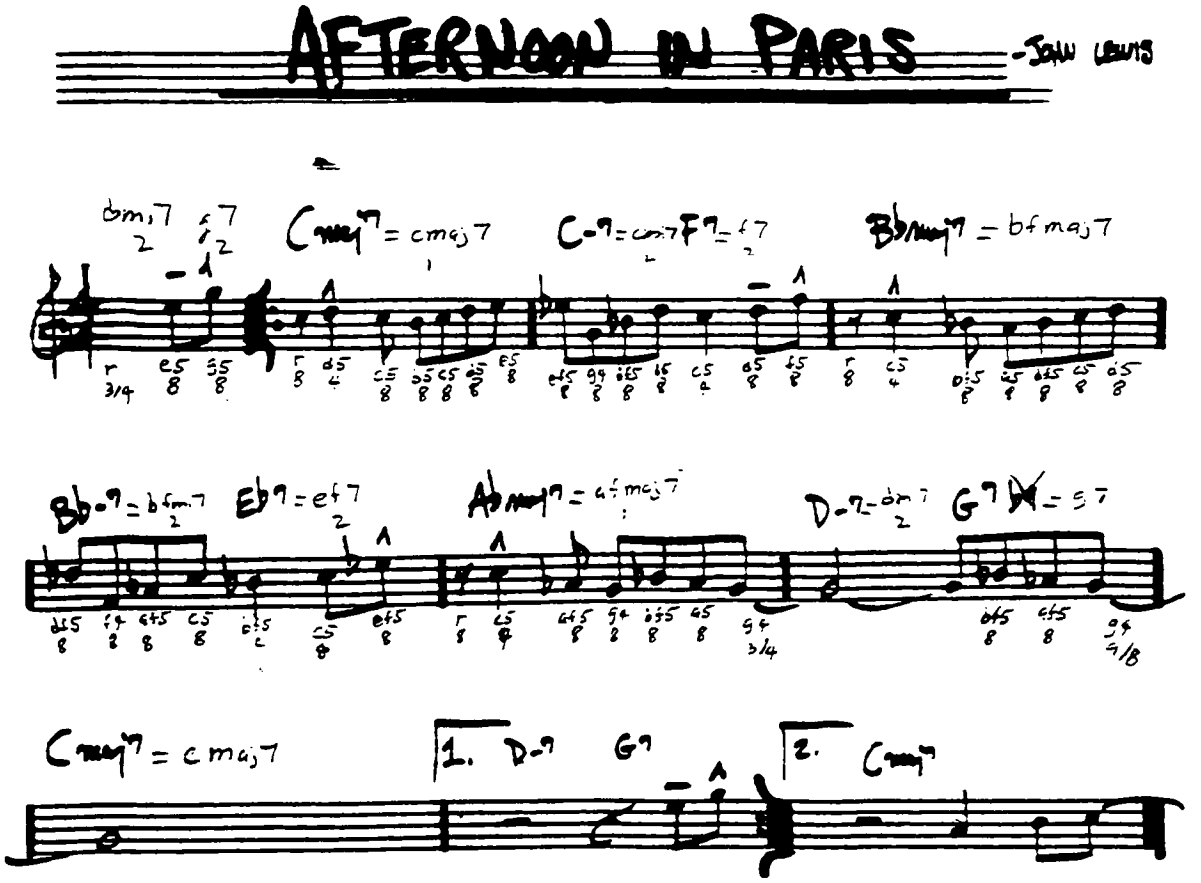
Figure 20 Sample Input Chords

is a d flat dominate seven chord that lasts for a half a measure. The last chord is an a major seven chord that also lasts for a duration of a half measure. A complete sample listing of an input file is shown in the appendix.

4.0 Results

This section will have the results of several songs that were improvised upon by the software. The format of the result section will be the input data file first, followed by the chord and note analysis, and then a comparison of the input with the output.

The first example will use the rule based note selection method. This will have the complete details, of how the results were attained. The others will give only the final result and a summary of the interesting aspects. The following is the music



score for the first part of "Afternoon in Paris" by John Lewis. Written below each note is the translation of the note to the input value. Next to each Chord is the chord translation to an input string. Since the song actually starts on the fourth

beat, a three quarter rest was inserted in the beginning to allow the measure boundaries to line up. The next diagram is the actual input list to the song "Afternoon In Paris."

```
r    -- notes for Afternoon in Paris
3/4
e5
8
g5
8
r
8
d5
4
c5
8
b5
8
c5
8
d5
8
e5
8
ef5
8
g4
8
bf5
8
d5
8
c5
4
d5
8
f5
8
r
8
c5
4
bf5
8
a5
8
bf5
8
c5
8
d5
8
df5
8
f4
8
af5
8
c5
8
bf5
4
c5
8
ef5
8
r
8
c5
4
af5
8
g4
8
bf5
8
a5
8
g4
3/4
bf5
8
af5
8
g4
9/8
-- separator is just a blank line
dmi7        -- start of chord list
2
g7
2
cmaj7
1
cmi7
2
f7
2
bfmaj7
1
bfmi7
2
ef7
2
afmaj7
1
dmi7
2
g7  -- removed flat 9.
2
cmaj7
1
```

Input file contents for "Afternoon in Paris

When using Note selection by rules, the chords must be analyzed for a progression. The left column indicates what is displayed, while the right column annotates what is occurring.

Trying with mode ionian	As described in the mode analysis section, a mode is selected first. ionian is the first mode that is always tried.
Trying in the key of |c,| ||	The first degree tried is II. The first chord is a dmi7 This is the II degree of c so the key used is c in the ionian scale.
Chords left to process: 11/12 Chords left to process: 10/12 Chords left to process: 9/12	Eleven out of 12 chords have been processed. The next two chords are G7 and CMaj7 Both of these succeed with chord progression rule 1.
A possible I II key change from root |c,| || Chords left to process: 8/12	Cmi7 succeeds with chord progression rule 3 which is a key change to b flat.
Chords left to process: 7/12 Chords left to process: 6/12	The chords F7 and B♭maj7 both succeed with chord progression rule 1
A possible I II key change from root |b,|f|| Chords left to process: 5/12	B♭mi7 succeeds with chord progression rule 3 which is a key change.
Chords left to process: 4/12 Chords left to process: 3/12	Both chords e♭7 and afmaj7 succeeds with chord progression rule 3.
A possible I II key change from root |a,|f|| Chords left to process: 2/12	Dmi7 succeeds with chord progression rule 3.
Chords left to process: 1/12 Chords left to process: 0/12	Both chords g7 and cmaj7 succeeds with chord progression rule 3.

The results of the chord analysis are then displayed as follows.

c: II V I
bf:II V I
af: II V I
c: II V I

The next operation is to apply the rule base to the original notes to generate a new set of notes. Because of the length and redundance of the notes for this song, only the first few note generations will be described. As in the chord analysis, the left side will represent the output on the display, and the right side is a description of what is occurring.

The complete chord to mode and note by note is as follows.

	The first note is a rest. Rests are considered part of the rhythm so it is left as a rest.
Rule 5 used on note at position 2.	The next note is a e. Rules 1-3 fail because the previous note is a rest so no interval can be determined between the current note and the next note. Rule 4 fails because a e does exist in ionian c (The current key). Rule 5 succeeds, which is to use the same note that appears in the melody.
Rule 5 used on note at position 3.	The next note is a g. Rules 1-3 compare intervals between the current and the previous. The interval between g and e is a minor third, so rules 1-3 fail. Rule 4 fails because g does exist in ionian c. Rule 5 will always succeed so the g is used as the next note.
Rest found at position 4 skipping to next note.	The note is not a note but a rest so it is skipped.
Rule 5 used on note at position 5.	The next note is a d. The preceding note is a rest so only rule 5 succeeds.

Rule 3 used on note at position 6.

The next note is a c. The interval between the c and the d is a major second. This succeeds for Rule 3. Rule 3 will randomly select a note from the current chord. In this case the chord is G7 and the note selected was a g.

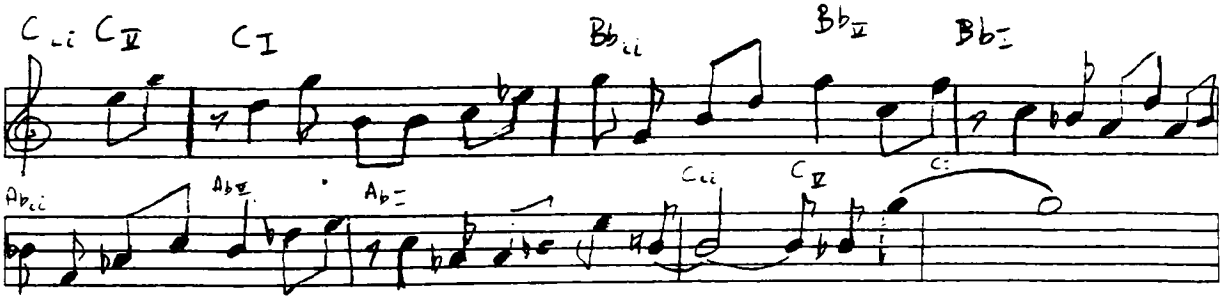
Rule 3 used on note at position 7.

The next note is a b. The interval between the b and the previous melody note c is a minor seconds. Rule 3 succeeds and the note b is selected form the chord G7

Rule 3 used on note at position 8.

The next note is a d. Again Rule 3 succeeds. The note selected from the chord G7 is a b.

Below is the resulting solo transcribed.



Listed below is the actual data output of the solo. As shown, the outputs are actual Prolog data structures. Comments were added and are shown in italics in the following output.

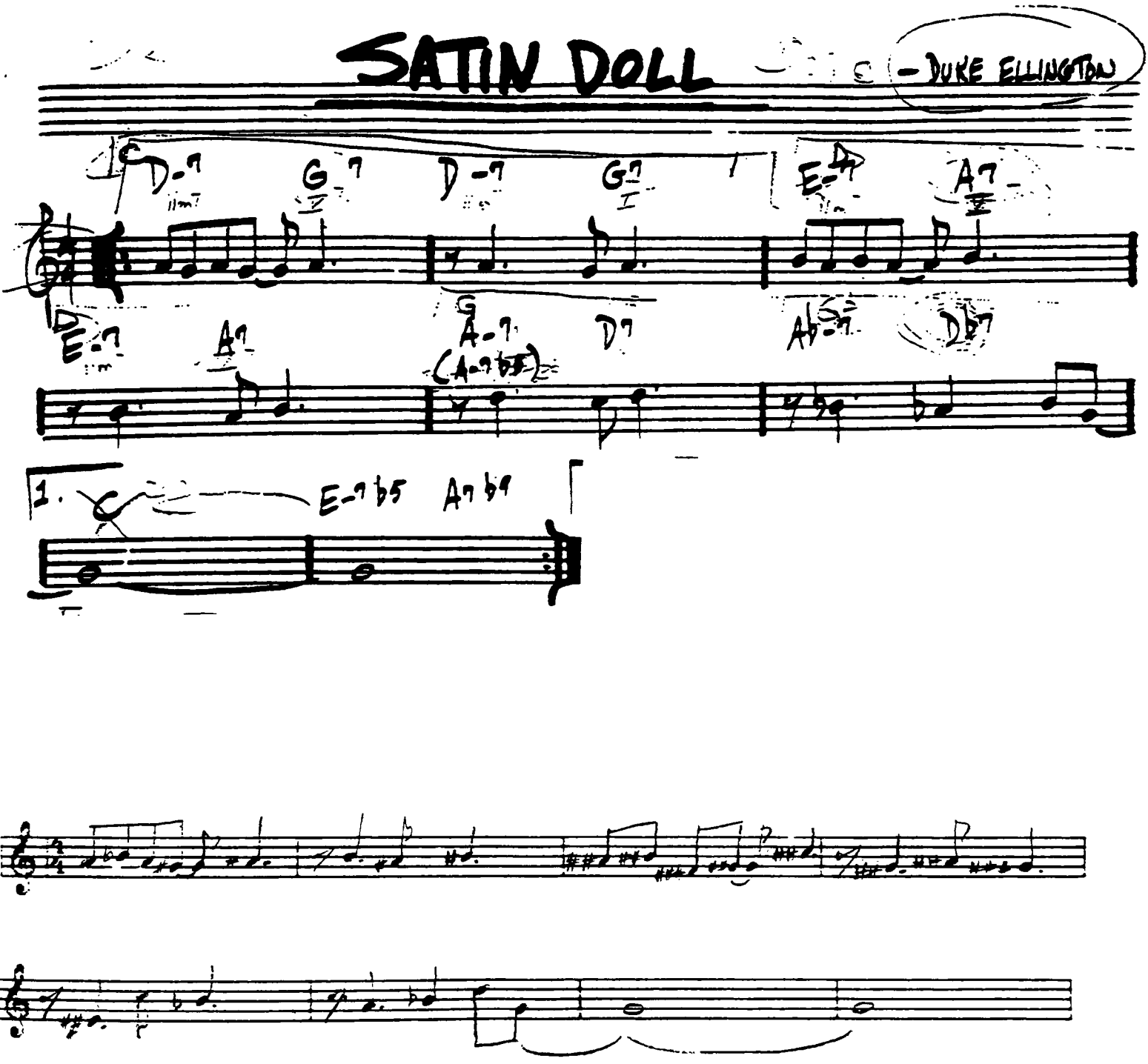
The first 4 lines are within comments and define the input file name. Following the input line is the menu selection command that was performed on the file.

```
/*    begin comment
Input File:HD20:Improvise Data: Input:Afternoon In Paris
"Note selection by rules."
      end comment
input(          The data used for input is contained in this structure.
|||r,|1,0|,|3,4|||,||e,|1,5|,|1,8|||,||g,|1,5|,|1,8|||,||r,|1,0|,|1,8|||,||d,|1,5|,|1,4||.    original melody notes
||c,|1,5|,|1,8|||,||b,|1,5|,|1,8|||,||c,|1,5|,|1,8|||,||d,|1,5|,|1,8|||,||e,|1,5|,|1,8||,
||e,|f|,5|,|1,8|||,||g,|1,4|,|1,8|||,||b,|f|,5|,|1,8|||,||d,|1,5|,|1,8|||,||c,|1,5|,|1,4||.
||d,|1,5|,|1,8|||,||f,|1,5|,|1,8|||,||r,|1,0|,|1,8|||,||c,|1,5|,|1,4|||,||b,|f|,5|,|1,8|||,
||a,|1,5|,|1,8|||,||b,|f|,5|,|1,8|||,||c,|1,5|,|1,8|||,||d,|1,5|,|1,8|||,||d,|f|,5|,|1,8|||,
||f,|1,4|,|1,8|||,||a,|f|,5|,|1,8|||,||c,|1,5|,|1,8|||,||b,|f|,5|,|1,4|||,||c,|1,5|,|1,8||.
||e,|f|,5|,|1,8|||,||r,|1,0|,|1,8|||,||c,|1,5|,|1,4|||,||a,|f|,5|,|1,8|||,||g,|1,4|,|1,8|||,
||b,|f|,5|,|1,8|||,||a,|1,5|,|1,8|||,||g,|1,4|,|3,4|||,||b,|f|,5|,|1,8|||,||a,|f|,5|,|1,8|||,
||g,|1,4|,|9,8|||,
|||d,|1,miseven|,|1,2|||,||g,|1,seven|,|1,2|||,||c,|1,maseven|,|1,1||.        original chords
||c,|1,miseven|,|1,2|||,||f,|1,seven|,|1,2|||,||b,|f|,maseven|,|1,1|||,
||b,|f|,miseven|,|1,2|||,||e,|f|,seven|,|1,2|||,||a,|f|,maseven|,|1,1||,
||d,|1,miseven|,|1,2|||,||g,|1,seven|,|1,2|||,||c,|1,maseven|,|1,1|||).
results(        This data structure cotains all the output information
||||c,|||,ionian,2,miseven|,|1,2|||,|||c,|||,ionian,5,seven|,|1,2||.        resulting chord progression
|||c,|||,ionian,1,maseven|,|1,1|||,|||b,|f|||,ionian,2,miseven|,|1,2||,
|||b,|f|||,ionian,5,seven|,|1,2|||,|||b,|f|||,ionian,1,maseven|,|1,1|||,
|||a,|f|||,ionian,2,miseven|,|1,2|||,|||a,|f|||,ionian,5,seven|,|1,2|||,
|||a,|f|||,ionian,1,maseven|,|1,1|||,|||c,|||,ionian,2,miseven|,|1,2|||,
|||c,|||,ionian,5,seven|,|1,2|||,|||c,|||,ionian,1,maseven|,|1,1|||,
|||r,|1,0|,|3,4|||,||e,|1,5|,|1,8|||,||g,|1,5|,|1,8|||,||r,|1,0|,|1,8|||,||d,|1,5|,|1,4||.    resulting solo
||g,|1,5|,|1,8|||,||b,|1,5|,|1,8|||,||b,|1,5|,|1,8|||,||c,|1,5|,|1,8|||,||e,|f|,5|,|1,8|||,
||g,|1,5|,|1,8|||,||g,|1,4|,|1,8|||,||b,|f|,5|,|1,8|||,||d,|1,5|,|1,8|||,||f,|1,5|,|1,4|||,
||c,|1,5|,|1,8|||,||f,|1,5|,|1;8|||,||r,|1,0|,|1,8|||,||c,|1,5|,|1,4|||,||b,|f|,5|,|1,8|||,
||a,|1,5|,|1,8|||,||d,|1,5|,|1,8|||,||a,|1,5|,|1,8|||,||b,|f|,5|,|1,8|||,||b,|f|,5|,|1,8|||,
||f,|1,4|,|1,8|||,||a,|f|,5|,|1,8|||,||c,|1,5|,|1,8|||,||b,|f|,5|,|1,4|||,||d,|f|,5|,|1,8|||,
||e,|f|,5|,|1,8|||,||r,|1,0|,|1,8|||,||c,|1,5|,|1,4|||,||a,|f|,5|,|1,8|||,||a,|f|,5|,|1,8|||,
||b,|f|,5|,|1,8|||,||e,|f|,5|,|1,8|||,||b,|1,5|,|3,4|||,||b,|f|,5|,|1,8|||,||g,|1,5|,|1,8|||,
||g,|1,5|,|9,8|||).
convert_parameters(10,300,|1,4|).        parameters that affect the audio output only
converted_solo(          The final solo converted to numbers for quicker playback
||0,900|,|77,150|,|80,150|,|0,150|,|75,300|,|73,150|,|72,150|,|73,150|,
|75,150|,|77,150|,|76,150|,|68,150|,|71,150|,|75,150|,|73,300|,|75,150|,|78,150|,|0,150|,
|73,300|,|71,150|,|70,150|,|71,150|,|73,150|,|75,150|,|74,150|,|66,150|,|69,150|,|73,150|,
|71,300|,|73,150|,|76,150|,|0,150|,|73,300|,|69,150|,|68,150|,|71,150|,|70,150|,|68,900|,
|71,150|,|69,150|,|68,1350|,|0,900|,|77,150|,|80,150|,|0,150|,|75,300|,|80,150|,|72,150|,
|72,150|,|73,150|,|76,150|,|80,150|,|68,150|,|71,150|,|75,150|,|78,300|,|73,150|,|78,150|,
|0,150|,|73,300|,|71,150|,|70,150|,|75,150|,|70,150|,|71,150|,|71,150|,|66,150|,|69,150|,
|73,150|,|71,300|,|74,150|,|76,150|,|0,150|,|73,300|,|69,150|,|69,150|,|71,150|,|76,150|,
|72,900|,|71,150|,|80,150|,|80,1350||).
```

Satin Doll

In this example, Satin Doll was used as the input file. The notes were generated based on weighted intervals. Since each interval was assigned a particular degree spacing, the resulting notes were sometimes double and even triple sharped or flatted. Using weighted intervals could be improved upon, by selecting the degree spacing for a minimal amount of accidentals. Also each note could be verified to as belonging to the current mode. This is shown in the next example, where only the chord notes are used.

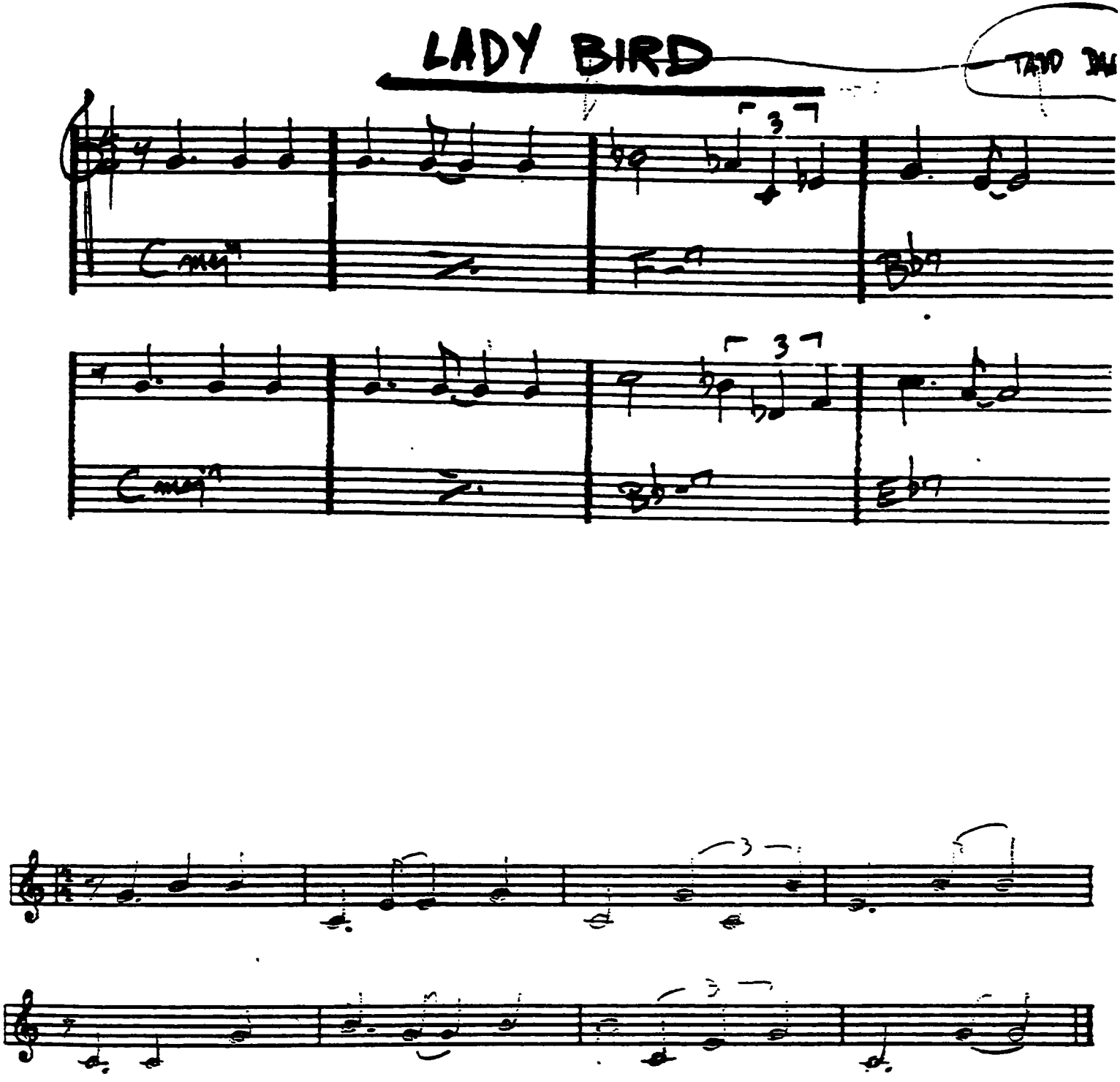
```
/*
Input File:HD20:Improvise Data:Input:Satin Doll
"Notes based on weighted intervals."
*/
input([[[a,[],5],[1,8]],[[g,[],4],[1,8]],[[a,[],5],[1,8]],[[g,[],4],[1,4]],[[a,[],5],[3,8]],
[[r,[],0],[1,8]],[[a,[],5],[3,8]]],[[g,[],4],[1,8]],[[a,[],5],[3,8]],[[b,[],5],[1,8]],[[a,[],5],[1,8]],
[[b,[],5],[1,8]]],[[a,[],5],[1,4]],[[b,[],5],[3,8]],[[r,[],0],[1,8]],[[b,[],5],[3,8]],[[a,[],5],[1,8]],
[[b,[],5],[3,8]]],[[r,[],0],[1,8]],[[d,[],5],[3,8]],[[c,[],5],[1,8]],[[d,[],5],[3,8]],[[r,[],0],[1,8]],
[[b,[f],5],[3,8]]],[[a,[f],5],[1,4]],[[a,[f],5],[1,8]],[[g,[],4],[17,8]]],[[[b,[f],seven],[1,1]]]).

results([[],[[[a,[],5],[1,8]],[[b,[f],5],[1,8]],[[a,[],5],[1,8]],[[g,[s],4],[1,4]],[[a,[s],5],[3,8]],
[[r,[],0],[1,8]]],[[b,[],5],[3,8]]],[[[g,[s],4],[1,8]],[[b,[s],5],[3,8]],[[a,[s,s],5],[1,8]],
[[b,[s,s],5],[1,8]]],[[f,[s,s,s],4],[1,8]]],[[g,[s,s],4],[1,4]],[[c,[s,s],5],[3,8]],[[r,[],0],[1,8]],
[[g,[s,s],4],[3,8]],[[a,[s,s],5],[1,8]]],[[g,[s,s,s],4],[3,8]],[[r,[],0],[1,8]],[[e,[s,s,s],4],[3,8]]],
[[c,[],5],[1,8]],[[b,[f],5],[3,8]],[[r,[],0],[1,8]]],[[a,[],5],[3,8]],[[b,[f],5],[1,4]],[[d,[],5],[1,8]],
[[g,[],4],[17,8]]]]).

convert_parameters(10,300,[1,4]).
converted_solo([[70,150],[68,150],[70,150],[68,300],[70,450],[0,150],[70,450],[68,150
],[70,450],[72,150],[70,150],[72,150],[70,300],[72,450],[0,150],[72,450],[70,150],[72,45
0],[0,150],[75,450],[73,150],[75,450],[0,150],[71,450],[69,300],[69,150],[68,2550],[70,1
50],[71,150],[70,150],[69,300],[71,450],[0,150],[72,450],[69,150],[73,450],[72,150],[74,
150],[69,150],[70,300],[75,450],[0,150],[70,450],[72,150],[71,450],[0,150],[68,450],[73,
150],[71,450],[0,150],[70,450],[71,300],[75,150],[68,2550]]).
```


SATIN DOLL
- DUKE ELLINGTON
D-7
G7
D-7
G7
E-7
A7
E-7
A7
A-7
D7
Ab-7
Db7
(A-7b5)
1.
E-7b5
A7b9

Lady Bird

In this example, of "Lady Bird", the notes were selected randomly by chord notes grouped by mode. The resulting melody here had the potential of being very choppy due to the fact that the mode analysis wasn't very accurate. In this case, there were 5 groups for the eight chords, so only two of the chords ended up being in the same mode. When only one chord is in a group, that duration of the chord has a limited set of notes to select from. The notes will vary in intervals between chord notes.

```
/*
Input File:HD20:Improvise Data:Input:Lady Bird
"Matches notes randomly by chord grouped by mode."
*/
input([[[r,[],0],[1,8]],[[g,[],4],[3,8]],[[g,[],4],[1,4]],[[g,[],4],[1,4]],[[g,[],4],[3,8]],
[[g,[],4],[3,8]],[[g,[],4],[1,4]],[[b,[f],5],[1,2]],[[a,[f],5],[1,6]],[[c,[],4],[1,6]],[[e,[f],4],[1,6]],
[[g,[],4],[3,8]],[[e,[],4],[5,8]],[[r,[],0],[1,8]],[[g,[],4],[3,8]],[[g,[],4],[1,4]],[[g,[],4],[1,4]],
[[g,[],4],[3,8]],[[g,[],4],[3,8]],[[g,[],4],[1,4]],[[c,[],5],[1,2]],[[b,[f],5],[1,6]],[[d,[f],4],[1,6]],
[[f,[],4],[1,6]],[[c,[],5],[3,8]],[[a,[],5],[5,8]]],
[[[c,[],maseven],[1,1]],[[c,[],maseven],[1,1]],[[f,[],miseven],[1,1]],[[b,[f],seven],[1,1]],
[[c,[],maseven],[1,1]],[[c,[],maseven],[1,1]],[[b,[f],miseven],[1,1]],[[e,[f],seven],[1,1]]]).

results([[[[g,[]]],ionian,4,maseven],[1,1]],[[[g,[]],ionian,4,maseven],[1,1]],
[[[c,[]],firstHM,4,miseven],[1,1]],[[[f,[]],melodicMinor,4,seven],[1,1]],
[[[c,[]],ionian,1,maseven],[1,1]],[[[c,[]],ionian,1,maseven],[1,1]],
[[[a,[f]]],ionian,2,miseven],[1,1]],[[[a,[f]],ionian,5,seven],[1,1]]],
[[[r,[],0],[1,8]],[[g,[],5],[3,8]],[[b,[],6],[1,4]],[[b,[],6],[1,4]],[[c,[],5],[3,8]],[[e,[],5],[3,8]],
[[g,[],5],[1,4]],[[c,[],5],[1,2]],[[g,[],5],[1,6]],[[c,[],5],[1,6]],[[b,[],6],[1,6]],[[e,[],5],[3,8]],
[[b,[],6],[5,8]],[[r,[],0],[1,8]],[[c,[],5],[3,8]],[[c,[],5],[1,4]],[[g,[],5],[1,4]],[[b,[],6],[3,8]],
[[g,[],5],[3,8]],[[b,[],6],[1,4]],[[b,[],6],[1,2]],[[c,[],5],[1,6]],[[e,[],5],[1,6]],[[g,[],5],[1,6]],
[[c,[],5],[3,8]],[[g,[],5],[5,8]]]).

convert_parameters(10,300,[1,4]).
converted_solo([[0,150],[68,450],[68,300],[68,300],[68,450],[68,450],[68,300],[71,600
],[69,200],[61,200],[64,200],[68,450],[65,750],[0,150],[68,450],[68,300],[68,300],[68,45
0],[68,450],[68,300],[73,600],[71,200],[62,200],[66,200],[73,450],[70,750],[0,150],[80,4
50],[84,300],[84,300],[73,450],[77,450],[80,300],[73,600],[80,200],[73,200],[84,200],[7
7,450],[84,750],[0,150],[73,450],[73,300],[80,300],[84,450],[80,450],[84,300],[84,600],[
73,200],[77,200],[80,200],[73,450],[80,750]]).
```


LADY BIRD
TADD DA
3
3

In a Mello Tone

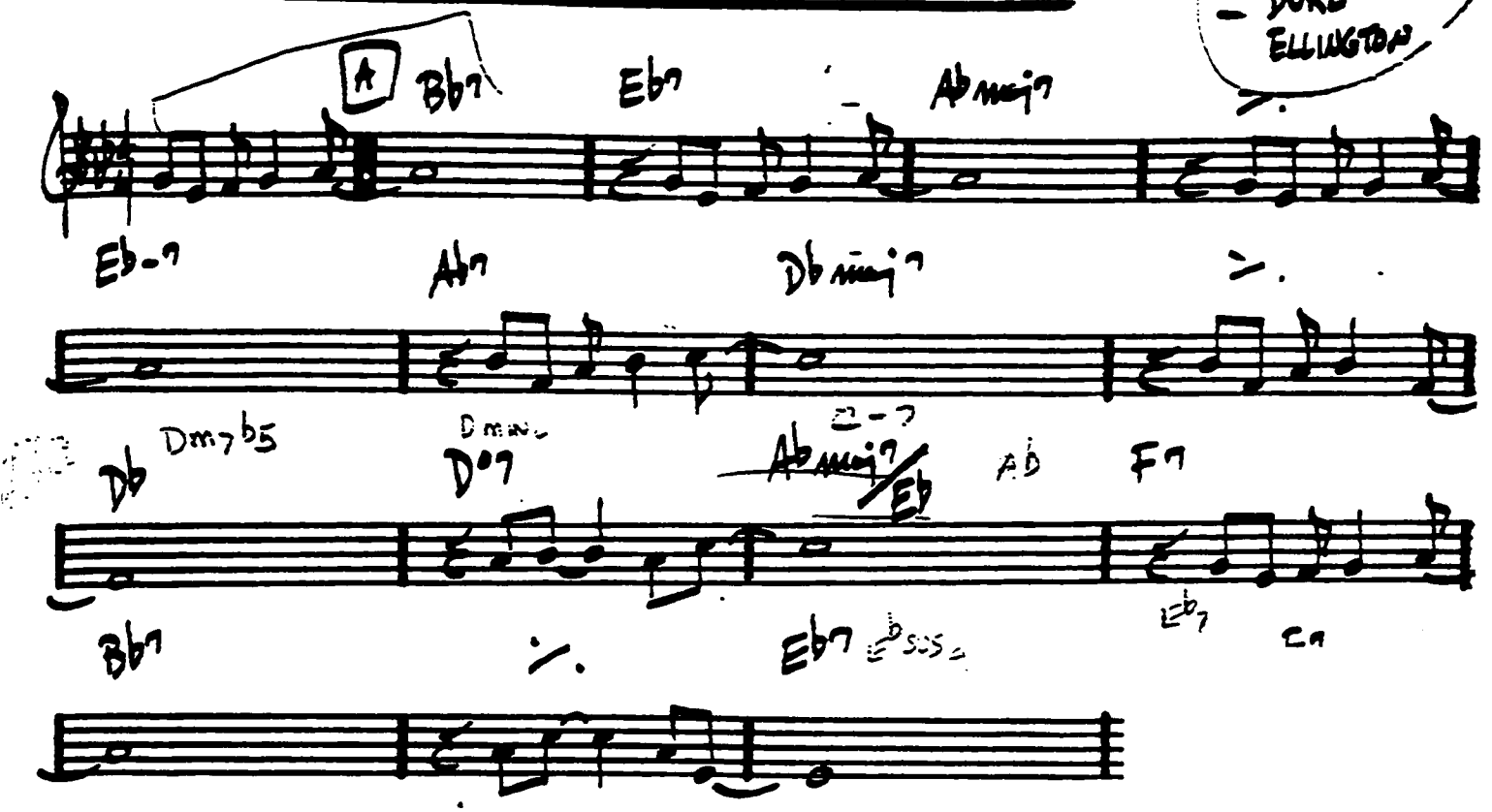
Bad progression analysis resulted in bad notes. Because in this case the chords were selected by mode, and since almost every measure changed modes it ended up using notes of the chord for that measure. For example in the first measure the notes selected were an F, an A natural and an E flat, which are the first, third, and seventh notes of the F7 chord. The result is a boring, and somewhat choppy solo. The A natural note also goes against the key signature, so to some it will also sound out of tune.

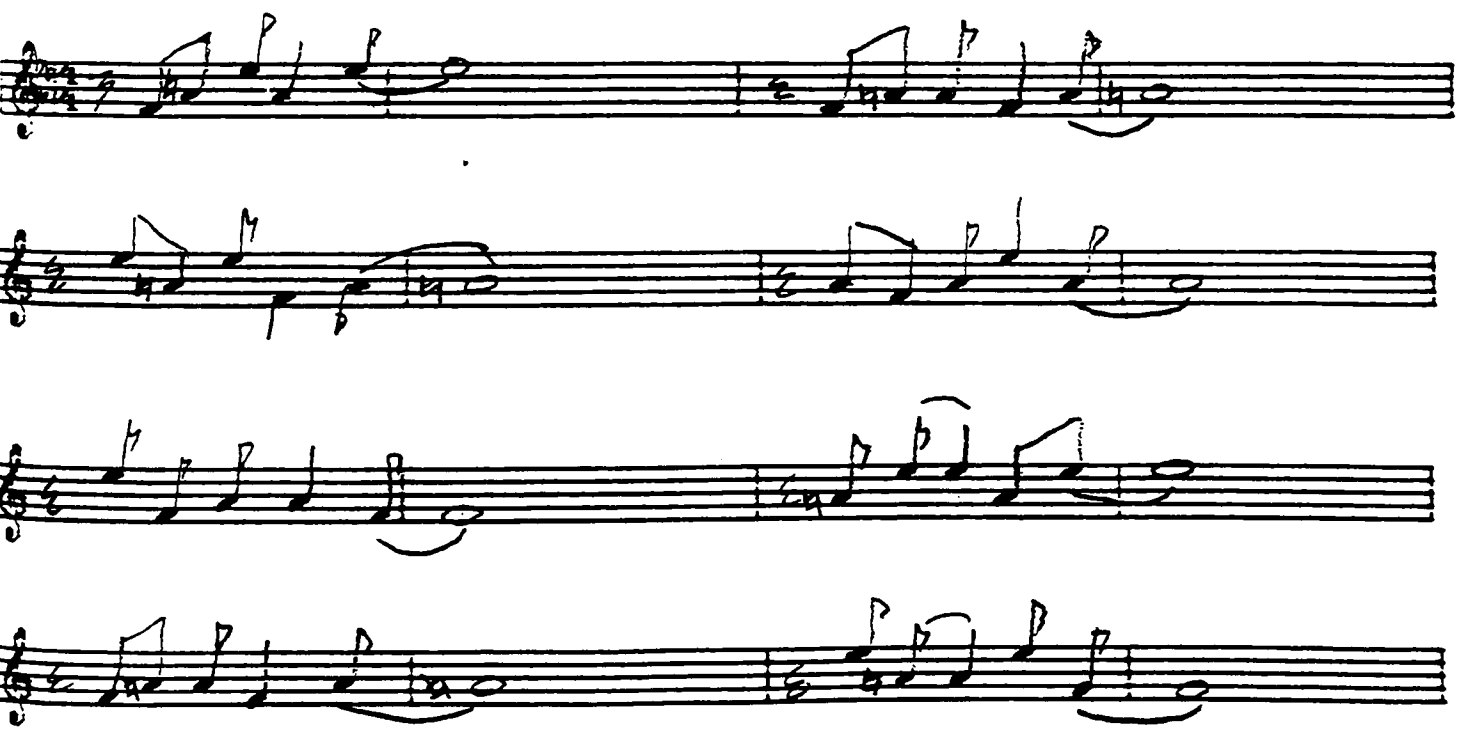
```
/*
Input File:HD20:Improvise Data:Input:In a Mellow Tone
"Matches notes randomly by chord grouped by mode."
*/
input([[[r,[],0],[1,4]],[[g,[],4],[1,8]],[[e,[f],4],[1,8]],[[f,[],4],[1,8]],[[g,[],4],[1,4]],
[[a,[f],5],[9,8]],[[r,[],0],[1,4]],[[g,[],4],[1,8]],[[e,[f],4],[1,8]],[[f,[],4],[1,8]],[[g,[],4],[1,4]],
[[a,[f],5],[9,8]],[[r,[],0],[1,4]],[[g,[],4],[1,8]],[[e,[f],4],[1,8]],[[f,[],4],[1,8]],[[g,[],4],[1,4]],
[[a,[f],5],[9,8]],[[r,[],0],[1,4]],[[b,[f],5],[1,8]],[[f,[],4],[1,8]],[[a,[f],5],[1,8]],
[[b,[f],5],[1,8]],[[c,[],5],[9,8]],[[r,[],0],[1,4]],[[b,[f],5],[1,8]],[[f,[],4],[1,8]],[[a,[f],5],[1,8]],
[[b,[f],5],[1,8]],[[f,[],4],[9,8]],[[r,[],0],[1,4]],[[a,[f],5],[1,8]],[[b,[f],5],[3,8]],
[[a,[f],5],[1,8]],[[c,[],5],[9,8]],[[r,[],0],[1,4]],[[g,[],4],[1,8]],[[e,[f],4],[1,8]],[[f,[],4],[1,8]],
[[g,[],4],[1,4]],[[a,[f],5],[9,8]],[[r,[],0],[1,4]],[[a,[f],5],[1,8]],[[f,[],5],[3,8]],[[a,[f],5],[1,8]],
[[e,[f],4],[9,8]],
[[[f,[],seven],[1,1]],[[b,[f],seven],[1,1]],[[e,[f],seven],[1,1]],[[a,[f],maseven],[1,1]],
[[a,[f],maseven],[1,1]],[[e,[f],miseven],[1,1]],[[a,[f],seven],[1,1]],[[d,[f],maseven],[1,1]],
[[d,[f],maseven],[1,1]],[[d,[],hdim],[1,1]],[[d,[],dim],[1,1]],[[a,[f],maseven],[1,1]],
[[f,[],seven],[1,1]],[[b,[f],seven],[1,1]],[[b,[f],seven],[1,1]],[[e,[f],seven],[1,1]]]).

results([[[[c,[]],melodicMinor,4,seven],[1,1]],[[[f,[]],dorian,4,seven],[1,1]],
[[[b,[f]],melodicMinor,4,seven],[1,1]],[[[e,[f]],ionian,4,maseven],[1,1]],
[[[e,[f]],ionian,4,maseven],[1,1]],[[[b,[f]],firstHM,4,miseven],[1,1]],
[[[e,[f]],melodicMinor,4,seven],[1,1]],[[[a,[f]],ionian,4,maseven],[1,1]],
[[[a,[f]],ionian,4,maseven],[1,1]],[[[a,[f]],lydian,4,hdim],[1,1]],
[[[a,[f]],fourthHM,4,dim],[1,1]],[[[e,[f]],ionian,4,maseven],[1,1]],
[[[c,[]],melodicMinor,4,seven],[1,1]],[[[f,[]],dorian,4,seven],[1,1]],
[[[f,[]],dorian,4,seven],[1,1]],[[[a,[f]],ionian,5,seven],[1,1]]],
[[[r,[],0],[1,4]],[[f,[],5],[1,8]],[[a,[],6],[1,8]],[[e,[f],6],[1,8]],[[a,[],6],[1,4]],[[e,[f],6],[9,8]],
[[r,[],0],[1,4]],[[f,[],5],[1,8]],[[a,[],6],[1,8]],[[a,[],6],[1,8]],[[f,[],5],[1,4]],[[a,[],6],[9,8]],
[[r,[],0],[1,4]],[[e,[f],6],[1,8]],[[a,[],6],[1,8]],[[e,[f],6],[1,8]],[[f,[],5],[1,4]],[[a,[],6],[9,8]],
[[r,[],0],[1,4]],[[a,[],6],[1,8]],[[f,[],5],[1,8]],[[a,[],6],[1,8]],[[e,[f],6],[1,8]],[[a,[],6],[9,8]],
[[r,[],0],[1,4]],[[e,[f],6],[1,8]],[[f,[],5],[1,8]],[[a,[],6],[1,8]],[[a,[],6],[1,8]],[[f,[],5],[9,8]],
[[r,[],0],[1,4]],[[a,[],6],[1,8]],[[e,[f],6],[3,8]],[[a,[],6],[1,8]],[[e,[f],6],[9,8]],[[r,[],0],[1,4]],
[[f,[],5],[1,8]],[[a,[],6],[1,8]],[[a,[],6],[1,8]],[[f,[],5],[1,4]],[[a,[],6],[9,8]],[[r,[],0],[1,4]],
[[e,[f],6],[1,8]],[[a,[],6],[3,8]],[[e,[f],6],[1,8]],[[f,[],5],[9,8]]]).
convert_parameters(10,300,[1,4]).
```

converted_solo([|0,300],|68,150],|64,150],|66,150],|68,300],|69,1350],|0,300],|68,150
],|64,150],|66,150],|68,300],|69,1350],|0,300],|68,150],|64,150],|66,150],|68,300],|69,1
350],|0,300],|71,150],|66,150],|69,150],|71,150],|73,1350],|0,300],|71,150],|66,150],|6
9,150],|71,150],|66,1350],|0,300],|69,150],|71,450],|69,150],|73,1350],|0,300],|68,150]
,|64,150],|66,150],|68,300],|69,1350],|0,300],|69,150],|78,450],|69,150],|64,1350],|0,3
00],|78,150],|82,150],|88,150],|82,300],|88,1350],|0,300],|78,150],|82,150],|82,150],|7
8,300],|82,1350],|0,300],|88,150],|82,150],|88,150],|78,300],|82,1350],|0,300],|82,150]
,|78,150],|82,150],|88,150],|82,1350],|0,300],|88,150],|78,150],|82,150],|82,150],|78,1
350],|0,300],|82,150],|88,450],|82,150],|88,1350],|0,300],|78,150],|82,150],|82,150],|7
8,300],|82,1350],|0,300],|88,150],|82,450],|88,150],|78,1350]|).

IN A MELLOW TONE

– DUKE ELLINGTON





Afternoon In Paris

In the next example, Afternoon in Paris is used again. This time a binary tree approach using post order traversal is used. Again the music is first analyzed for a progression. The resulting progression is first in C (II V I), to B flat (II V I), to A flat (II V I), and finally back to C (II V I). The problem with this is that the resulting solo only uses the notes from the chords for that mode. This limits the creativity, but the successful mode analysis reduces that effect. Parts of the song also seem rigid because of the structure of the binary tree.

```
/*
Input File:HD20:A A I S:AAIS Prolog Folder:Thesis Data Folder:Afternoon in Paris
"Binary tree"
"Postorder"
*/
input([[[r,[],0],[3,4]],[[e,[],5],[1,8]],[[g,[],5],[1,8]],[[r,[],0],[1,8]],[[d,[],5],[1,4]],
[[c,[],5],[1,8]],[[b,[],5],[1,8]],[[c,[],5],[1,8]],[[d,[],5],[1,8]],[[e,[],5],[1,8]],[[e,[f],5],[1,8]],
[[g,[],4],[1,8]],[[b,[f],5],[1,8]],[[d,[],5],[1,8]],[[c,[],5],[1,4]],[[d,[],5],[1,8]],[[f,[],5],[1,8]],
[[r,[],0],[1,8]],[[c,[],5],[1,4]],[[b,[f],5],[1,8]],[[a,[],5],[1,8]],[[b,[f],5],[1,8]],[[c,[],5],[1,8]],
[[d,[],5],[1,8]],[[d,[f],5],[1,8]],[[f,[],4],[1,8]],[[a,[f],5],[1,8]],[[c,[],5],[1,8]],[[b,[f],5],[1,4]],
[[c,[],5],[1,8]],[[e,[f],5],[1,8]],[[r,[],0],[1,8]],[[c,[],5],[1,4]],[[a,[f],5],[1,8]],[[g,[],4],[1,8]],
[[b,[f],5],[1,8]],[[a,[],5],[1,8]],[[g,[],4],[3,4]],[[b,[f],5],[1,8]],[[a,[f],5],[1,8]],
[[g,[],4],[9,8]]],
[[[d,[],miseven],[1,2]],[[g,[],seven],[1,2]],[[c,[],maseven],[1,1]],[[c,[],miseven],[1,2]],
[[f,[],seven],[1,2]],[[b,[f],maseven],[1,1]],[[b,[f],miseven],[1,2]],[[e,[f],seven],[1,2]],
[[a,[f],maseven],[1,1]],[[d,[],miseven],[1,2]]],[[g,[],seven],[1,2]],[[c,[],maseven],[1,1]]]).

results([[[[c,[]],ionian,2,miseven],[1,2]],[[[c,[]],ionian,5,seven],[1,2]],
[[[c,[]],ionian,1,maseven],[1,1]],[[[b,[f]],ionian,2,miseven],[1,2]],
[[[b,[f]],ionian,5,seven],[1,2]],[[[b,[f]],ionian,1,maseven],[1,1]],
[[[a,[f]],ionian,2,miseven],[1,2]],[[[a,[f]],ionian,5,seven],[1,2]],
[[[a,[f]],ionian,1,maseven],[1,1]],[[[c,[]],ionian,2,miseven],[1,2]],
[[[c,[]],ionian,5,seven],[1,2]],[[[c,[]],ionian,1,maseven],[1,1]]],
[[[r,[],0],[3,4]],[[c,[],5],[1,8]],[[d,[],5],[1,8]],[[r,[],0],[1,8]],[[f,[],5],[1,4]],[[g,[],5],[1,8]],
[[a,[],6],[1,8]],[[b,[],6],[1,8]],[[c,[],6],[1,8]],[[f,[],6],[1,8]],[[d,[],6],[1,8]],[[b,[],6],[1,8]],
[[g,[],5],[1,8]],[[e,[],5],[1,8]],[[c,[],5],[1,4]],[[d,[],5],[1,8]],[[f,[],5],[1,8]],[[r,[],0],[1,8]],
[[g,[],5],[1,4]],[[a,[],6],[1,8]],[[b,[],6],[1,8]],[[c,[],6],[1,8]],[[f,[],6],[1,8]],[[d,[],6],[1,8]],
[[b,[],6],[1,8]],[[g,[],5],[1,8]],[[e,[],5],[1,8]],[[c,[],5],[1,8]],[[d,[],5],[1,4]],[[f,[],5],[1,8]],
[[g,[],5],[1,8]],[[r,[],0],[1,8]],[[a,[],6],[1,4]],[[b,[],6],[1,8]],[[c,[],6],[1,8]],[[f,[],6],[1,8]],
[[d,[],6],[1,8]],[[b,[],6],[3,4]],[[g,[],5],[1,8]],[[e,[],5],[1,8]],[[c,[],5],[9,8]]]).

convert_parameters(10,300,[1,4]).
converted_solo([[0,900],[77,150],[80,150],[0,150],[75,300],[73,150],[72,150],[73,150],
[75,150],[77,150],[76,150],[68,150],[71,150],[75,150],[73,300],[75,150],[78,150],[0,150
],[73,300],[71,150],[70,150],[71,150],[73,150],[75,150],[74,150],[66,150],[69,150],[73,1
```

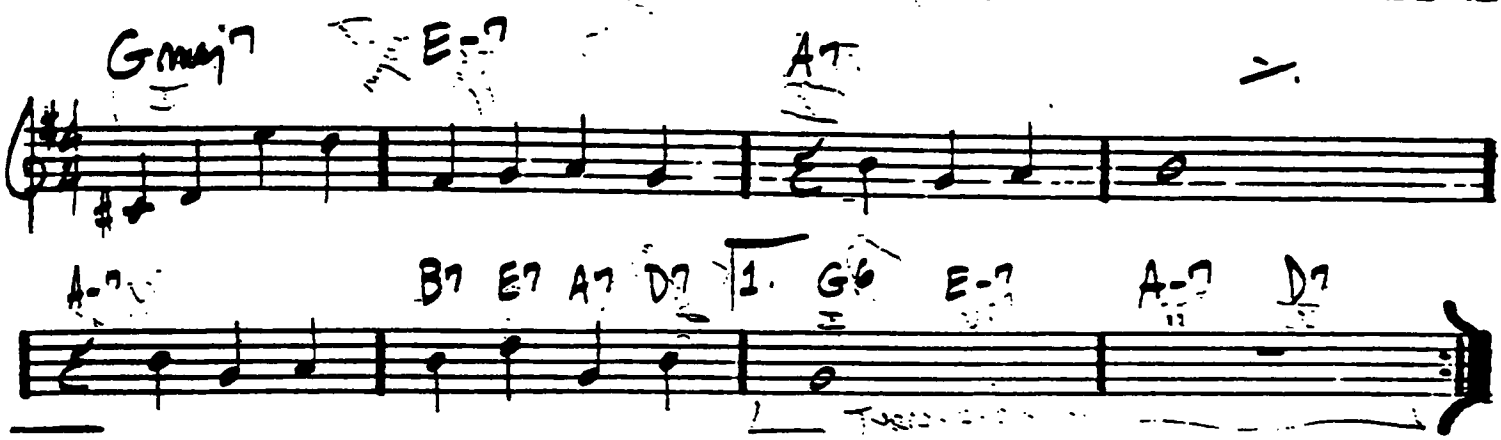
50],[71,300[,[73,150],[76,150],[0,150],[73,300],[69,150],[68,150[,[71,150[,[70,150[,[68,
900[,[71,150],[69,150[,[68,1350[,[0,900[,[73,150[,[75,150[,[0,150[,[78,300[,[80,150],[8
2,150[,[84,150[,[85,150],[90,150[,[87,150[,[84,150[,[80,150[,[77,150],[73,300[,[75,150[
,[78,150[,[0,150[,[80,300[,[82,150[,[84,150[,[85,150[,[90,150[,[87,150[,[84,150[,[80,15
0[,[77,150[,[73,150[,[75,300[,[78,150[,[80,150[,[0,150[,[82,300[,[84,150[,[85,150[,[90,1
50[,[87,150[,[84,900[,[80,150[,[77,150[,[73,1350[[).

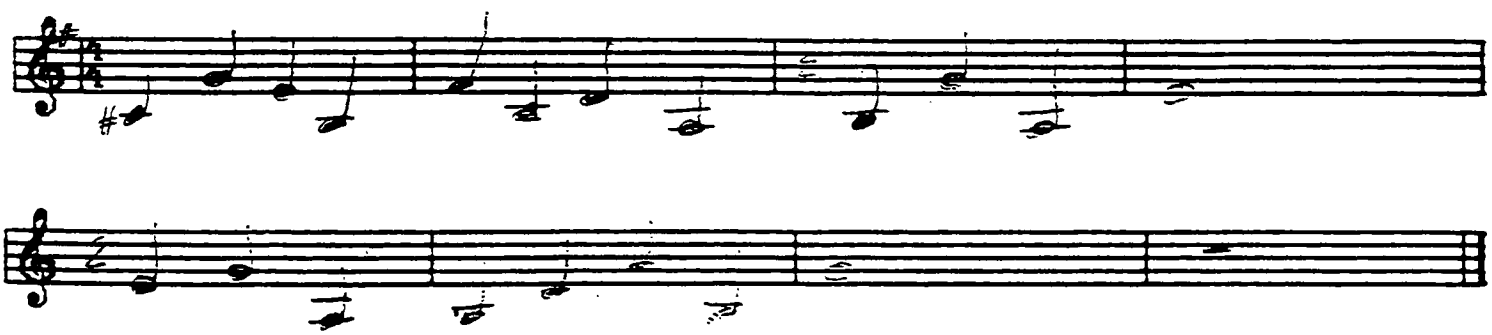
I Got It Bad

In this example, "I Got It Bad" note selection based on rules was used. Here, the problem of a note being out of tonic, or large interval changes does not occur, but some of the selected notes may be out of range. Since the note selection by rules does not put any pitch constraint on the output notes, the notes generated may be too high or too low. In this case, 3 G's were generated which may be beyond a practical range. A new rule should probably be added to the note selection by rules which would pick the high and the low notes of the song, and constrain the output within these pitches.

```
/*
Input File:HD20:Improvise Data:Input:I Got it Bad
"Note selection by rules."
*/
input([[[c,[s],4],[1,4]],[[d,[],4],[1,4]],[[e,[].5],[1,4]],[[d,[],5],[1,4]],[[f,[s],4],[1,4]],
[[g,[],4],[1,4]]],[[a,[],5],[1,4]],[[g,[],4],[1.4]],[[r,[],0],[1,4]],[[b,[],5],[1,4]],[[g,[],4],[1,4]],
[[a,[],5],[1,4]],[[b,[],5],[1,1]],[[r,[],0],[1,4]],[[b,[],5],[1,4]],[[g,[],4],[1,4]],[[a,[].5],[1,4]],
[[b,[],5],[1,4]],[[d,[],5],[1,4]],[[g,[],4],[1,4]],[[b,[],5],[1,4]],[[g,[],4],[1,1]],[[r,[].0],[1,1]]],
[[[g,[,maseven],[1,1]],[[e,[],miseven],[1,1]],[[a,[],seven],[1,1]],[[a,[],seven],[1,1]],
[[a,[],miseven],[1,1]],[[b,[],seven],[1,4]],[[e,[],seven],[1,4]],[[a,[],seven],[1,4]],
[[d,[],seven],[1,4]],[[g,[],masix],[1,2]],[[e,[],miseven],[1.2]],[[a,[],miseven],[1,2]],
[[d,[],seven],[1,2]]]).
results([[[[d,[]],ionian,4,maseven],[1,1]],[[[b,[]],firstHM,4,miseven],[1,1]],
[[[d,[]],melodicMinor,5,seven],[1,1]],[[[d,[]],melodicMinor,5,seven],[1,1]],
[[[g,[]],melodicMinor,2,miseven],[1,1]],[[[f,[s]],melodicMinor,4,seven],[1,4]],
[[[b,[]],dorian,4,seven],[1,4]]],[[[e,[]],melodicMinor,4,seven],[1,4]],
[[[a,[]],dorian,4,seven],[1,4]]],[[[d,[f]],lydian,4,masix],[1,2]],
[[[d,[]],ionian,2,miseven],[1,2]]],[[[g,[]],ionian,2,miseven],[1,2]],
[[[g,[]],ionian,5,seven],[1,2]]]],
[[[c,[s],4],[1,4]],[[g,[],4],[1,4]],[[e,[],5],[1,4]],[[b,[],5],[1,4]]],[[f,[s],4],[1,4]],[[b,[],4],[1,4]],[
[d,[],4],[1,4]],[[a,[],4],[1,4]],[[r,[],0],[1,4]],[[b,[],5],[1,4]],[[g,[],4],[1,4]],[[a,[],4],[1,4]],[[e,
[],4],[1,1]]],[[r,[],0],[1,4]],[[e,[],4],[1,4]],[[g,[],4],[1,4]]],[[a,[],4],[1,4]],[[b,[],4],[1,4]],[[d,[].
5],[1,4]]],[[a,[],6],[1,4]],[[b,[],5],[1,4]],[[g,[],4],[1,1]],[[r,[],0],[1,1]]]).
convert_parameters(10,300,[1,4]).
converted_solo([[62,300],[63,300],[77,300],[75,300],[67.300],[68,300],[70,300],[68,30
0],[0,300],[72,300],[68,300],[70,300],[72,1200],[0,300],[72,300],[68,300],[70,300],[72,3
00],[75,300],[68,300],[72,300],[68,1200],[0,1200],[62,300],[68,300],[77,300],[72,300],
67,300],[60,300],[63,300],[58,300],[0,300],[72,300],[68,300],[58,300],[65,1200],[0,300
,[65,300],[68,300],[58,300],[60,300],[75,300],[82,300],[72,300],[68,1200],[0,1200]]).
```

I GOT IT BAD — DUKE





How High the Moon

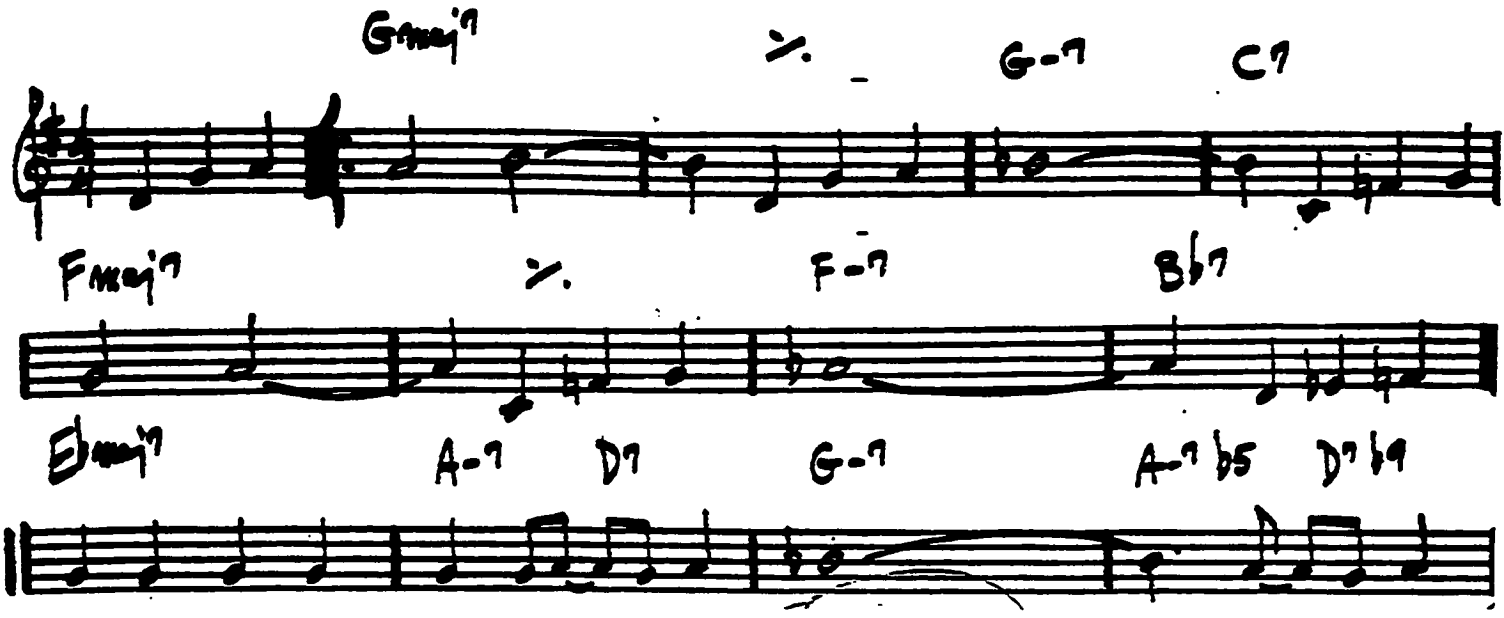
In this example of "How High the Moon", notes were also selected by rules. In this case, the progression analysis results were not very good. The resulting progression here was a series of "4" chords, A melodicMinor 4, D ionian 4, D ionian 4, D first harmonic minor 4, G melodic minor 4,C ionian 4, C ionian 4, C first harmonic minor 4, F melodic minor 4, B flat ionian 4, E first harmonic minor 4, A melodic minor 4, D first harmonic minor 1, G ionian 2, and G ionian 5. The resulting melody output wasn't that bad considering the chord analysis. This is due to the melody generation method used. In note selection by rules, only one of the rules needs the current mode, and this rule isn't used often in this case.

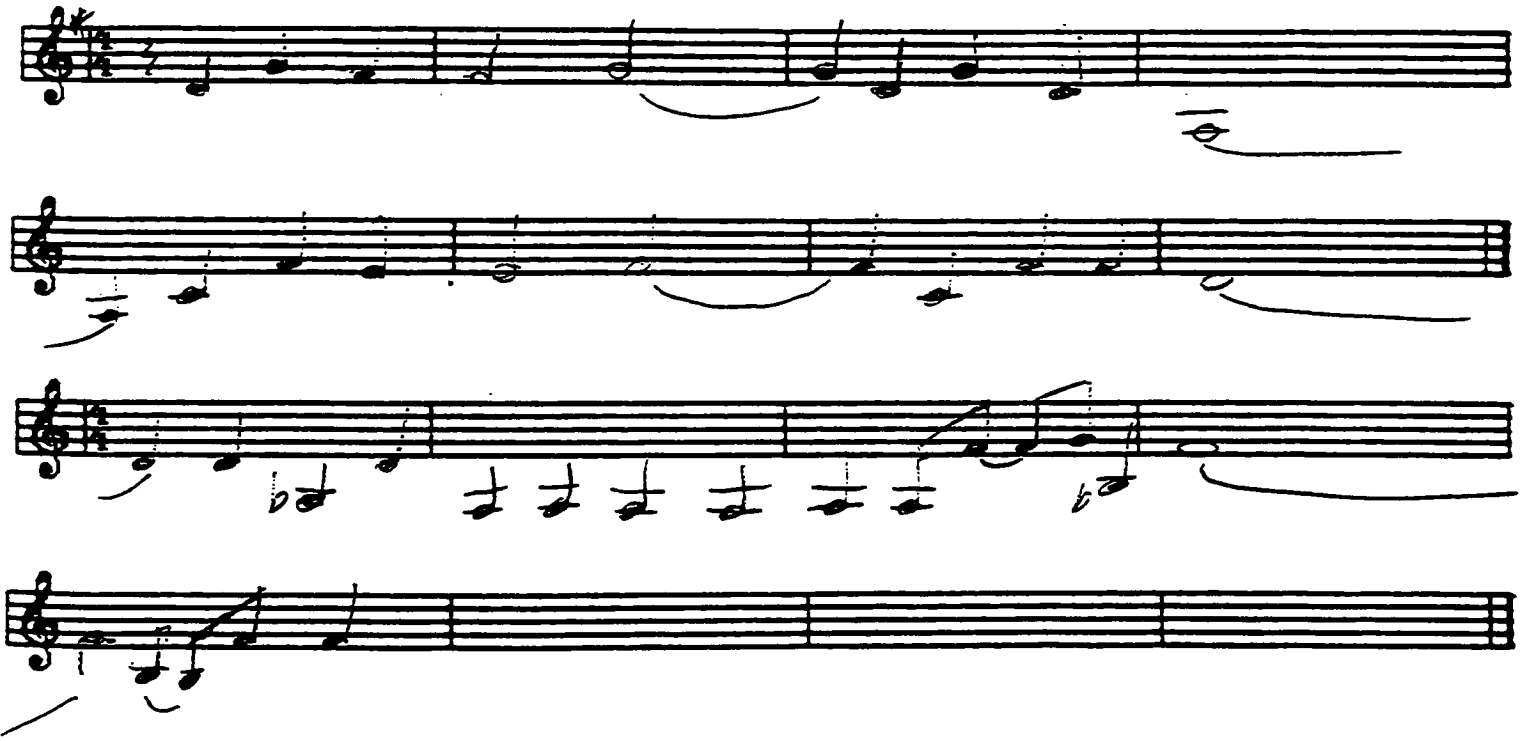
```
/*
Input File:HD20:Improvise Data:Input:How High the Moon
"Note selection by rules."
*/
input([[[r,[]],0],[1,4]],[[d,[],4],[1,4]],[[g,[]],4],[1,4]],[[a,[],5],[1,4]],[[a,[],5],[1,2]],
[[b,[],5],[3,2]],[[d,[],4],[1,4]],[[g,[],4],[1,4]],[[a,[],5],[1,4]],[[b,[f],5],[5,4]],[[c,[],4],[1,4]],
[[f,[],4],[1,4]],[[g,[],4],[1,4]],[[g,[],4],[1,2]],[[a,[],5],[3,4]],[[c,[],4],[1,4]],[[f,[],4],[1,4]],
[[g,[],4],[1,4]]],[[a,[f],5],[5,4]],[[d,[],4],[1,4]],[[e,[f],4],[1,4]],[[f,[],4],[1,4]],[[g,[],4],[1,4]],
[[g,[],4],[1,4]]],[[g,[],4],[1,4]],[[g,[],4],[1,4]],[[g,[],4],[1,4]],[[a,[],5],[1,4]],
[[g,[],4],[1,8]],[[a,[],5],[1,4]],[[b,[f],5],[11,8]],[[a,[],5],[1,4]],[[g,[],4],[1,8]],[[a,[],5],[1,4]]],
[[[d,[],seven],[1,1]],[[g,[]],maseven],[1,1]],[[g,[],maseven],[1,1]],[[g,[],miseven],[1,1]],
[[c,[],seven],[1,1]],[[f,[]],maseven],[1,1]],[[f,[]],maseven],[1,1]],[[f,[],miseven],[1,1]],
[[b,[f]],seven],[1,1]],[[e,[f]],maseven],[1,1]],[[a,[]],miseven],[1,2]],[[d,[],seven],[1,2]],
[[g,[]],miseven],[1,1]],[[a,[]],hdim],[1,2]],[[d,[],seven],[1,2]],[[g,[],maseven],[1,1]],
[[a,[],miseven],[1,2]],[[d,[],seven],[1,2]]]).

results([[[[a,[]],melodicMinor,4,seven],[1,1]],[[[d,[]],ionian,4,maseven],[1,1]],
[[[d,[]],ionian,4,maseven],[1,1]],[[[d,[]],firstHM,4,miseven],[1,1]],
[[[g,[]],melodicMinor,4,seven],[1,1]],[[[c,[]],ionian,4,maseven],[1,1]],
[[[c,[]],ionian,4,maseven],[1,1]],[[[c,[]],firstHM,4,miseven],[1,1]],
[[[f,[]],melodicMinor,4,seven],[1,1]],[[[b,[f]],ionian,4,maseven],[1,1]],
[[[e,[]],firstHM,4,miseven],[1,2]],[[[a,[]],melodicMinor,4,seven],[1,2]],
[[[d,[]],firstHM,4,miseven],[1,1]],[[[e,[f]],lydian,4,hdim],[1,2]],
[[[g,[]],ionian,5,seven],[1,2]],[[[g,[]],ionian,1,maseven],[1,1]],
[[[g,[]],ionian,2,miseven],[1,2]],[[[g,[]],ionian,5,seven],[1,2]]],
[[[r,[],0],[1,4]],[[d,[],4],[1,4]],[[g,[],4],[1,4]],[[f,[s],4],[1,4]]],[[f,[s],4],[1,2]],[[g,[],4],[3,2]],
[[d,[],4],[1,4]]],[[g,[],4],[1,4]]],[[d,[],4],[1,4]]],[[a,[],4],[5,4]],[[c,[],4],[1,4]],[[f,[],4],[1,4]],
[[e,[],4],[1,4]]],[[e,[],4],[1,2]],[[f,[],4],[3,4]],[[c,[],4],[1,4]],[[f,[],4],[1,4]],[[f,[],4],[1,4]],
[[d,[],4],[5,4]],[[d,[],4],[1,4]]],[[b,[f],4],[1,4]],[[d,[],4],[1,4]],[[a,[],4],[1,4]],[[a,[],4],[1,4]],
[[a,[],4],[1,4]]],[[a,[],4],[1,4]],[[a,[],4],[1,4]],[[a,[],4],[1,8]],[[f,[],4],[1,4]],[[g,[],4],[1,8]],
[[b,[f],4],[1,4]]],[[f,[s],4],[11,8]],[[b,[],4],[1,4]]],[[f,[s],4],[1,8]],[[f,[s],4],[1,4]]]).
convert_parameters(10,300,[1,4]).
```

converted_solo([[0,300],[63,300],[68,300],[70,300],[70,600],[72,1800],[63,300],[68,300],[70,300],[71,1500],[61,300],[66,300],[68,300],[68,600],[70,900],[61,300],[66,300],[68,300],[69,1500],[63,300],[64,300],[66,300],[68,300],[68,300],[68,300],[68,300],[68,150],[70,300],[68,150],[70,300],[71,1650],[70,300],[68,150],[70,300],[0,300],[63,300],[68,300],[67,300],[67,600],[68,1800],[63,300],[68,300],[63,300],[58,1500],[61,300],[66,300],[65,300],[65,600],[66,900],[61,300],[66,300],[66,300],[63,1500],[63,300],[59,300],[63,300],[58,300],[58,300],[58,300],[58,300],[58,300],[58,150],[66,300],[68,150],[59,300],[67,1650],[60,300],[67,150],[67,300]]).

HOW HIGH THE MOON

(- MORGAN LEWIS





5.0 Conclusions

This thesis is meant to test the feasibility of using computers to create a Jazz solo. The four different approaches provided by this thesis resulted in four different types of solos. As seen in the previous results section, each type has its imperfections and need for improvements.

Weighted intervals

Weighted intervals was an easy way to generate random notes without total randomness. In some cases, the results were surprisingly good, but in almost all cases, the intervals resulted in many accidentals, which only affects the transcribed solo and not the sound of the music. The accidentals are due to the current implementation, which confines each interval to a degree separation. Removing this constraint, would greatly decrease the number of accidentals while retaining the sound. Another problem is the fact that since the intervals are chosen at random the interval could result in notes that are not in tonic with the current mode, or a non-existant, or misspelled note could occur. Since this uses a random process, it was highly unlikely that a pattern or repeated phrases would occur. These patterns would give the solo more structure. Currently the weights are based on a 1st order analysis. If this was increased to a higher order, there would be a greater chance of a structured note sequence. This process was included just to see how effective weighted intervals would be. The resulting output of this process was sufficient to conclude that some kind of note structure is

important, and some kind of constraints should be made to ensure the notes are in consonance with the mode to develop a good solo.

Note selection from chords

Note selection from chords is another simple way of selecting notes. It is basically the next step from the weighted note selection. Since the notes used are from the chords, they will always be in tonic with the mode. Misspelled notes and excessive accidentals will not occur. One problem here is similar to the weighted notes, -there is no structure. Again better mode analysis might help reduce this effect of just the chord notes played over the chord. Another problem is the excessively large intervals that can occur between chords of different modes. Adding structure and reducing the constraints placed upon what notes can be played will help in giving solos produced by this method more body.

Binary Tree

The binary tree approach is similar to the note selection from chords method of selecting a note, only a binary tree structure is used instead of randomly selecting notes. This results in the notes being in consonance, correct note spelling, and a basic note selection structure. The results weren't too bad, but since repeated notes are very rare using this method, some of the solos were choppy sounding. In some cases, the solo actually sounded mechanical or repetitive. The repeated notes did not occur very often due to the fact that the binary tree couldn't hold two notes of the same pitch. The binary

tree method seemed to have too much repetitive structure, and still lacked originality in selecting notes.

Rule database

The rule database basically consisted of 5 first order rules. Built into the rules were constraints that prevented some of the problems that occurred with the previously mentioned methods. The number of rules was kept to a minimum to reduce too many differences from the original melody. Other improvements to the rule database could include constraints like the weighted interval section. A higher degree of analysis could also be done, which would retain some of the structure from the original melody, and still have its own originality with the addition of randomness to the rules.

The best approach found in creating a Jazz Solo seemed to be the rule data based method. With the addition of more rules and a combination of approaches, a fairly good sounding solo can be produced by the computer. Another change that would improve the overall solos would be to incorperate song form. This would allow the solo to flow into the original melody more smoothly. The overall rule to remember is that music is a form of art. A more complicated solo would have many little intricacies. Human interaction is required to make those changes and adjustments. Because of this, it is very difficult if not impossible for a computer to create a Jazz Solo that everyone would appreciate. However, this Thesis has proven that it is possible to create a simple melody through the use of a computer.

References

[Bake63]Baker, R. A., "A Statistical Analysis of the Harmonic Practice of the 18th and 19th Centuries," D.M.A. Dissertation, University of Illinois, 1963.

[Bate80]Bateman, Wayne Introduction to Computer Music, John Wiley & Sons, Inc, Canada 1980.

[Benw83]Benward, Bruce, Jackson, Barbara G. Practical Beginning Theory, Fifth Edition, Wm. C. Brown Company Publishers, Dubuque, Iowa,1983.

[Bert85]Bertoncini, Gene The Guitar Approach, GJB Music, 1985.

[Coke64]Coker, Jerry Improvising Jazz, Prentice-Hall Inc. Englewood Cliffs, New Jersey 1964.

[Cope87]Cope, David "An Expert System for Computer-assisted Composition", Computer Music Journal, Vol11, No.4 Winter 1987.

[Dame]Dameron Tadd Lady Bird.

[Ebci84]Ebcioglu, Kemal "An expert system for Schenkerian synthesis of ...", International Computer Music Conference 1984.

[Elli]Ellington, Duke I Got it Bad.

[Elli]Ellington, Duke In a Mello Tone.

[Elli]Ellington, Duke Satin Doll.

[Fry80]Fry, C. "Computer Improvisation", Computer Music Journal, Vol. 4, No. 3, Fall 1980.

[Garn]Garner Errol Misty.

[Hear80]Hearle, Dan The Jazz Language, STUDIO224, Miami, Florida 1980.

[Hill70]Hiller, Lejaren "Music Composed with Computers- A Historical Survey" The Computer and Music, Cornell University Press, 1970.

[Jone74]Jones, George T. Music Theory, Barnes & Noble Books, 1974.

[Jone81]Jones, Kevin "Compositional Applications of Stochastic Processes", Computer Music Journal, Vol 5, No2. Smmer 1981.

[Karp85]Karpinski, Gary S. Music and Data Structures: "The Application of Music Theory in Programming Computer Assisted Instruction." Brooklyn College Conservatory of Music. 1985 Proceedings of the Fifth Annual Symposium on Small Computers in the Arts. IEEE 1985.

[Kram85]Kram, Richard "Algorithmic Composition", Temple University College of Music. 1985 Proceedings of the Fifth Annual Syumposium on Small Computers in the Arts. IEEE 1985.

[Levi81]Levitt, David, A., "A Melody Description System for Jazz Improvisation" Master's Thesis, MIT June 1981.

[Lewi]Lewis, John Afternoon In Paris.

[MLew]Lewis, Morgan How High The Moon.

[Merc]Mercer, Johnny Autumn Leaves.

[Mutc87]Mutch, Mark Expert System Development Tools for the IBM PC or Compatible, Rochester Institute of Technology, January 11, 1987.

[Road85]Roads, Curtis "Research in Music and Artificial Intelligence", Computing Surveys, Vol 17, No. 2, June 1985.

[Simo]Simons & Marks All of Me.

[Tipe87]Tipei, Sever Maiden Voyages: "A Score Produced with MP1", University of Illinois School of Music. Computer Music Journal Vol 11, No 2 Summer 1987.

Appendix A

Musical sounds can be classified as having four properties: pitch, duration, intensity, and timbre. Pitch or tone is a property perceived as a high or low sound. Duration is the length of time the sounds is present. Intensity is how loud the sound is and timbre is how it sounds (e.g. a trumpet sound versus a violin sound).

Music can be divided into three basic parts, rhythm, melody, and harmony. Rhythm is an order recurrent alternation of strong and weak elements in the flow of notes and rests in music. This thesis will use the original rhythm of the input song as the rhythm. Melody is the movement of one pitch to another involving both rhythm and pitch. Harmony is the arrangements of pitches sounded together to form chords.

In Western music, each note is assigned a name consisting of a letter A through G. A sequential series (in terms of pitch) of seven notes would each have a letter associated with it. This sequential series of notes is termed a scale. The labeling process can be done in a way such that every eighth note (termed an octave) in a sequential series repeats its label (name), but not its pitch. The spacing (in terms of pitch between the notes in the sequential series of notes is very important. The spacing between notes is termed an interval. The smallest interval between any two notes is called a half step (or minor second). The number of half steps in an octave is 12. In a well-tempered scale, an octave is exactly twice or half the original frequency depending on whether it is an octave higher or lower. Table A-1 Common names for Harmonic Intervals describes the names of the intervals between two notes based on half steps:

Half Steps/Distance	Interval
0	Unison
1	minor second(m2)
2	Major second(M2)
3	minor third(m3)
4	Major third(M3)
5	Perfect fourth(P4)
6	Augmented fourth(A4) or diminished fifth(d5) or tritone
7	Perfect fifth(P5)
8	minor sixth(m6)
9	Major sixth(M6)
10	minor seventh(m7)
11	Major seventh(M7)
12	Perfect eighth(P8) or Octave

Table A-1 Common names for Harmonic Intervals

The first note in the scale is used to name the scale. For the sequential series C D E F G A B C, the scale would be called a C scale. This terminology is incomplete, because it does not define the intervals between the notes. The interval between each note varies from a half step to a whole step (2 half steps). The placement of these half steps and whole steps within a scale defines types of scales. A major (Ionian) scale has half steps between the third and fourth notes and the seventh and eighth notes in the scale (whole steps between the other notes in the scale). Table A-2 describes the locations of half steps in some common scales.

As stated earlier, each note in a scale is given a name. The interval between each pair of notes is defined. Between A and B, C and D, D and E, F and G, and G and A there is a whole step. Between notes B and C, and notes E and F there is a half step. In a given scale there is always one of these named notes in the scale. The distance between these notes is changed by either sharping or flatting the note. The

	Scale Name(Mode)	Half step location	Quality of 7 chord
1	Ionian(major)	3 and 4; 7 and 8	Major
2	Dorian	2 and 3;6 and 7	minor
3	Phrygian	1 and 2;5 and 6	minor
4	Lydian	4 and 5;7 and 8	Major
5	Mixolydian	3 and 4;6 and 7	Dominant
6	Aeolian(rel minor)	2 and 3;5 and 6	minor
7	Locrian	1 and 2;4 and 5	half-diminished

Table A-2

addition of a sharp raises the pitch of the note by a half step and the addition of a flat lowers the pitch of the note by a half step. In a major scale, placing a flat on the third note will result in a half step from the second to the third note, but a whole step from the third to the fourth note in the scale. The symbol for flat is ^{b} and the symbol for sharp is #.

As mentioned before, chords are the arrangement of pitches or notes played simultaneously. Each note in a chord can be numbered relative to a root note in the chord. This root note is usually the lowest (in terms of pitch) in the chord. Exceptions to this will not be dealt with in this thesis. Certain arrangements of a chord have names. This arrangement of chords is much like scales in that there is a chord type and a chord name. The notes used in the chord are usually the first, third, fifth, and seventh notes of a given scale. The chords used in Western music usually deal with notes spaced from a half step to 2 whole steps. Using letters, chords are usually spaced (when ordered by pitch) such that every other letter is used. This interval is a type of third (minor third, major third, etc.). For example a C major chord would consist of the notes C, E, and G. As stated earlier a sharp or flat would change the pitch of the note that is associated with a sharp or flat. If the E in the C E G chord was made flat, the chord would be C E^{b} G. Changing the interval between the chords changes the type of chord it is. C E G is called a C major chord. C E^{b} G is

called a C minor chord. Table A-3 Common chord types shows the intervals between the notes of several chord types.

7TH chord names	Intervals based upon the Root	Intervals between notes
Major 7th	Unison(0), M3(4), P5(7), M7(11)	M3, m3, M3
Dominant 7th	Unison(0), M3(4), P5(7), m7(10)	M3, m3, m3
Minor 7th	Unison(0), m3(3), P5(7), m7(10)	m3, M3, m3
Half Diminished	Unison(0), m3(3), d5(6), m7(10)	m3, m3, M3
Diminished 7th	Unison(0), m3(3), d5(6), M6(9)	m3, m3, m3

Table A-3 Common Chord Types

There is also something termed a key. For a given key, certain notes have flats or sharps associated with them. For example, a scale produced on a major scale would indicate a half step between the third and fourth notes and the seventh and eighth notes. A G major scale would start on a G and the F (seventh note in the scale) needs to be sharped so that there is a half-step between the seventh and eighth degree of the scale.

```
c5  -- notes for the first eight measures of AutumnLeaves
1
c5
4
d4
4
e4
4
f4
4
b5
2
b5
2
b5
4
c4
4
d4
4
e4
4
a5
1
a5
4
b4
4
cs4
4
ds4
4
g4
1
r         -- a quarter rest
4
e4
4
f4
4
g4
4
-- separator is just a blank line
ami7        -- start of chord list
1
d7
1
gmaj7
1
cmaj7
1
fsmi7f5
1
b7
```

```
1
emi7
1
emi7
1
```

Glossary

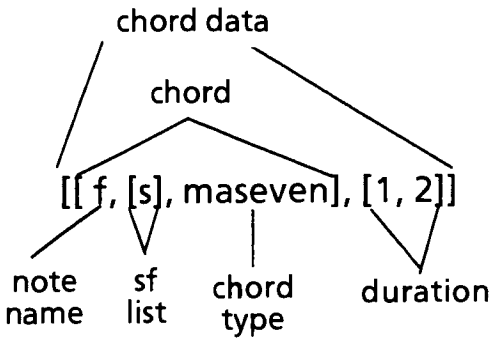
The glossary will describe terms as how they are used in the implementation. This section should be used in conjunction with the implementation section.

atom is a Prolog data type that can be either a **symbol** or a list.

bound variable is a Prolog term describing a variable that has a value associated with it.

chord data data type is a list of two lists. The first list is a **chord** data type and the second is a **duration** data type.

chord type is a **symbol**. Some valid chord types are seven, maseven, miseven... Internally the chord will be defined by the root of the chord with a **note id** and a **chord type**. Together this is called the **chord** data



type. A sample **chord** data type would be [e, [f], maseven].

degree is represented as a number, which specifies a particular position in a scale. 1 is the first and lowest degree of the scale. The highest and last degree used is 8. The first degree or one **degree** of a scale is the root note of the scale.

Duration is represented as a fraction of a whole note. A quarter note would have the value of 1/4. Internally this is represented as a list of two numbers where the first number of the list is the numerator and the second number is the denominator. The quarter note would be represented internally as [1,4].

Intervals are represented as **symbols**. The following list is a sample of some valid interval symbols: ma2, mi2, mi3, ma3,...

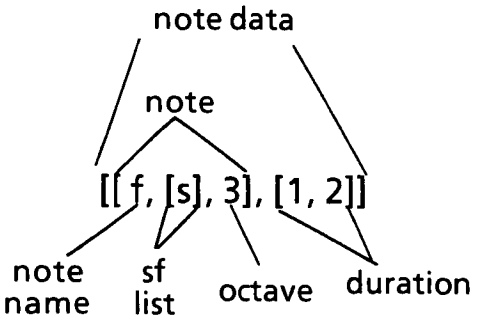
list is a Prolog data type of a group of **atoms**. Each list begins with an open bracket '[and ends with a closed bracket ']. Each atom within the list is separated by commas. A sample list is as follows: [a, b, [list, df],ewa].

mode is a **symbol** describing the notes. This thesis deals with three groups of modes. The symbols used for a major mode are as follows {ionian, dorian, phrygian, lydian, mixolydian, aeolian, locrian}. The symbols used for the harmonic minor modes are {firstHM, secondHM, thirdHM, fourthHM, fifthHM, sixthHM, seventhHM}. The melodic minor symbols are {melodicMinor, dorianF2, lydianAug, lydianF7, mixolydianF6, locrianS2, superLocrian}. Variable names will usually be in the form of Mode.

note data is a **list** of two **lists**, where the first **list** is a **note** data type and the second is a **duration** data type. A sample **note data** data type would be [[a, [], 4], [1,4]]. This is also sometimes referred to as the note info data type.

note name is a **symbol** describing the name of the note. The note name symbols are internally represented as {a, b, c, d, e, f, g}. These 7 symbols can individually represent only 7 pitches. To extend beyond this range, octaves are used.

note id is a **list** of two elements. The first being the **note name** and the second being the **sf list**.



note data type is the name associated with a complete pitch. The note data type is a list of three elements, the first is a **note** name, the second is an **sf list**, and the third is the **octave**. A sample note data type would be [f, [s], 3].

octave is a numeric number representing each group of symbols. So the first or lowest pitches are represented by a1, b1, c1, ...f1, g1, a2, b2...

progression data type is a list of four elements. The first element is a note name specifying the root, the second element is the mode, the third is the degree, and the fourth is the chord type. A sample progression data type would be [d, [], 5, seven].

progression data.type is a list of two elements where the first element is a progression data type and the second is a duration data type. A sample **progression data** data type would be [[b, [f], 1, maseven], [1,2]].

sf list is a **list** of **symbols** where the number of **symbols** describes the number of either sharps or flats associated with the note or chord name. The sharp and flat symbols are as follows {s, f}. Variable names will usually be in the form of SFList. A sample **sf** list would be [s, s].

string is either a **list** of characters or a sequence of characters contained by quotes. Sample strings would be "ab cde", ['a, 'b, 'c, 'd]...

symbol can be either a sequence of letters or numbers. If the **symbol** is a sequence of letters, the first letter must be a lower case letter. Typical symbols would be 123, abc, flat...

unbound variable is a Prolog data type that represents a variable that is not associated with any value yet.