Center for Computer Research in Music and Acoustics May 1984

Department of Music
Report No. STAN-M-19

AUTOMATIC SPECIES COUNTERPOINT

by

Bill Schottstaedt

Research sponsored by

System Development Foundation

CCRMA
DEPARTMENT OF MUSIC
Stanford University
Stanford, California 94305

Center for Computer Research in Music and Acoustics **May 1984**

Department of Music
Report No. STAN-M-19

AUTOMATIC SPECIES COUNTERPOINT

by

Bill Schottstaedt

Species counterpoint as presented by J. J. Fux in *Gradus Ad Parnassum* appears to be a ready made case for a rul based "expert system". In programs of this sort, knowledge is encoded as a list of IF..THEN statements. These attributes can easily be defined in such a manner that a computer program can use them to find acceptable solutions to species counterpoint problems. In this paper we present a program that can write counterpoint of this sort. Rather than get bogged down in circumlocutions, we provide the actual code, and the reader can if he so desires re-implement the entire program. The language used is SAIL, an Algol-like language fully described elsewhere. The original implementation was done in Pla, an offshoot of SAIL also described elsewhere. We translated everything to SAIL to speed up execution. This file is itself an executable program which solves species counterpoint problems as described in the text.

This research was supported by the System Development Foundation under Grant SDF #346. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, any agency of the U. S. Government, or of sponsoring foundations.

Table of Contents

Automatic Species Counterpoint	1
Assumptions	1
Basic Definitions	2
Data Representation	5
Basic Rules	10
Species Definition	18
Multi-Part Counterpoint	22
Searching Methods	25
Acknowledgments	35

Automatic Species Counterpoint

Species counterpoint as presented by J. J. Fux in "Gradus Ad Parnassum" appears to be a ready made case for a rule based "expert system". In programs of this sort, knowledge is encoded as a list of IF..THEN statements. Obvious examples from Fux are:

IF Melodic-Leap > Octave THEN Try-Something-Else.

or

IF Interval-With-Bass = Fourth AND Species = First-Species
THEN Try-Something-Else.

These attributes can easily be defined in such a manner that a computer program can use them to find acceptable solutions to species counterpoint problems. In this paper we present a program that can write counterpoint of this sort. Rather than get bogged down in circumlocutions, we provide the actual code, and the reader can if he so desires re-implement the entire program. The language used is SAIL, an Algol-like language fully described elsewhere. The original implementation was done in Pla, an offshoot of SAIL also described elsewhere. We translated everything to SAIL to speed up execution. This file is itself an executable program which solves species counterpoint problems as described in the text.

Assumptions

The octave is divided into 12 semitones. We can therefore give a unique integer to every possible pitch. In our examples, this value is the distance in semitones from the given pitch to low C (16 Hz).

Rhythms can be represented as a certain number of eighthnotes.

Notes within a voice do not overlap. Each note's onset time can therefore be represented as a certain number of eighthnotes from time 0. Of course a prior assumption is that there are things called voices that can be assigned a unique succession of notes. Each note has a duration, an onset time, and a pitch. If more than one voice exists, there is also a notion that a "vertical" interval exists between these voices equivalent to the melodic interval necessary to jump from one voice to the other.

The cantus firmus and the starting note of each counterpoint voice are specifed by hand. We also currently assume that the cantus firmus is entirely in whole notes, but it would not be difficult to remove this restriction.

Basic Definitions

We must first define our terms. These include the interval names, consonance and dissonance classifications, and so on. To define the intervals:

```
;
                                              BEGIN "foox"
                                              REQUIRE 20000 SYSTEM_PDL;
                                              DEFINE !="COMMENT";
  DEFINE Unison=       0,
         MinorSecond=1,
         MajorSecond=2,
         MinorThird= 3,
         MajorThird= 4,
         Fourth=      5,
         Tritone=     6,
         Fifth=       7,
         MinorSixth= 8,
         MajorSixth= 9,
         MinorSeventh=10,
         MajorSeventh=11,
         Octave=     12;
!
```

Each interval is defined by the number of semitones within it. These intervals are impervious to octave interpolations and transpositions.

We classify each interval as either a perfect consonance, imperfect consonance, or dissonance:

```
;
  BOOLEAN ARRAY PerfectConsonance[0:12];          SIMPLE PROCEDURE PerfInit; BEGIN
  PerfectConsonance[Unison]←TRUE;
  PerfectConsonance[Fifth]←TRUE;
  PerfectConsonance[Octave]←TRUE;                 END; REQUIRE PerfInit INITIALIZATION;

  BOOLEAN ARRAY ImperfectConsonance[0:12];        SIMPLE PROCEDURE ImperfInit; BEGIN
  ImperfectConsonance[MinorThird]←TRUE;
  ImperfectConsonance[MajorThird]←TRUE;
  ImperfectConsonance[MinorSixth]←TRUE;
  ImperfectConsonance[MajorSixth]←TRUE;           END; REQUIRE ImperfInit INITIALIZATION;

  BOOLEAN ARRAY Dissonance[0:12];                 SIMPLE PROCEDURE DisInit; BEGIN
  Dissonance[MinorSecond]←TRUE;
  Dissonance[MajorSecond]←TRUE;
  Dissonance[MinorSeventh]←TRUE;
  Dissonance[MajorSeventh]←TRUE;
  Dissonance[Fourth]←TRUE;
  Dissonance[Tritone]←TRUE;                       END; REQUIRE DisInit INITIALIZATION;
!
```

To ascertain whether a given interval is a dissonance, we merely use that interval as an index into the *Dissonance* array. The other arrays are used similarly. Under certain circumstances, the fourth is considered an imperfect consonance, but we will leave that complication until later. (In SAIL all **Boolean** variables are initialized to **False**, so we need only set the ones that should be **True**. These assignments take place at initialization time before the main program is executed).

We must also define the basic scales used in species counterpoint. The representation chosen here takes advantage of the fact that the scales are only an octave in extent, so octave transpositions can be removed before a decision is made as to whether a given pitch is in a mode. However, we do have to take into account transpositions by some interval other than an octave. In the definition given here, we assume that the tranposition (if any) has also been removed – from this code's viewpoint every mode starts on pitch 0 and has only 12 pitches to choose from. Any pitch that is in the mode has a value of 1 in the corresponding mode array. A value of 0 means that the given pitch is illegal (or at least a chromatic alteration) in that mode.

```
;
    DEFINE  Aeolian=1,Dorian=2,Phrygian=3,Lydian=4,Mixolydian=5,Ionian=6,Locrian=7;

    PRESET_WITH 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1; INTEGER ARRAY _Ionian[0:11];
    PRESET_WITH 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0; INTEGER ARRAY _Dorian[0:11];
    PRESET_WITH 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0; INTEGER ARRAY _Phrygian[0:11];
    PRESET_WITH 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1; INTEGER ARRAY _Lydian[0:11];
    PRESET_WITH 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0; INTEGER ARRAY _Mixolydian[0:11];
    PRESET_WITH 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0; INTEGER ARRAY _Aeolian[0:11];
    PRESET_WITH 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0; INTEGER ARRAY _Locrian[0:11];

    BOOLEAN PROCEDURE InMode(INTEGER Pitch,Mode);
    BEGIN
    INTEGER Pit;
    Pit←Pitch MOD 12;   ! remove any lingering octave information;
    CASE Mode OF
      BEGIN
      [Ionian]     RETURN(_Ionian[Pit]);
      [Aeolian]    RETURN(_Aeolian[Pit]);
      [Dorian]     RETURN(_Dorian[Pit]);
      [Phrygian]   RETURN(_Phrygian[Pit]);
      [Lydian]     RETURN(_Lydian[Pit]);
      [Mixolydian] RETURN(_Mixolydian[Pit]);
      [Locrian]    RETURN(_Locrian[Pit])
      END;
    END;
!
```

Although defined here, the Ionian and Locrian modes have not actually been used during testing and may contain unnoticed bugs.

Certain intervals are not allowed melodically because they are considered hard to sing:

```
;
    BOOLEAN ARRAY BadMelodyInterval[0:12];            SIMPLE PROCEDURE BadInit; BEGIN
    BadMelodyInterval[Tritone]←TRUE;
    BadMelodyInterval[MajorSixth]←TRUE;
    BadMelodyInterval[MinorSeventh]←TRUE;
    BadMelodyInterval[MajorSeventh]←TRUE;             END; REQUIRE BadInit INITIALIZATION;
!
```

Other "bad" intervals such as the augmented second can be avoided by forbidding chromatic alterations within the mode. The latter are necessary unfortunately at cadences, so more elaborate checks will be made later. We must also avoid a leap greater than an octave and a leap of a minor sixth down. The leap of a minor sixth up is acceptable.

```
;
    BOOLEAN PROCEDURE BadMelody(INTEGER Intv);
    RETURN( ABS(Intv)>Octave OR                 ! more than an octave either way;
            BadMelodyInterval[ABS(Intv)] OR     ! tritone or major sixth or seventh;
            Intv=-MinorSixth);                  ! downward minor sixth;
!
```

We will often need to distinguish a "skip" (or leap) from a "step". These are not explicitly defined in my version of Fux, but it is clear that a skip is any interval greater than a major second, and that a step is an interval that is not a skip and also not a unison.

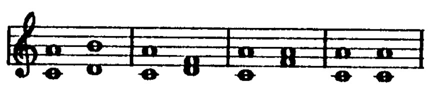
```
;
    BOOLEAN PROCEDURE ASkip(INTEGER Interval);
    RETURN(ABS(Interval)>MajorSecond);

    BOOLEAN PROCEDURE AStep(INTEGER Interval);
    RETURN((ABS(Interval)=MinorSecond) OR (ABS(Interval)=MajorSecond));
!
```

The intervals are further divided up into the groups second, third, sixth, seventh, and so on. We therefore define the obvious procedures:

```
;
    BOOLEAN PROCEDURE AThird(INTEGER Interval);
    RETURN((Interval=minorThird) OR (Interval=MajorThird));

    BOOLEAN PROCEDURE ASeventh(INTEGER Interval);
    RETURN((Interval=minorSeventh) OR (Interval=MajorSeventh));

    BOOLEAN PROCEDURE AnOctave(INTEGER interval);
    RETURN((Interval≠Unison) AND ((ABS(Interval) MOD 12)=0));

    BOOLEAN PROCEDURE ATenth(INTEGER Interval);
    RETURN((ABS(Interval)>14) AND AThird((ABS(Interval) MOD 12)));
!
```

The last procedure (*ATenth*) is used to implement only one rather obscure rule.

Now we define the four kinds of "motion": direct, oblique, contrary, and no motion:

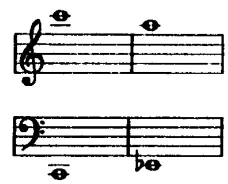
```
;
    DEFINE  DirectMotion=1,
            ContraryMotion=2,
            ObliqueMotion=3,
            NoMotion=4;

    INTEGER PROCEDURE MotionType(INTEGER Pitch1,Pitch2,Pitch3,Pitch4);
    COMMENT Pitch1 and Pitch2 are from Voice 1, Pitch3 and Pitch4 from voice 2.
            That is the passage in question is:

                Voice1:   Pitch1   Pitch2
                Voice2:   Pitch3   Pitch4

            This procedure decides which kind of motion these pitches indicate.
            Either voice may go up (Pitch2>Pitch1), down (Pitch2<Pitch1),
            or hold its pitch (Pitch1=Pitch2).
            ;
    IF (Pitch1=Pitch2 OR Pitch3=Pitch4)         ! at least one voice does not move;
      THEN
        IF (Pitch1=Pitch2 AND Pitch3=Pitch4)    ! neither voice moves;
          THEN RETURN(NoMotion)
        ELSE RETURN(ObliqueMotion)
    ELSE
      IF ((Pitch2-Pitch1)*(Pitch4-Pitch3)>0)    ! either both >0 or both <0;
        THEN RETURN(DirectMotion)
    ELSE RETURN(ContraryMotion);
!
```

Direct Motion   Contrary Motion   Oblique Motion   No Motion

Two other procedures will be useful later when we apply rules involving direct motion to a perfect consonance and consecutive leaps:

```
;
    BOOLEAN PROCEDURE DirectMotionToPerfectConsonance(
        INTEGER Pitch1,Pitch2,Pitch3,Pitch4);
    RETURN(PerfectConsonance[ABS(Pitch4-Pitch2) MOD 12] AND
          MotionType(Pitch1,Pitch2,Pitch3,Pitch4)=DirectMotion);
```

```
BOOLEAN PROCEDURE ConsecutiveSkipsInSameDirection(
    INTEGER Pitch1,Pitch2,Pitch3);
RETURN( (((Pitch1>Pitch2) AND (Pitch2>Pitch3)) OR
        ((Pitch1<Pitch2) AND (Pitch2<Pitch3))) AND
        ASkip(Pitch2-Pitch1) AND    ! first interval is a skip;
        Askip(Pitch3-Pitch2));      ! so is second;
!
```

The extreme high and low pitches in a melody define its overall range. Fux emphasizes on many occasions that this range should be constrained to fit human vocal ranges.

```
;
DEFINE  HighestSemitone=72;          ! this is a high C (1025 Hz);
DEFINE  LowestSemitone=24;           ! this is a low C (65 Hz);
                                     ! these are liberal ranges;
BOOLEAN PROCEDURE OutOfRange(INTEGER Pitch);
RETURN((Pitch>HighestSemitone) OR (Pitch<LowestSemitone));

BOOLEAN PROCEDURE ExtremeRange(INTEGER Pitch);
RETURN(Pitch>(HighestSemitone-3) OR Pitch<(LowestSemitone+3));
!
```

These procedures give a rather liberal range check. We also need code to find the overall melody range so that it normally does not exceed an octave and a fifth. This code, however, depends on the representation we choose for voice data.

Data Representation

For the purposes of this description we will use a simple representation which uses parallel arrays for onset time, duration, and pitch. First we have integers for the transposition (*BasePitch*), the current mode (*Mode*), and the length of the cantus firmus in eighth notes (*TotalTime*).

```
;
DEFINE MostNotes=128;           ! most notes any one voice can have;
DEFINE MostVoices=8;

INTEGER BasePitch,Mode,TotalTime;
!
```

We arbitrarily limit the longest melody to 128 notes and the number of voices to 8 (once again, these are purely for array allocation purposes and are not built into any of the counterpoint writing procedures). We need a place to hold the pitch, duration, and onset time data for each voice, including the cantus firmus.

```
;
INTEGER ARRAY CtrPt,Onset,Dur[1:MostNotes,0:MostVoices];
INTEGER ARRAY TotalNotes[0:MostVoices];
!
```

The cantus firmus is voice 0 in this scheme, but is not otherwise distinguished from the other voices. Each note of voice V keeps its note information for note number N in these three parallel arrays. *OnSet*[V,N] is the begin time in eighth notes of the note, *Dur*[V,N] is its duration (in eighth notes), and *CtrPt*[V,N] is the pitch of the note (in semitones above the low c). Next we need a variety of procedures to access the data in these arrays.

```
;
    INTEGER PROCEDURE Us(INTEGER n,v);
    RETURN(Ctrpt[n,v]);               ! returns pitch of note N in voice V;

    BOOLEAN PROCEDURE LastNote(INTEGER n,v);
    RETURN(n=TotalNotes[v]);          ! returns true if N is the last note of V;

    BOOLEAN PROCEDURE FirstNote(INTEGER n,v);
    RETURN(n=1);                      ! returns true if N is the first note;

    BOOLEAN PROCEDURE NextToLastNote(INTEGER n,v);
    RETURN(n=(TotalNotes[v]-1));      ! returns true if N is the next to last note;
                                      ! these are used mainly for cadential formulas;
    PROCEDURE SetUs(INTEGER n,p,v);
    Ctrpt[n,v]←p;                     ! sets the pitch of note N in voice V to P;
!
```

We also need a procedure that returns the total range of the voice *V* at the note *CN* (counting from the beginning of the melody), given the current pitch *CP*.

```
;
    INTEGER PROCEDURE TotalRange(INTEGER Cn,Cp,v);
    BEGIN                        ! return total range of melody so far (including CP);
    INTEGER Minp,Maxp,i;
    MinP←Cp;
    Maxp←Cp;
    FOR i←1 STEP 1 UNTIL Cn-1 DO
      BEGIN
      MinP←MinP MIN Us(i,v);
      MaxP←MaxP MAX Us(i,v);
      END;
    RETURN(MaxP-MinP);
    END;
!
```

We present most of the data analysis procedures in this form. In words, we have a proposed pitch (*CP*) in a voice (*V*) at the point (*CN*) in that voice, and need a decision about the acceptability of that pitch.

Since we assume for simplicity's sake that the cantus firmus is always in whole notes, it is easy to write a procedure which returns the cantus firmus note at any given eighth note beat *N*.

```
;
    INTEGER PROCEDURE Cantus(INTEGER n,v);
    RETURN(CtrPt[((Onset[n,v] DIV 8)+1),0]);
!
```

A similar procedure is needed to return the current pitch in some other voice (we could use this procedure for the cantus firmus also because it is just voice 0, but as long as we are assuming the cantus moves in whole notes we can handle it separately to save some time):

```
;
    INTEGER PROCEDURE VIndex(INTEGER Time,VNum);
    BEGIN
    INTEGER i;
    FOR i←1 STEP 1 UNTIL TotalNotes[VNum] DO
      IF Onset[i,vNum]≤Time AND Onset[i,vNum]+Dur[i,vNum]>Time THEN DONE;
    RETURN(i);
    END;
    INTEGER PROCEDURE Other(INTEGER Cn,v,v1);
    RETURN(Ctrpt[VIndex(Onset[Cn,v],v1),v1]);

    INTEGER PROCEDURE Bass(INTEGER Cn,v);
    BEGIN
    INTEGER j,LowestPitch;
    LowestPitch←Cantus(Cn,v);
    FOR j←1 STEP 1 UNTIL v-1 DO LowestPitch←LowestPitch MIN Other(Cn,v,j);
    RETURN(LowestPitch);
    END;
!
```

This is not an optimal search procedure, but normally there are not too many notes in a melody. We can therefore search laboriously from the beginning for the note in voice *V1* that sounds at the same time as the note *CN* in voice *V*.

We must also define various rhythmic values:

```
;
    DEFINE  WholeNote=          8,
            HalfNote=           4,
            DottedHalfNote=     6,
            QuarterNote=        2,
            EighthNote=         1;
!
```

A measure contains 8 eighth notes (triple time is not supported).

```
;
    INTEGER PROCEDURE Beat8(INTEGER n);
    RETURN(n MOD 8);                    ! 0=first beat,7=last;
!
```

A downbeat occurs on the first (0-th) beat of the measure.

```
;
    BOOLEAN PROCEDURE DownBeat(INTEGER n,v);
    RETURN(Beat8(Onset[n,v])=0);
!
```

An upbeat can be considered to be any beat that is not a downbeat. There are several cases where beat 4 (the second half note beat) is a downbeat, but these will be handled separately.

```
;
    BOOLEAN PROCEDURE UpBeat(INTEGER n,v);
    RETURN(NOT DownBeat(n,v));
!
```

Fux mentions that it is desirable to maintain "variety" in the melodies. This variety seems to entail a mix of melodic intervals coupled with an avoidance of too many repetitions of any one pitch. To check the latter condition we define a procedure which looks for repeated pitches within a melody:

```
;
    INTEGER PROCEDURE PitchRepeats(INTEGER Cn,Cp,V);
    BEGIN
    INTEGER i,k;
    i←0;
    FOR k←1 STEP 1 UNTIL Cn-1 DO IF Us(k,v)=Cp THEN i←i+1;
    RETURN(i);
    END;
!
```

We also need to encourage the melody to contain a nice mixture of intervals (in third and fifth species). We keep track of how many times a given interval type has been used so far, then check to see if others have been used nearly as many times. The interval type ignores distinctions of major and minor, but does not ignore direction (a rising minor second is considered the same interval type as a rising major second, but not the same as a descending minor second).

;

```
DEFINE  One=0, Two=2, Three=3, Four=4, Five=5, Six=6, Eight=8;

INTEGER PROCEDURE Size(INTEGER MelInt);
BEGIN
INTEGER ActInt,IntTyp;
ActInt←ABS(MelInt);
CASE ActInt OF
  BEGIN
  [unison]                      IntTyp←One;
  [minorSecond] [majorSecond]   IntTyp←Two;
  [minorThird] [majorThird]     IntTyp←Three;
  [fourth]                      IntTyp←Four;
  [fifth]                       IntTYp←Five;
  [minorsixth]                  IntTyp←Six;
  [octave]                      IntTyp←Eight;
  ELSE PRINT("illegal melodic interval: ",MelInt)
  END;
RETURN(IF MelInt>0 THEN IntTyp ELSE -IntTyp);
END;

BOOLEAN PROCEDURE TooMuchOfInterval(INTEGER Cn,Cp,v);
BEGIN
INTEGER ARRAY Ints[-8:8];
INTEGER i,k,MinL;
ARRCLR(Ints);
FOR i←2 STEP 1 UNTIL Cn-1 DO
  BEGIN
  k←Size(CtrPt[i,v]-CtrPt[i-1,v]);
  Ints[k]←Ints[k]+1;
  END;
k←Size(Cp-Ctrpt[Cn-1,v]);
MinL←-8;
FOR i←-7 STEP 1 UNTIL 8 DO
  IF i≠k AND Ints[i]>Ints[MinL] THEN MinL←i;
RETURN(Ints[k]>(Ints[MinL]+6));
END;
```

!

Next we must define when a dissonance is legal in each species, and also when chromatic (altered) notes occur. The rules governing the cadential formulas are pretty clear, but the rules governing ficta seem to consist mostly of handwaves and exceptions. We first define dissonance handling in each species:

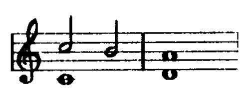
;

```
BOOLEAN PROCEDURE ADissonance(INTEGER Interval,Cn,Cp,v,Species);
IF Species=1 OR Dur[Cn,v]=wholenote THEN RETURN(Dissonance[Interval])
```

!

In first species no dissonances are allowed, so we simply return true in every case in which *Interval* is a dissonance. Similarly, we do not allow wholenotes to form a dissonance (in fifth species).

;

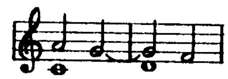
```
ELSE
IF Species=2
  THEN
    IF (DownBeat(Cn,v) OR (NOT (Astep(Cp-Us(Cn-1,v)))))
      THEN RETURN(Dissonance[Interval])
    ELSE RETURN(FALSE)
```

Passing Tone

!

In second species, a dissonance is allowed only as a passing tone on an upbeat. Since we don't yet know what the next note will be, we must put off the rest of the dissonance check until later.

Cambiata

```
;
    ELSE
    IF Species=3
      THEN
        BEGIN
        INTEGER MelInt;
        IF Beat8(OnSet[Cn,v])=0 OR FirstNote(Cn,v) OR LastNote(Cn,v)
          THEN RETURN(Dissonance[Interval]);
        MelInt←Cp-Us(Cn-1,v);
        IF (NOT AStep(MelInt)) THEN RETURN(Dissonance[Interval]);
        COMMENT
                0 cannot be dissonant (downbeat)
                1 can be if passing either way, but must be approached by step.
                2 can be if passing 2 to 4 (both latter cons)
                3 can be if passing but must be approached and left by step
        ;
        RETURN(FALSE);
        END
!
```

In third species, the downbeat cannot be a dissonance, but any of the other beats can if they are passing tones or cambiatas. As with second species, the dissonance resolution is checked later.

Suspension

```
;
    ELSE
    IF Species=4
      THEN
        BEGIN
        INTEGER MelInt;
        IF UpBeat(Cn,v) OR FirstNote(Cn,v) OR LastNote(Cn,v)
          THEN RETURN(Dissonance[Interval]);
        MelInt←Cp-Us(Cn-1,v);
        IF MelInt=0 THEN RETURN(Dissonance[Interval]);
        RETURN(FALSE);                    ! i.e. unison to downbeat is ok, but needs check later;
        END
!
```

In fourth species, a dissonance is legal as a suspension. The suspension is handled here as a note repeated from the upbeat to the downbeat (consider it tied across the bar), so the melodic interval is obviously a unison.

```
;
    ELSE
    IF Species=5
      THEN
        BEGIN
        IF Beat8(Onset[Cn,v])=0          ! first note;
          THEN                           ! so can be tied from previous to diss now;
            IF Cp=Us(Cn-1,v)             ! is a unison (assume tie);
              THEN RETURN(FALSE)         ! so dissonance is ok here (check resolution later);
            ELSE RETURN(Dissonance[Interval])
        ELSE                             ! not first beat;
          IF NOT AStep(Cp-Us(Cn-1,v))    ! cannot jump to dissonance;
            THEN RETURN(Dissonance[Interval]);
        RETURN(FALSE);
        END;
!
```

Finally, in fifth species we have passing note dissonances, cambiatas, and suspensions. Therefore, if the downbeat (beat 0) of the measure is dissonant, the note must have been tied across the bar (the melodic interval must be a unison). On any other beat the dissonance must be approached by step. We will check later for its proper resolution.

Lastly, in multi-part counterpoint we often need to check for various kinds of pitch doublings:

```
;
    BOOLEAN PROCEDURE Doubled(INTEGER Pitch,Cn,v);
    BEGIN
    INTEGER vNum;
    FOR Vnum←0 STEP 1 UNTIL v-1 DO
      IF (Other(Cn,v,VNum) MOD 12)=Pitch THEN RETURN(TRUE);
    RETURN(FALSE);
    END;
```

Basic Rules

The basic rules of species counterpoint describe how melodies are formed and combined. In most cases any given rule can be broken if other factors necessitate it. Fux repeatedly presents the rules as guidelines, not absolutes. In our implementation we define the relative importance of the rules by assigning each rule a penalty. The higher the penalty, the worse it is to break the associated rule. The relations between these penalties determine what kind of solutions the program will find.

Our penalty types and the associated penalty values are:

```
;
    DEFINE Infinity="2↑18";
    DEFINE Bad!=100;
    DEFINE RealBad!=200;
    DEFINE  UnisonPenalty                          =bad!,
            DirectToFifthPenalty                   =Realbad!,
            DirectToOctavePenalty                  =RealBad!,
            ParallelFifthPenalty                   =infinity,
            ParallelUnisonPenalty                  =infinity,
            EndOnPerfectPenalty                    =infinity,
            NoLeadingTonePenalty                   =infinity,
            DissonancePenalty                      =infinity,
            OutOfRangePenalty                      =realBad!,
            OutOfModePenalty                       =infinity,
            TwoSkipsPenalty                        =1,
            DirectMotionPenalty                    =1,
            PerfectConsonancePenalty               =2,
            CompoundPenalty                        =1,
            TenthToOctavePenalty                   =8,
            SkipTo8vePenalty                       =8,
            SkipFromUnisonPenalty                  =4,
            SkipPrecededBySameDirectionPenalty     =1,
            FifthPrecededBySameDirectionPenalty    =3,
            SixthPrecededBySameDirectionPenalty    =8,
            SkipFollowedBySameDirectionPenalty     =3,
            FifthFollowedBySameDirectionPenalty    =8,
            SixthFollowedBySameDirectionPenalty    =34,
            TwoSkipsNotInTriadPenalty              =3,
            BadMelodyPenalty                       =infinity,
            ExtremeRangePenalty                    =5,
            LydianCadentialTritonePenalty          =13,
            UpperNeighborPenalty                   =1,
            LowerNeighborPenalty                   =1,
            OverTwelfthPenalty                     =infinity,
            OverOctavePenalty                      =bad!,
            SixthLeapPenalty                       =2,
            OctaveLeapPenalty                      =5,
            BadCadencePenalty                      =infinity,
            DirectPerfectOnDownbeatPenalty         =infinity,
            RepetitionOnUpbeatPenalty              =bad!,
            DissonanceNotFillingThirdPenalty       =infinity,
            UnisonDownbeatPenalty                  =3,
            TwoRepeatedNotesPenalty                =2,
            ThreeRepeatedNotesPenalty              =4,
            FourRepeatedNotesPenalty               =7,
```

```
LeapAtCadencePenalty                    =13,
NotaCambiataPenalty                     =infinity,
NotBestCadencePenalty                   =8,
UnisonOnBeat4Penalty                    =3,
NotaLigaturePenalty                     =21,
UnresolvedLigaturePenalty               =infinity,
NoTimeForaLigaturePenalty               =infinity,
EighthJumpPenalty                       =bad!,
HalfUntiedPenalty                       =13,
UnisonUpbeatPenalty                     =21,
MelodicBoredomPenalty                   =1,
SkipToDownBeatPenalty                   =1,
ThreeSkipsPenalty                       =3,
DownBeatUnisonPenalty                   =bad!,
VerticalTritonePenalty                  =2,
MelodicTritonePenalty                   =8,
AscendingSixthPenalty                   =1,
RepeatedPitchPenalty                    =1,
NotContraryToOthersPenalty              =1,
NotTriadPenalty                         =34,
InnerVoicesInDirectToPerfectPenalty =21,
InnerVoicesInDirectToTritonePenalty =13,
SixFiveChordPenalty                     =infinity,
UnpreparedSixFivePenalty                =bad!,
UnresolvedSixFivePenalty                =bad!,
AugmentedIntervalPenalty                =infinity,
ThirdDoubledPenalty                     =5,
DoubledLeadingTonePenalty               =infinity,
DoubledSixthPenalty                     =5,
DoubledFifthPenalty                     =3,
TripledBassPenalty                      =3,
UpperVoicesTooFarApartPenalty           =1,
UnresolvedLeadingTonePenalty            =infinity,
AllVoicesSkipPenalty                    =8,
DirectToTritonePenalty                  =bad!,
CrossBelowBassPenalty                   =infinity;
!
```

These penalty values have come partly from Fux's comments (the more important rules get a higher penalty), and partly from experience running the program. Obviously not all the penalties apply to every species of counterpoint. The penalties are all positive (that is, there are no rewards), because we want to be able to abandon a line as soon as its penalty gets too high. This implies that the penalty function should never descend. Even the smaller penalties have a profound effect on the outcome of the counterpoint search. The solver is consistently "bent" in the direction determined by the penalties causing the music to reflect the slightest changes in that direction.

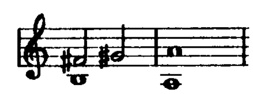
We need a procedure which examines the current note of the current voice and returns a measure of its goodness (the lower the penalty, the better the note).

```
;
  FORWARD INTEGER PROCEDURE SpecialSpeciesCheck(
      INTEGER Cn,Cp,v,Other0,Other1,Other2,NumParts,
          Species,MelInt,Interval,ActInt,LastIntClass,Pitch,LastMelInt);
  FORWARD INTEGER PROCEDURE OtherVoiceCheck(
      INTEGER Cn,Cp,v,v1,Species);
!
```

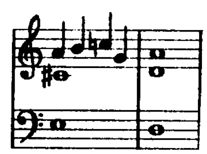
We define the special attributes of each species later. These are checked by the SpecialSpeciesCheck procedure (declared here for convenience, but defined later to make the presentation more clear).

```
;
    INTEGER PROCEDURE Check(INTEGER Cn,Cp,v,NumParts,Species);
    BEGIN       ! return penalty associated with pitch CP in voice V at note CN;
    INTEGER Val,                    ! accumulated penalty for CP;
            i,k,                    ! temp counters;
            Interval,               ! current pitch to other voice current pitch;
            IntClass,               ! octaves and direction removed from Interval;
            Pitch,                  ! CP pitch class;
            LastIntClass,           ! previous such interval;
            MelInt,                 ! melodic interval in current voice;
            LastMelInt,             ! last melodic interval in voice;
            Other0,                 ! current other voice pitch;
            Other1,                 ! previous other voice pitch;
            Other2,                 ! pitch 2 back in other voice;
            Cross;                  ! voice cross counter;
    BOOLEAN SameDir;                ! did last 2 melodic intervals go same direction;
    IF v=1
      THEN
        BEGIN
        Other0←Cantus(Cn,v);        ! current pitch in cantus;
        Other1←Cantus(Cn-1,v);
        IF Cn>2 THEN Other2←Cantus(Cn-2,v);
        END
    ELSE
        BEGIN
        Other0←Bass(Cn,v);          ! check current against bass line;
        Other1←Bass(Cn-1,v);
        IF Cn>2 THEN Other2←Bass(Cn-2,v);
        END;
    Val←0;                          ! accumulated penalty;
    Interval←Cp-Other0;             ! vertical interval between current pitch and cantus;
    IntClass←ABS(Interval) MOD 12;  ! remove direction and octave info;
    MelInt←(Cp-Us(Cn-1,v));
    Pitch←CP MOD 12;
!
```

The rules are applied here in an order which hopes to reduce redundant checks of various kinds. At the start we apply rules that apply to every pitch in the melodies.

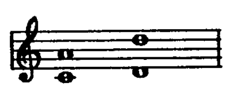
```
;
    ! melody must stay in range;
    IF OutOfRange(Cp+BasePitch) THEN Val←Val+OutOfRangePenalty;

    ! extremes of range are also bad (to be avoided);
    IF ExtremeRange(Cp+BasePitch) THEN Val←Val+ExtremeRangePenalty;

    ! Chromatically altered notes are accepted only at the
    ! cadence.  Other alterations (such as ficta) will be handled later);
    IF NOT NextToLastNote(Cn,v)
      THEN
        BEGIN
        IF (Species=2)
          THEN
            IF NOT InMode(Pitch,Mode)
              THEN Val←Val+OutOfModePenalty
            ELSE
        ELSE
          IF (Cn=TotalNotes[v]-2) OR (Mode=Aeolian) OR (Cp≤Other0) OR (Intclass=Fifth)
            THEN
              IF NOT InMode(Pitch,Mode)
                THEN Val←Val+OutOfModePenalty;
        END
    ELSE
        BEGIN
        BOOLEAN WeHaveARealLeadingTone;
        WeHaveARealLeadingTone←
                (Pitch=11) OR ((Pitch=10) AND (Mode=Phrygian));
!
```

Aeolian Cadence
in Second Species

The leading tone cannot be doubled, nor should the raised leading tone be preceded by its unraised form. The next to last chord of the counterpoint must have a leading tone somewhere.

```
;
      IF WeHaveARealLeadingTone
        THEN
          IF Doubled(Pitch,Cn,v)      ! leading tone doubled;
            THEN Val←Val+DoubledLeadingTonePenalty
          ELSE
      ELSE
        IF (Pitch=10)                 ! unraised leading tone, but need raised form;
          THEN Val←Val+BadCadencePenalty
        ELSE                          ! are we last -- does anyone have one;
          IF NOT InMode(Pitch,Mode)   ! maybe not the leading tone at all;
            THEN Val←Val+OutOfModePenalty
          ELSE
            IF v=Numparts             ! last voice -- check that someone has it;
              THEN
                IF NOT Doubled(11,Cn,v) AND
                   NOT Doubled(10,Cn,v)
                  THEN Val←Val+NoLeadingTonePenalty;
      END;
                                             IF Val≥infinity THEN RETURN(val);

    IF Cn>2
      THEN
        BEGIN
        LastMelInt←(Us(Cn-1,v)-Us(Cn-2,v));
        SameDir←((MelInt*LastMelInt)≥0);
        END;
    IF Cn>1 THEN LastIntClass←(ABS(Us(Cn-1,v)-Other1) MOD 12);
    ! dissonance must be handled correctly.  The procedure
    ! ADISSONANCE returns true if a CP forms a dissonance that is
    ! not correctly prepared.  Other rules later check that it is
    ! resolved correctly;
    IF ADissonance(IntClass,Cn,Cp,v,Species) THEN Val←Val+DissonancePenalty;

                                             IF Val≥infinity THEN RETURN(val);
    Val←Val+SpecialSpeciesCheck(Cn,Cp,v,Other0,Other1,Other2,NumParts,
            Species,MelInt,Interval,intClass,LastIntClass,Pitch,LastMelInt);
    IF v>1 THEN Val←Val+OtherVoiceCheck(Cn,Cp,v,NumParts,species);
!
```

This procedure call looks at the current state from the point of view of the current species. There are a number of rules unique to each species.

```
;
    IF FirstNote(Cn,v) THEN RETURN(Val);
                                   ! no further rules apply to first note;
                                             IF Val≥infinity THEN RETURN(val);
!
```

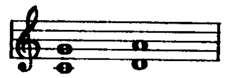
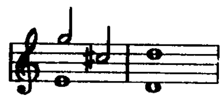
The "fundamental rule" defines which intervallic relationships are acceptable between two melodies. Fux presents this as four rules, but as Martini remarks, these four are actually reducible to one – voices cannot move by direct motion to a perfect consonance. In multi-voice counterpoint, this rule can be broken, especially at cadences.

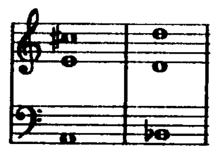


```
;
    ! direct motion to perfect consonances considered harmful;
    IF NOT LastNote(Cn,v) OR NumParts=1
      THEN
        IF DirectMotionToPerfectConsonance(Us(Cn-1,v),Cp,Other1,Other0)
          THEN                        ! we do have direct motion to perfect;
            IF IntClass=unison        ! reached an octave or unison;
              THEN Val←Val+DirectToOctavePenalty
            ELSE Val←Val+DirectToFifthPenalty;
!
```

The expression "NumParts=1" checks that we are indeed in a multi-voice situation. In two part counterpoint we do not allow direct motion to a perfect consonance at all.

```
;
    ! check for more blatant examples of the same error;
    IF IntClass=fifth AND LastIntClass=Fifth
      THEN Val←Val+ParallelFifthPenalty;

    IF IntClass=unison AND LastIntClass=unison
      THEN Val←Val+ParallelUnisonPenalty;
                                 ! voices parallel in octave or fifths;
                                 ! which Fux seems to consider to be worse;
                                 ! than just direct motion to these;
                                 IF Val≥infinity THEN RETURN(val);
    ! certain melodic intervals are disallowed;
    IF BadMelody(MelInt) THEN Val←Val+BadMelodyPenalty;
                                           IF Val≥infinity THEN RETURN(val);
!
```

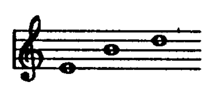


This rule (the meodic interval check) is actually built into the search procedures given below so its inclusion here is mostly for completeness.

```
;
    ! must end on unison or octave in two parts, fifth and major third
    ! allowed in 3 and 4 part writing;
    IF (LastNote(Cn,v)) AND          ! it is the last note;
       (IntClass≠unison)             ! must be octave or unison here (2 parts);
      THEN
        IF NumParts=1 OR Interval<0 ! two part writing or bass;
          THEN Val←Val+EndOnPerfectPenalty
        ELSE                         ! NumParts>1 so we must have 3 or more part writing;
          IF Intclass≠fifth AND Intclass≠MajorThird
            THEN Val←Val+EndOnPerfectPenalty;
!
```

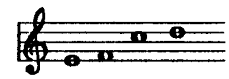
If we end on a major third, this should of course be a major third above the cantus, not below it (similarly for the fifth).

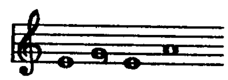
```
;
    ! penalize direct motion any kind (contrary motion is better);
    IF MotionType(Us(Cn-1,v),Cp,Other1,Other0)=DirectMotion
      THEN
        BEGIN
        Val←Val+DirectMotionPenalty;
        IF IntClass=Tritone
          THEN Val←Val+DirectToFifthPenalty;
        END;
    ! penalize compound intervals (close position is favored);
    IF ABS(Interval)>Octave          ! avoid compound intervals wherever possible;
      THEN Val←Val+CompoundPenalty;
!
```

Now we present a raft of rules involving skips. For example, it is considered bad to have too many consecutive skips in the same directions.



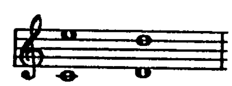
```
;
    ! penalize consecutive skips in the same direction;
    IF Cn>2 AND ConsecutiveSkipsInSameDirection(Us(Cn-2,v),Us(Cn-1,v),Cp)
      THEN
        BEGIN
        INTEGER totalJump;
        Val←Val+TwoSkipsPenalty;
        totalJump←ABS(Cp-Us(Cn-2,v));
    ! do not let these skips traverse more than an octave, nor a seventh;
        IF ((totalJump>MajorSixth) AND (totalJump≠Octave))
          THEN Val←Val+TwoSkipsNotInTriadPenalty;
        END;
    ! penalize a skip to an octave;
    IF ((IntClass=unison) AND (ASkip(MelInt) OR ASkip(Other0-Other1)))
      THEN Val←Val+SkipTo8vePenalty;

    ! do not skip from a unison (not a very important rule);
    IF (Other1=Us(Cn-1,v)) AND ASkip(MelInt)
      THEN Val←Val+SkipFromUnisonPenalty;
```



```
! penalize skips followed or preceded by motion in same direction;
IF Cn>2 AND ASkip(MelInt) AND SameDir
  THEN                              ! skip preceded by same direction;
    BEGIN
! especially penalize fifths, sixths, and octaves of this sort;
    IF ABS(MelInt)<Fifth         ! skip of third or fourth;
      THEN Val←Val+SkipPrecededBySameDirectionPenalty
    ELSE
      IF ABS(MelInt)=Fifth OR ABS(MelInt)=Octave
        THEN Val←Val+FifthPrecededBySameDirectionPenalty
      ELSE Val←Val+SixthPrecededBySameDirectionPenalty;
    END;
IF Cn>2 AND ASkip(LastMelInt) AND SameDir
  THEN
    BEGIN
    IF ABS(LastMelInt)<Fifth    ! skip of third or fourth;
      THEN Val←Val+SkipFollowedBySameDirectionPenalty
    ELSE
      IF ABS(LastMelInt)=Fifth OR ABS(LastMelInt)=Octave
        THEN Val←Val+FifthFollowedBySameDirectionPenalty
      ELSE Val←Val+SixthFollowedBySameDirectionPenalty;
    END;
```

```
! too many skips in a row -- favor a mix of steps and skips;
IF Cn>4 AND
   ASkip(MelInt) AND
   ASkip(LastMelInt) AND
   ASkip(Us(Cn-2,v)-Us(Cn-3,v))
      THEN Val←Val+MelodicBoredomPenalty;
```

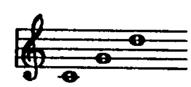
```
! avoid tritones melodically;
 IF Cn>4 AND
     (ABS(Cp-Us(Cn-2,v))=Tritone OR
      ABS(Cp-Us(Cn-3,v))=Tritone OR
      ABS(Cp-Us(Cn-4,v))=Tritone)
       THEN Val←Val+MelodicTritonePenalty;
!
```

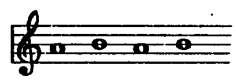
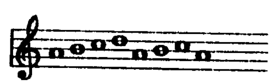
As can be seen from the solutions given at the end of the paper, tritone handling is far from perfect. Perhaps the *MelodicTritone* penalty should be higher, and more elaborate checks should be added for tritones over larger melodic distances.



```
;
   ! do not allow movement from a tenth to an octave by contrary motion;
   IF Species=5 AND NumParts=1
     THEN
       IF ATenth(Other1-Us(Cn-1,v)) AND AnOctave(Interval)
         THEN Val←Val+TenthToOctavePenalty;
!
```

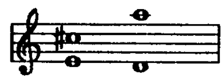
Fux admits this rule (disallowing motion from a tenth to an octave) is without obvious musical justification, but states that the rule merely follows the practice of the great composers.



```
;
   ! more range checks -- did we go over an octave recently;
   IF Cn>2 AND ABS(Cp-Us(Cn-2,v))>Octave THEN Val←Val+OverOctavePenalty;
   ! same for a twelfth;
   IF Cn>3 AND TotalRange(Cn,Cp,v)>(Octave+Fifth) THEN Val←Val+OverTwelfthPenalty;
                                                  IF Val≥infinity THEN RETURN(val);
   ! slightly penalize repeated notes;
   IF (Cn>3) AND (Cp=Us(Cn-2,v)) AND (Us(Cn-1,v)=Us(Cn-3,v))
     THEN Val←Val+TwoRepeatedNotesPenalty;
   IF  (Cn>5) AND (Cp=Us(Cn-3,v)) AND
       (Us(Cn-1,v)=Us(Cn-4,v)) AND (Us(Cn-2,v)=Us(Cn-5,v))
     THEN Val←Val+ThreeRepeatedNotesPenalty;
   IF  (Cn>6) AND (Cp=Us(Cn-4,v)) AND
       (Us(Cn-1,v)=Us(Cn-5,v)) AND (Us(Cn-2,v)=Us(Cn-6,v))
     THEN Val←Val+ThreeRepeatedNotesPenalty-1;
   IF  (Cn>7) AND (Cp=Us(Cn-4,v)) AND
       (Us(Cn-1,v)=Us(Cn-5,v)) AND (Us(Cn-2,v)=Us(Cn-6,v)) AND
       (Us(Cn-3,v)=Us(Cn-7,v))
     THEN Val←Val+FourRepeatedNotesPenalty;
   IF  (Cn>8) AND (Cp=Us(Cn-5,v)) AND
       (Us(Cn-1,v)=Us(Cn-6,v)) AND (Us(Cn-2,v)=Us(Cn-7,v)) AND
       (Us(Cn-3,v)=Us(Cn-8,v))
     THEN Val←Val+FourRepeatedNotesPenalty;
!
```



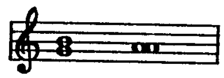
These rules do not penalize sequences, nor do they catch widely separated pattern repetitions. We may add such checks to a later version of the program.

```
;
    IF LastNote(Cn,v)
      THEN
        BEGIN
        INTEGER LastPitch;
        LastPitch←(Us(Cn-1,v) MOD 12);
        IF ((LastPitch=11) OR ((LastPitch=10) AND (Mode=Phrygian))) AND (Pitch≠0)
          THEN Val←Val+UnresolvedLeadingTonePenalty;
        END;
!
```



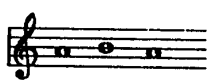
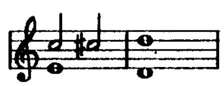
Without this rule forcing the leading tone to go to the tonic, in multi-voice situations the program sometimes skips from the leading tone to some other note than the tonic to avoid direct motion to an octave.

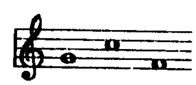
```
;
                                                    IF Val≥Infinity THEN RETURN(val);
    ! an imperfect consonance is better than a perfect consonance;
    IF PerfectConsonance[IntClass] THEN Val←Val+PerfectConsonancePenalty;

    ! no unisons allowed within counterpoint unless more than 2 parts;
    IF NumParts=1 AND (Interval=unison) THEN Val←Val+UnisonPenalty;
                                                    IF Val≥Infinity THEN RETURN(val);

    ! seek variety by avoiding pitch repetitions;
    Val←Val+PitchRepeats(Cn,Cp,v)/2;
!
```

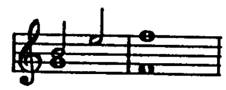


Without a few rules governing melodic practices, the counterpoint writer is as happy to bounce around by octaves as by steps. This can lead to some rather funny looking melodies.

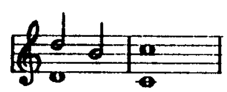
```
;
    ! penalize octave leaps a little;
    IF AnOctave(MelInt) THEN Val←Val+OctaveLeapPenalty;

    ! similarly for minor sixth leaps;
    IF MelInt=MinorSixth THEN Val←Val+SixthLeapPenalty;
    ! penalize upper neighbor notes slightly (also lower neighbors);
    IF  Cn>2 AND
        (MelInt<0) AND                  ! last interval was downward;
        AStep(MelInt) AND               ! downward step in fact;
        (Cp=Us(Cn-2,v))                 ! upper neighbor (this note repeated 2 back);
      THEN Val←Val+UpperNeighborPenalty;
    IF  Cn>2 AND
        (MelInt>0) AND                  ! last interval was upward;
        AStep(MelInt) AND               ! upward step in fact;
        (Cp=Us(Cn-2,v))                 ! lower neighbor (this note repeated 2 back);
      THEN Val←Val+LowerNeighborPenalty;
    ! do not allow normal leading tone to precede raised leading tone;
    ! also check here for augmented fifths and diminished fourths;
    IF (NOT InMode(Pitch,Mode)) AND     ! must be raised leading tone;
       ((MelInt=minorsecond) OR         ! and preceding was a normal leading tone;
        (MelInt=MinorSixth) OR          ! preceded by augmented fifth;
        (MelInt=-MajorThird))           ! preceded by diminshed fourth;
      THEN
        Val←Val+OutOfModePenalty;
    ! slightly frown upon leap back in the opposite direction;
    IF Cn>2 AND ASkip(MelInt) AND Askip(LastMelInt) AND NOT SameDir
      THEN
        BEGIN
        Val←Val+(0 MAX ((ABS(MelInt)+ABS(LastMelInt))-8));
        IF Cn>3 AND ASkip(Us(Cn-2,v)-Us(Cn-3,v))
          THEN Val←Val+ThreeSkipsPenalty;
        END;
!
```





This rule only penalizes a leap of more than a major third followed immediately by a leap in the opposite direction. Fux has an example that leaps up an octave, then immediately down an octave, so this is not a terrible flaw. More "common sense" checks of this sort are given below.

```
;
    ! try to approach cadential passages by step;
    IF NumParts=1 AND Cn≥TotalNotes[v]-4 AND (ABS(MelInt)>4)
      THEN Val←Val+LeapAtCadencePenalty;

    ! check for entangled voices;
    Cross←0;
    IF NumParts=1
      THEN
        FOR k←4 STEP 1 UNTIL Cn DO
          IF (Us(k,v)-Cantus(k,v))*(Us(k-1,v)-Cantus(k-1,v))<0
            THEN Cross←Cross+1;
    IF Cross>0 THEN Val←Val+(0 MAX ((Cross-2)*3));
!
```

This formula "(Cross-2)*3" does not penalize the melody until there are three crossings.

```
;
    ! don't repeat note on upbeat;
    IF (Upbeat(Cn,v) AND (MelInt=Unison))
      THEN Val←Val+RepetitionOnUpbeatPenalty;
    ! avoid tritones near Lydian cadence;
    IF (Mode=Lydian) AND (Cn>TotalNotes[v]-4) AND (Pitch=6)
      THEN Val←Val+LydianCadentialTritonePenalty;
    ! various miscellaneous checks.  More elaborate dissonance resolution and
    ! cadential formula checks will be given under "Species definition";
    IF Species≠1 AND DownBeat(Cn,v)
      THEN
        BEGIN
        IF Species<4
          THEN
            BEGIN
            IF MelInt=unison AND        ! don't repeat upbeat note on downbeat;
               NOT LastNote(Cn,v)
              THEN Val←Val+UnisonDownbeatPenalty;
            ! check for dissonance that doesn't fill a third as a passing tone;
            IF Dissonance[LastIntClass] AND
               ((NOT AStep(MelInt)) OR (NOT SameDir))
              THEN Val←Val+DissonanceNotFillingThirdPenalty;

            END;
        ! check for Direct 8ve or 5 where the intervening interval is less than a fourth;
        IF DirectMotionToPerfectConsonance(Us(Cn-2,v),Cp,Other2,Other0) AND
          (ABS(LastMelInt)<Fourth)
          THEN Val←Val+DirectPerfectOnDownbeatPenalty;

        END;
    ! check for tritone with cantus or bass;
    IF IntClass=Tritone THEN Val←Val+VerticalTritonePenalty;

    ! check for melodic interval variety;
    IF Cn>10 AND TooMuchOfInterval(Cn,Cp,v)
      THEN Val←Val+MelodicBoredomPenalty;

    RETURN(Val);
    END;
!
```

We have not yet provided rules guiding the program toward good overall melodic shapes. It may be useful in this regard to treat a melody as a waveform and look for its low frequency components. We then presumably want the various melodies to have different such components or at least different initial phases.

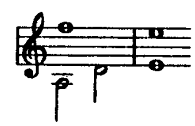
Species Definition

There are differences between the species that can most easily be handled by providing a special set of rules for each species. This procedure is called in *Check* (given above). The two major special areas are cadential formulas and dissonance handling.

```
;
    INTEGER PROCEDURE SpecialSpeciesCheck(
        INTEGER Cn,Cp,v,Other0,Other1,Other2,NumParts,
                Species,MelInt,Interval,ActInt,LastIntClass,Pitch,LastMelInt);
    BEGIN "SSC"
    INTEGER Val;
    IF Species=1 THEN RETURN(0);                ! no special rules for 1st species;
    Val←0;                                      ! accumulated penalty;
!
```

In first species all vertical intervals must be consonances (this is handled by the procedure *ADissonance* given above), the first and last intervals must be perfect consonances, imperfect consonance are better than perfect, and the next to last interval must be a major sixth if the cantus firmus is below or a minor third if the cantus firmus is above. All these rules are handled by *Check*, so we have no further rules to apply in first species.

In second species the interval at the downbeat must be consonant, but the upbeat can be dissonant if it fills in a third (a "passing tone"). The next to last measure has a fifth to a major sixth if the counterpoint is above, and a fifth to a minor third if it is below. The phrygian cadence is also special. Direct motion to a perfect consonance between successive down beats is accepted if the intervening interval is larger than a major third. *ADissonance* handles the passing tone check, so we need only define the cadences here.

```
;
    IF Species=2
      THEN
        BEGIN "2"
        IF NextToLastNote(Cn,v) AND (Pitch=11 OR Pitch=10)
          THEN                              ! we have the leading tone;
            IF ((Mode=Phrygian) OR Interval≥0)
              THEN                          ! phrygian below is special case;
                BEGIN
                IF LastIntClass=Fifth
                  THEN Val←Val+BadCadencePenalty;
                END
            ELSE
              IF LastIntClass=MinorSixth
                THEN Val←Val+BadCadencePenalty;
        END "2"
    ELSE
!
```

Fourth species introduces suspensions and accented dissonances. The dissonance must be resolved by a step downward. Since there are more similarities between third and fifth species and second and fourth, we can save some code by combining third and fifth below.

```
;
      BEGIN "3 4 5"                         ! 3rd, 4th, and 5th need more info;
      INTEGER k;
      IF Species=4
        THEN
          BEGIN "4"
          IF DownBeat(Cn,v) AND MelInt=Unison     ! strongly encourage ligatures;
            THEN Val←Val+NotaLigaturePenalty;
!
```

Fux says we should try to use a ligature wherever possible, so we penalize anything else.

```
;
          IF Upbeat(Cn,v) AND
             Dissonance[LastIntClass]
            THEN
              BEGIN
              IF (MelInt≠-minorSecond AND MelInt≠-majorSecond)
                THEN Val←Val+UnresolvedLigaturePenalty;
!
```

If the last interval was a downbeat and a dissonance, the dissonance must be resolved on this beat by a step downward.

```
;
              IF ActInt=Unison AND
                 (Interval<0 OR
                 (ABS(Us(Cn-2,v)-Other2) MOD 12)=Unison)
                THEN Val←Val+NoTimeForaLigaturePenalty;
!
```

The vertical interval cannot resolve to a unison or octave if the preceding downbeat was also a unison or octave (the ligature does not make direct motion to a unison or octave acceptable, but does enable one to move from one fifth to another). It is also bad to resolve to an octave if the cantus firmus is above the melody.

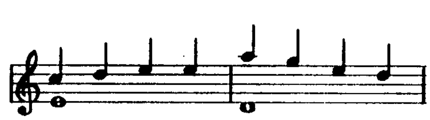
```
;
              IF ActInt=Fifth OR actInt=Tritone
                THEN Val←Val+NoTimeForALigaturePenalty;

              END;
!
```

And, finally, the resolution must be to a consonance (not a tritone for example).

```
;
          END "4"
     ELSE
!
```

In third species in addition to the passing tone dissonance (which can occur on any beat except the first), we must also accept cambiatas. The result is that any dissonance must be approached by step. If the dissonance occurs on the second beat and is approached from above and is left by a third down, then it must be left by two steps up. In the latter case every interval must be consonant except the second (and the fourth if it is a passing tone). In addition, this particular dissonance (the cambiata) can only be a seventh if the cantus firmus is below, or a fourth if it is above (in two parts).

```
;
        BEGIN "3 5"
        INTEGER Above,Cross;
        Above←(Interval≥0);

        ! added check to stop optimizer from changing 4th beat passing tones into
        ! repeated notes+skip;
        IF (Beat8(Onset[Cn,v])=6 OR Beat8(Onset[Cn,v])=7) AND Cp=Us(Cn-1,v)
          THEN Val←Val+UnisonOnBeat4Penalty;
```

```
! skip to down beat seems not so great;
IF Beat8(Onset[Cn,v])=0
  THEN
    BEGIN
    IF ASkip(MelInt) THEN Val←Val+SkipToDownbeatPenalty;
    IF Cn>2 AND (ActInt=unison OR ActInt=Fifth)
      THEN                          ! look for parallel 8ve or 5 on downbeat;
        BEGIN
        INTEGER i;
        IF Species=5
          THEN
            FOR i←Cn-1 STEP -1 UNTIL 1 DO
              BEGIN
              IF Beat8(OnSet[i,v])=0 THEN DONE;
              END
        ELSE i←Cn-4;
        IF (ABS(Us(i,v)-Bass(i,v)) MOD 12)=ActInt
          THEN Val←Val+DownbeatUnisonPenalty;
        END;
    END;
! check for cambiata not resolved correctly (on 4th beat);
IF Beat8(Onset[Cn,v])=6 AND
   AThird(ABS(LastMelInt)) AND
   Dissonance[ABS(Us(Cn-2,v)-Other2) MOD 12] AND
   ( (MelInt<0) OR
     (ABS(MelInt)≠MajorSecond AND ABS(MelInt)≠MinorSecond))
  THEN Val←Val+NotaCambiataPenalty;
                                             IF Val≥infinity THEN RETURN(Val);
IF Species=3 AND
   Cn>1 AND
   Dissonance[LastIntClass]
  THEN                                       ! cambiata or passing tones?;
    BEGIN
    CASE Beat8(Onset[Cn,v]) OF
      BEGIN
      [0][6]
          IF (NOT AStep(MelInt)) OR          ! (4 was diss or 3 was diss);
             (NOT AStep(LastMelInt)) OR
             ((MelInt⊗LastMelInt)<0)
            THEN Val←Val+DissonancePenalty;
      [2] Val←Val+DissonancePenalty;         ! can't happen (1 can't be diss);
      [4] IF (NOT AStep(LastMelInt)) OR      ! cambiata or passing tone are alike in this;
             (ABS(MelInt)>MajorThird) OR     ! step if passing, third if cambiata;
             (MelInt=0) OR                   ! neither being a unison;
             ((LastMelInt⊗MelInt)<0)         ! both continue same direction;
            THEN Val←Val+DissonancePenalty   ! so lose;
          ELSE                               ! now check more cases;
            IF (NOT AStep(MelInt))           ! passing tone ok (so ELSE clause ok);
              THEN
                BEGIN
                IF Above
                  THEN
                    BEGIN
                    IF NOT ASeventh(LastIntClass)
                      THEN Val←Val+DissonancePenalty;
                    END
                ELSE
                  IF LastIntClass≠Fourth
                    THEN Val←Val+DissonancePenalty;
                END
      END;
    END;
```

```
IF Species=5
  THEN
    BEGIN "5"
    INTEGER LastDisInt;
    IF Cn>1 AND Beat8(Onset[Cn,v])=0 AND (Cp≠Us(Cn-1,v)) AND (Dur[Cn,v]≤Dur[Cn-1,v])
      THEN Val←Val+(NotaLigaturePenalty/3);
    IF Cn>3 AND
       Dur[Cn,v]=halfnote AND
       Beat8(Onset[Cn,v])=4 AND
       Dur[Cn-1,v]=quarternote AND
       Dur[Cn-2,v]=quarternote
      THEN Val←Val+HalfUntiedPenalty;
    IF Dur[Cn,v]=EighthNote AND DownBeat(Cn,v) AND Dissonance[ActInt]
      THEN Val←Val+DissonancePenalty;
                                               IF Val≥infinity THEN RETURN(Val);
    IF Cn>1 THEN LastDisInt←ABS(Us(Cn-1,v)-Other1) MOD 12;
    IF Cn>1 AND
       Dissonance[LastDisInt]
      THEN                                     ! cambiata or passing tones?;
        BEGIN "Diss"
        CASE Beat8(Onset[Cn-1,v]) OF
          BEGIN
          [6][4]
               IF LastDisInt=Fourth AND        ! dissonance was a fourth;
                  MelInt=Unison AND            ! tied to current;
                  (Other0-Other1)=unison AND   ! and bass is also motionless;
                  Beat8(Onset[Cn,v])=0         ! and tied across bar;
                 THEN                          ! "consonant fourth"!;
               ELSE                            ! cambiata or passing tone or suspension;
               IF (NOT AStep(MelInt)) OR       ! (4 was diss or 3 was diss);
                  (NOT AStep(LastMelInt)) OR
                  ((MelInt×LastMelInt)<0) OR
                  (Dur[Cn-1,v]=eighthnote) OR
                  (Dur[Cn-1,v]=quarterNote AND Dur[Cn-2,v]=halfNote)
                 THEN Val←Val+DissonancePenalty;
          [1][3][5][7]
               IF (NOT AStep(MelInt)) OR       ! (4 was diss or 3 was diss);
                  (NOT AStep(LastMelInt)) OR
                  ((MelInt×LastMelInt)<0)
                 THEN Val←Val+DissonancePenalty;
          [0]
            BEGIN
            IF (Dur[Cn-2,v]=eighthnote) OR
               (Dur[Cn-2,v]<Dur[Cn-1,v])
              THEN Val←Val+NoTimeForALigaturePenalty;
            IF (MelInt≠-minorSecond AND MelInt≠-majorSecond)
              THEN Val←Val+UnresolvedLigaturePenalty;
            IF ActInt=Fourth OR ActInt=Tritone
              THEN Val←Val+NoTimeForALigaturePenalty;
            IF ActInt=Fifth AND Interval<0
              THEN Val←Val+NoTimeForALigaturePenalty;
            IF ActInt=0 AND
               (ABS(Us(Cn-2,v)-Other2) MOD 12)=0
              THEN Val←Val+NoTimeForaLigaturePenalty;
            IF LastMelInt≠unison THEN Val←Val+DissonancePenalty;
            END;
```

```
        [2] IF (NOT AStep(LastMelInt)) OR      ! cambiata or passing tone are alike in this;
              (ABS(MelInt)>MajorThird) OR      ! step if passing, third if cambiata;
              (MelInt=0) OR                    ! neither being a unison;
              (Dur[Cn-1,v]=eighthnote) OR
              ((LastMelInt*MelInt)<0)          ! both continue same direction;
            THEN Val←Val+DissonancePenalty     ! so lose;
          ELSE                                 ! now check more cases;
            IF (NOT AStep(MelInt))             ! passing tone ok (so ELSE clause ok);
              THEN
                BEGIN
                IF Above
                  THEN
                    BEGIN
                    IF NOT ASeventh(LastIntClass)
                      THEN Val←Val+DissonancePenalty;
                    END
                ELSE
                  IF LastIntClass≠Fourth
                    THEN Val←Val+DissonancePenalty;
                END
        END;
      END "Diss";
    IF Cn>1 AND Dur[Cn-1,v]=eighthnote AND (NOT AStep(MelInt))
      THEN Val←Val+EighthJumpPenalty;

    IF Cn>1 AND Dur[Cn-1,v]=halfnote AND Beat8(Onset[Cn,v])=4 AND MelInt=Unison
      THEN Val←Val+UnisonUpbeatPenalty;
    END "5";
  END "3 5";
 END "3 4 5";
RETURN(Val);
END "SSC";
!
```

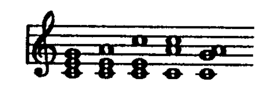
It is perhaps interesting that we do not need to add any special encouragement for cambiatas – they occur as a side effect of the rules (once the dissonance handling is defined as acceptable). The passing tone (second note of the cambiata) is more likely to jump to an imperfect consonance (the leap down by a third making a sixth with the cantus) than to a perfect consonance (the latter is slightly penalized), so we get cambiatas simply as a result of the preference for imperfect consonances.

Multi-Part Counterpoint

Each added voice must obey all the normal rules with regard to the bass voice, but need not be quite so particular in relation to other voices. We assume here that the bass voice is either the cantus firmus or the first of the voices calculated by the Search mechanism. The extra rules for multi-voice counterpoint deal mainly with the makeup of chords.

```
;
    INTEGER ARRAY IntervalsWithBass[0:7];
        ! 0 = octave, 2 = step, 3 = third, 4 = fourth, 5 = fifth, 6 = sixth, 7 = seventh;
!
```

We need to look at the chord currently formed by the counterpoint and decide whether the pitch doublings are acceptable. In general, the leading tone should not be doubled, the fifth is rarely doubled, the third and sixth (figuring from the bass) can be doubled, but rarely tripled, and the octave can occur as often as necessary. A full chord (octave, third, and fifth for example), is better than a partial one (octave, octave, and third for example). In any case, the chord must contain at least one imperfect consonance. A root position triad is better than a first inversion triad (octave, third, and fifth is better than octave, third and sixth). Dissonances between upper voices are acceptable as long as the voices are individually consonant with the bass, but the 6-5 chord is a special case. In this chord the upper voices contain both the fifth and sixth above the bass, and according to some sources, the fifth should be treated as a form of suspension. To simplify matters we allow the 6-5 chord only in fifth species.

```
;
    PROCEDURE AddInterval(INTEGER n);
    BEGIN                               ! collect current notes in chord;
    INTEGER ActInt;
    ActInt←(CASE (n MOD 12) OF (0,2,2,3,3,4,4,5,6,6,7,7));
    IntervalsWithBass[ActInt]←IntervalsWithBass[ActInt]+1;
    END;
!
```

After each voice has checked that its proposed pitch agrees with the basic rules, it must also check for agreement with the other voices currently active. The following procedure handles this:

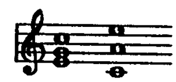
```
;
    INTEGER PROCEDURE OtherVoiceCheck(INTEGER Cn,Cp,v,NumParts,Species);
    BEGIN
    INTEGER Val,k,CurBass,Other0,Other1,Int0,Int1,ActPitch,IntBass,LastCp;
    BOOLEAN AllSkip;
    IF v=1 THEN RETURN(0);              ! two part or bass voice, so nothing to check;
    ARRCLR(IntervalsWithBass);          ! our current chord;
    Val←0;
    CurBass←Bass(Cn,v);
!
```

Since we assume that either the first voice or the cantus firmus is the bass, we cannot allow other voices to cross below the current bass and thereby render invalid all the previous calculations of voice leading and chord type.

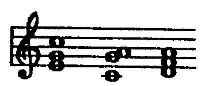
```
;
    IF Cp≤CurBass
      THEN Val←Val+CrossBelowBassPenalty;
!
```

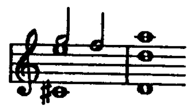
We must also ensure that the raised leading tone in the bass (if present) does not confuse the consonance checker into thinking a diminished fourth is a major third.

```
;
    IntBass←((Cp-CurBass) MOD 12);
    IF IntBass=MajorThird AND NOT InMode(CurBass,Mode)
      THEN Val←Val+AugmentedIntervalPenalty;
    ActPitch←(Cp MOD 12);
    IF Val≥infinity OR (v=NumParts AND Dissonance[IntBass]) THEN RETURN(Val);
    ! logic here is that only the last part can be non-1st species
      and may therefore have various dissonances that don't want to be
      calculated as chord tones
    ;
    AllSkip←ASkip(Cp-Us(Cn-1,v));
    AddInterval(IntBass);
    LastCp←Us(Cn-1,v);
    FOR k←0 STEP 1 UNTIL v-1 DO
      BEGIN                             ! check our pitch against each other voice;
      Other0←Other(Cn,v,k);
      Other1←Other(Cn-1,v,k);
      IF NOT ASkip(Other0-Other1) THEN AllSkip←FALSE;
      AddInterval(Other0-CurBass);      ! add up tones in chord;
```

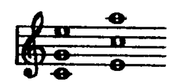


```
! avoid unison with other voice;
IF NOT LastNote(Cn,v) AND Other0=Cp THEN Val←Val+UnisonPenalty;

! keep upper voices closer together than lower;
IF Other0≠CurBass AND ABS(Cp-Other0)≥Octave+Fifth
  THEN Val←Val+UpperVoicesTooFarApartPenalty;
! check for direct motion to perfect consonace between these two voices;
Int0←ABS(Other0-Cp) MOD 12;
Int1←ABS(Other1-LastCp) MOD 12;
IF Int1=Int0
  THEN
    IF Int0=unison THEN Val←Val+ParallelUnisonPenalty ELSE
    IF Int0=fifth THEN Val←Val+ParallelFifthPenalty;

IF Cn>2 AND Int0=Unison AND (ABS(Us(Cn-2,v)-Other(Cn-2,v,k)) MOD 12)=Unison
  THEN Val←Val+ParallelUnisonPenalty;

IF Val≥infinity THEN RETURN(Val);

! penalize tritones between voices;
IF Int0=Tritone THEN Val←Val+VerticalTritonePenalty;
!
```

The following block of code attempts to ensure that a 6-5 chord is properly prepared and resolved. If there is a dissonance between these two voices, it is either a fourth or tritone (6-3 or 5-3 chord), a second (6-5 with 6th above), or a seventh (6-5 with fifth above). The dissonance in the 6-3 chord is considered unobjectionable. In the implementation here, the 6-5 chord is prepared by holding the fifth while the sixth is added (a sort of suspension below), then "resolve" the fifth by moving downward by step.

```
;
IF Species=5
  THEN
    BEGIN
    IF Dissonance[Int1] AND Int1≠fourth
      THEN                           ! must be a 6-5 chord;
        BEGIN
        INTEGER ourLastInt;
        ourLastInt←((LastCp-Bass(Cn-1,v)) MOD 12);
        IF ourLastInt≠unison      ! if unison, 6-6 somewhere else?;
          THEN
            IF (ourLastInt=fifth)
              THEN
                IF (ASkip(Cp-LastCp) OR Cp≥LastCp)
                  THEN Val←Val+UnresolvedSixFivePenalty
                ELSE
            ELSE
              IF (ASkip(Other0-Other1) OR Other0≥Other1)
                THEN Val←Val+UnresolvedSixFivePenalty;
        END;
    IF Dissonance[Int0] AND Int0≠fourth AND IntBass≠unison
      THEN
        IF (IntBass=fifth AND (Cp-LastCp)≠unison) OR
           (IntBass≠fifth AND (Other0-Other1)≠unison)
          THEN Val←Val+UnpreparedSixFivePenalty;
    END;
! penalize direct motion to perfect consonance except at the cadence;
IF NOT LastNote(Cn,v) AND
   DirectMotionToPerfectConsonance(LastCp,Cp,Other1,Other0)
  THEN Val←Val+InnerVoicesInDirectToPerfectPenalty;

! if we have an unraised leading tone it is possible that some other;
! voice has the raised form thereof (since the voices can move at very
! different paces, one voice's next to last note may be long before
! another's);
IF ActPitch=10 AND          ! if 11 we've aready checked;
   (Other0 MOD 12)=11       ! They have the raised form;
  THEN Val←Val+DoubledLeadingTonePenalty;

! similarly for moition to a tritone;
IF MotionType(LastCp,Cp,Other1,Other0)=DirectMotion AND Int0=Tritone
  THEN Val←Val+InnerVoicesInDirectToTritonePenalty;
```

```
  ! look for a common diminished fourth (when a raised leading tone is in
  ! the bass, a "major third" above it is actually a diminished fourth.
  ! Similarly, an augmented fifth can be formed in other cases;
  IF ActPitch=3 AND (Other0 MOD 12)=11
    THEN Val←Val+AugmentedIntervalPenalty;

  ! try to encourage voices not to move in parallel too much;
  IF MotionType(LastCp,Cp,Other1,Other0)≠ContraryMotion
    THEN Val←Val+NotContraryToOthersPenalty;
  END;
!
```

Now we must check the current contents of the chord being formed:

```
;
  ! check for doubled third;
  IF intervalsWithBass[3]>1 THEN Val←Val+ThirdDoubledPenalty;

  ! check for doubled sixth;
  IF IntervalsWithBass[3]=0 AND IntervalsWithBass[6]>1 THEN Val←Val+DoubledSixthPenalty;
  ! check for too many voices at octaves;
  IF IntervalsWithBass[0]>2 THEN Val←Val+TripledBassPenalty;

  ! check for doubled fifth;
  IF IntervalsWithBass[5]>1 THEN Val←Val+DoubledFifthPenalty;
  ! check that chord contains at least one third or sixth;
  IF v=NumParts AND (NOT LastNote(Cn,v)) AND IntervalsWithBass[3]=0 AND IntervalsWithBass[6]=0
    THEN Val←Val+NotTriadPenalty;

  ! discourage all voices from skipping at once;
  IF v=NumParts AND AllSkip
    THEN Val←Val+AllVoicesSkipPenalty;
  ! except in 5th species, disallow 6-5 chords altogether;
  IF IntervalsWithBass[5]>0 AND IntervalsWithBass[6]>0 AND Species≠5
    THEN Val←Val+SixFiveChordPenalty;
  RETURN(Val);
  END;
```



Searching Methods

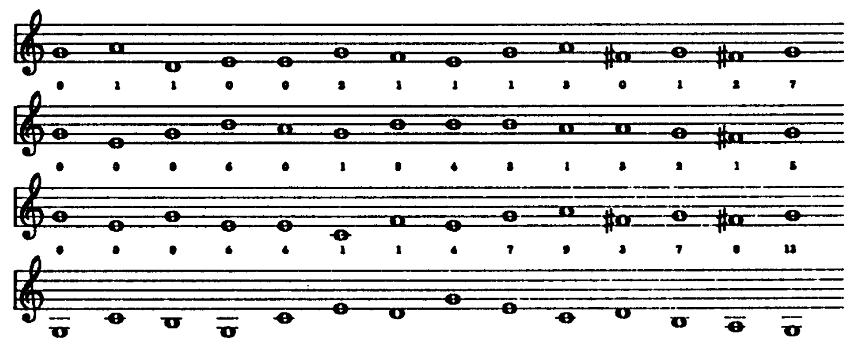
There are many ways to search for acceptable solutions to a counterpoint problem. The main constraint is compute time. If we make an exhaustive search of every possible branch of a short (10 note) two voice first species problem, we have 16↑10 possible solutions. Even if we could check each in a nanosecond, an exhaustive search in this extremely simple case would take 1000 cpu seconds (about 20 minutes) on the F4. Because we hope to handle problems far more complex than this simple one and hope to do it reasonably fast, we must find a smarter search method.

The underlying method is a recursive search. It starts from its current pitch and tries in succession all possible melodic intervals from that pitch (if necessary), looking for any such interval whose associated cumulative penalty is less than the current best overall penalty. The cumulative penalty is the sum of all the penalties associated with each of the notes in the counterpoint melody. The overall best penalty is the lowest cumulative penalty of any complete counterpoint solution found to that point in the counterpoint. At the beginning of time this overall best penalty is infinity (no solutions have been found), but once any solution at all is found, the overall penalty is reset to that new number. If a position is reached that makes further progress impossible, the searcher backs up one note and tries some other interval. The intervals are chosen in an order that maximizes the chance of finding a good interval quickly. If any solution at all exists, we are guaranteed to find it. Given enough time, we are also guaranteed to find the "best" solution according to the rules. By checking only those branches whose penalty is less than the current minimum, we can drastically reduce the number of branches that must be checked, but total execution times can still be high. For a two part counterpoint with a short cantus firmus it is not unreasonable to carry out such a search, but more complex cases drag to a halt. Since our initial goal was to write five to eight part mixed species counterpoint, we obviously need more intelligence guiding the search.

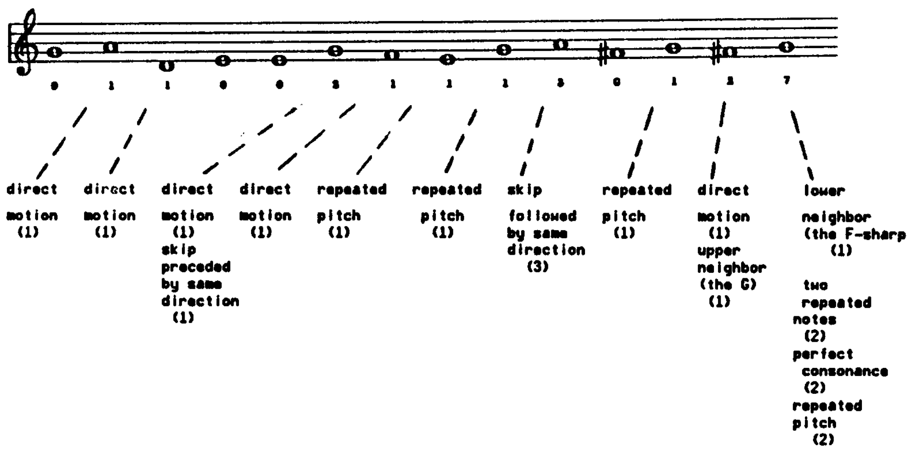
The next method tried was inspired by an article in Science about computer circuit design by simulated annealing. Like the other methods discussed below, this method gains much of its efficiency by accepting less than optimal solutions -- it is not guaranteed to find the best solution, and may in some cases not find any solution at all. In this algorithm, our initial counterpoint is just any random collection of notes. Our annealing "temperature" is the number of semitones each of these notes can move. Time is represented by successive passes over the counterpoint applying the same rules as were applied in the recursive search case, but here we look for the local minimum penalty (whereas in the recursive case we grabbed the first acceptable branch and started down it). Each note independently moves to its local minimum and the next pass is started. This method is extremely fast, and works well in first species counterpoint. It does not always converge on a very good solution, but we originally thought that we could run it across several random collections, and thereby increase our chance of getting something reasonable. In practice, however, these successive runs do not improve much. But more important, beyond first species the annealing process sometimes cannot find any acceptable solution. This problem is most easily observed in second species where a note can be dissonant if it is a passing tone. As the rules are structured, we do not penalize a step to a dissonance because one more step (if possible) will resolve the dissonance correctly. Nor do we ever look ahead to see if such a resolution is possible. If there is no possible resolution after all (if a passing tone is impossible at that point), the annealer has no good way to back out. After many fruitless attempts to get around this problem we finally jettisoned the entire notion.

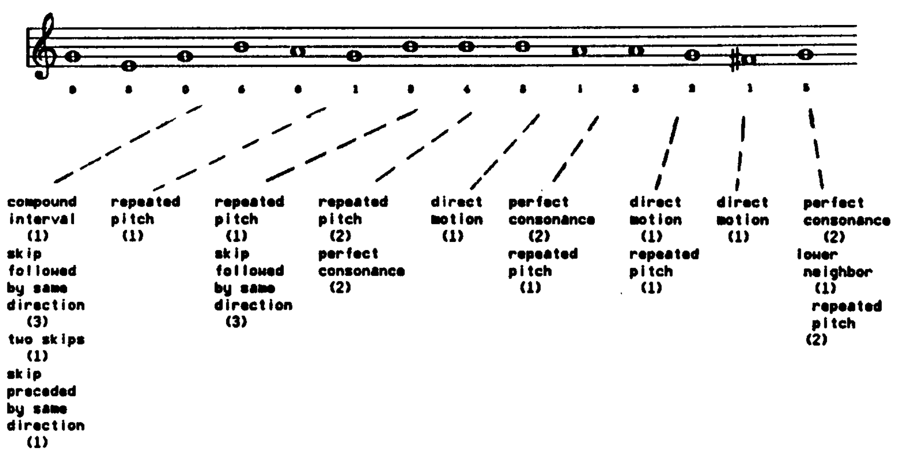
Bernard Mont-Reynaud suggested changing the search to be a best-first search. In this version we compute the penalty associated with each possible melodic interval from the current pitch, then continue recursively using the best of these results first. If forced to abandon a branch, we back up and try the next best interval until a complete solution is found. The first such solution may not be a very good solution, however, because a melody can be lead down a primrose path into a quagmire (by accepting the smallest local penalty we risk falling into a bad overall pathway). If the program is told to search every branch (as in the earlier method), we once again get bogged down in long computations. So a new twist is added. Once we have a solution we drop back to the very beginning and try a different beginning interval. This rather odd looking practice grew from experience watching many hundreds of runs -- generally most of the wasted effort (branches checked that led nowhere) seemed to be attributable to the fact that these new branches made no real difference in the global appearance of the given melody -- much time was being spent optimizing something that had already given all it had to give. By trying a new starting interval, we maximize the chance of finding a truly different solution. Once again we abandon any branch if its accumulated penalty is above that of the best complete solution found so far. Although first through fourth species are quickly solved, fifth species is still a problem. The number of choices increases in this species not only because it generally has more notes in the counterpoint melody, but also because we have a number of possible rhythmic values to assign each note. Several optimizations were added to reduce this problem: we search only for those solutions that are markedly better than the current one (as determined by the variable *PenaltyRatio*), and once a solution has been found we don't spend too much time on any other single interval (controlled by the variables *MaxBranch* and *Branches*). Compute times for multi-voice fifth species can be high even with all this machinery.

To illustrate how the rules help the program decide whether one melody is better than another, take the following three acceptable counterpoints to the bottom line (the cantus firmus). The number under each note is the penalty associated with that note. Despite the fact that the first (top) solution starts out with a higher penalty, it ends up with a lower overall penalty. This case illustrates that a simple best first search is not entirely adequate in all cases.



We can point out in detail how each note in these lines gets its penalty:





In the first attempt at multi-part counterpoint we solved one voice at a time and built up the entire ensemble by layering. This worked well for three voices because the first voice added was always pretty good, and the second added voice still had enough degrees of freedom to find an acceptable solution. As more voices were added however, the later layers became less and less acceptable. It became clear that the entire ensemble has to be calculated together, that all the voices must be examined to decide the current overall best configuration. The search routines that follow implement this form of search.

First we need a place to save the last complete solution while we search for a better one:

```
;
    INTEGER ARRAY BestFit[1:MostNotes,1:MostVoices];
!
```

and variables to hold the current maximum penalty (*MaxPenalty*), the current actual best (lowest) penalty (*BestFitPenalty*), the ratio that determines how much better a new solution has to be to be worth pursuing (*PenaltyRatio*), a flag to tell the searcher when to quit (*AllDone*), the current branch counter (*Branches*), and the maximum number of branches we are willing to search before giving up (*MaxBranch*).

```
;
    INTEGER BestFitPenalty,MaxPenalty,Branches,MaxBranch;
    BOOLEAN AllDone;
    REAL PenaltyRatio;
!
```

We will examine only the best continuations from any given point, making no effort to save every possible continuation. The macro *NumFields* determines how many continuations we save at each branch.

```
;
    DEFINE NumFields=16;
!
```

Each continuation consists of its associated penalty and the melodic intervals of each voice that make up the next note of the continuation. At each branch we search for the *NumFields* best continuations and save them in an array. The following code implements this mechanism.

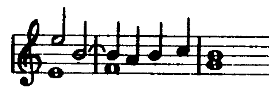
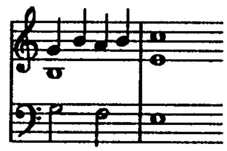
```
;
    DEFINE Field="(MostVoices+1)";
    DEFINE EndF="Field*NumFields";

    PROCEDURE ClearSpace(INTEGER ARRAY Sp);
    ARRCLR(Sp,infinity);
    INTEGER PROCEDURE SaveIndx(INTEGER indx; INTEGER ARRAY sp);
    BEGIN                          ! if INDX is less than current NUMFIELD-th worst,
                                   ! find its position in SP, insert space for its
                                   ! data, and return a pointer to the block.  The
                                   ! blocks are stored "backwards" for ARRBLT;
    INTEGER i;
    FOR i←EndF STEP -Field UNTIL 0 DO
      IF Sp[i]>indx THEN DONE;
    IF i>0                         ! 0 is the end of the list.  If i>0 then we insert INDX;
      THEN
        BEGIN
        ARRBLT(Sp[0],Sp[Field],i);
        Sp[i]←Indx;
        ! SP[i]=penalty for block starting at I.  SP[i-1]=index into
        ! melodic interval array for voice 1, SP[i-2] for voice 2 and
        ! so on.  The searcher starts at SP[EndF] and works backwards
        ! through the stored continuations as it searches for a satisfactory
        ! overall solution;
        END;
    RETURN(i);
    END;
!
```

The next procedure takes a newly completed solution and saves it. At this time we also check for problems associated with raised leading tones. If the musica ficta rules were at all clear, we could also add them to the solution. The raised leading tone code can run into problems – a future version may try to be smarter about cadential passages.

```
;
   PROCEDURE SaveResults(INTEGER CurrentPenalty,Penalty,v1,Species);
   BEGIN
   INTEGER i,v,LastPitch,v,Cn,k;
   FOR v←1 STEP 1 UNTIL v1 DO                     ! check all voices for raised leading tone;
     BEGIN
     Cn←TotalNotes[v];
     LastPitch←(Us(Cn-1,v) MOD 12);               ! must be raised if any are;
     IF (NOT InMode(LastPitch,Mode))              ! it is a raised leading tone;
       THEN
         FOR k←2 STEP 1 WHILE TRUE DO             ! look backwards through voice's notes;
           BEGIN "outer"
           INTEGER Pitch;
           IF k≥(Cn-1) THEN DONE;                 ! ran off start!!;
           Pitch←(Us(Cn-k,v) MOD 12);             ! current pitch;
           IF (Pitch<8 AND Pitch≠0) OR            ! not 6-7-1 scale degree anymore;
              (ASkip(Us(Cn-k+1,v)-Us(Cn-k,v)))    ! skip breaks drive to cadence;
             THEN DONE;                           ! so exit;
           Pitch←ABS(Us(Cn-k,v)-Us(Cn-k-1,v));    ! interval with raised leading tone;
           IF pitch=Fourth OR Pitch=Fifth OR Pitch=unison OR Pitch=Octave THEN DONE;
                                                  ! don't create illegal melody;
           FOR i←0 STEP 1 UNTIL v1 DO             ! do others have unraised form?;
             IF i≠v AND (Other(Cn-k,v,i) MOD 12)=11 THEN DONE "outer";
           IF ((Us(Cn-1,v)-Us(Cn-k,v))=MinorThird) OR
              ((Us(Cn-1,v)-Us(Cn-k,v))=MinorSecond)
             THEN                                 ! raise it and maybe 6th degree too;
               SetUs(Cn-k,Us(Cn-k,v)+1,v);
           END "outer";
     END;
   END;
!
```

Musica ficta are not entirely trivial to add to the program. If we were doing so, this would be the place to do it. The ficta-finder would run through all the voices here looking for tritones. If one is found a decision has to be made whether to alter (flatten or sharpen) the current note, or leave it alone. In a simple version tried during the development of this program, we used the following sequence:

```
Step through all voices
  Step through all notes of each voice
    If current pitch is B (11) and it is approached and left by either
        a unison, a rising major second, a rising major second, a
        descending minor second, or a descending minor third (to
        avoid creating illegal augmented or diminished intervals
        melodically), and it forms a tritone with some other voice,
      Then
        If melodic interval is not a unison
          Then flatten the B
        Else
          While pitch is B, look to see if the flattened form will
          create tritones (E in other voices), and check to see that
          the interval on either side of the repeated B's is the
          correct form of interval. If all this is true, flatten all
          the B's.
```

This algorithm is moderately conservative (it makes no attempt to sharpen F's for example), but still makes a mess of certain passages. For example, take the simple passage:

Here our algorithm flattens the second B but not the first (correctly avoiding a tritone melodically), but perhaps the correct ficta is an F-sharp. The algorithm also ignores the possible presence of the unaltered pitch in other voices as in:

In any case, since ficta are not a primary concern of the author, the entire problem was shunted aside and left as an exercise for the interested reader.

```
;
    BestFitPenalty←CurrentPenalty+Penalty;
    MaxPenalty←(BestFitPenalty*PenaltyRatio) MIN MaxPenalty;
    AllDone←TRUE;
!
```

We have now updated the value of the best fit penalty and the maximum penalty that a new solution should have, and have signalled the searcher to back up to the start again. Next we save the current state of the *CtrPt* array (which holds the soution transposed to C) in the *BestFit* array, adding back the original transposition:

```
;
    FOR v←1 STEP 1 UNTIL v1 DO
      FOR i←1 STEP 1 UNTIL TotalNotes[v] DO BestFit[i,v]←CtrPt[i,v]+BasePitch;
!
```

Now we print out the results for debugging and what not.

```
;
    PRINT("
        ! [",BestFitPenalty,"] ");
    FOR v←1 STEP 1 UNTIL v1 DO
      BEGIN
      PRINT("
            ");
      FOR i←1 STEP 1 UNTIL TotalNotes[v] DO
        BEGIN
        PRINT(BestFit[i,v]," ");
        IF i≠1 THEN PRINT("[",Check(i,BestFit[i,v]-BasePitch,v,v1,
                  (IF v=v1 THEN Species ELSE 1)),"] ")
        ELSE PRINT("[0] ");
        END;
      END;
    PRINT("
    ");
    END;
!
```

We need an array containing the legal melodic intervals in the order in which we want them to be checked. Whether this order is of great importance or not depends on how the searching is carried out. In the current scheme, the order doesn't matter much.

```
;
    PRELOAD_WITH
            1,-1,2,-2,3,-3,0,4,-4,5,7,-5,8,12,-7,-12;

      INTEGER ARRAY Indx[1:16];
 !
```

We need a procedure that examines the possible continuations and puts the best such continuations in order in a local array (on the stack). Each time the searcher lurches forward a note, a new recursive call is made allocating this and other arrays of local data. We can back up without losing previous state merely by returning from a call.

```
;
    RECURSIVE PROCEDURE Look(
            INTEGER CurPen,CurVoice,NumParts,Species;
            REFERENCE INTEGER Lim;
            INTEGER ARRAY Pens,Is,CurNotes);
    BEGIN
    INTEGER penalty,Pit,i;
    FOR is[CurVoice]←1 STEP 1 UNTIL 16 DO
      BEGIN
      pit←Indx[is[CurVoice]]+Ctrpt[CurNotes[CurVoice]-1,CurVoice];
!
```

Pit is the proposed new pitch for the current voice (*CurVoice*). For each such pitch we check how it affects the global penalty of the rest of the active voices

```
;
      penalty←CurPen+Check(CurNotes[CurVoice],Pit,CurVoice,NumParts,
            (IF CurVoice=NumParts THEN Species ELSE 1));
      SetUs(CurNotes[CurVoice],Pit,CurVoice);
      IF penalty<Lim
        THEN
          IF CurVoice<NumParts
            THEN
              BEGIN
!
```

We have a possible continuation for *CurVoice*. Now we need to find whether other voices can continue also given this pitch. Since the voices may not be moving together, we check new pitches only for those voices that are getting a new note at the current time.

```
;
              FOR i←CurVoice+1 STEP 1 UNTIL NumParts DO
                IF CurNotes[i]≠0 THEN DONE;
              IF i≤NumParts                        ! there is another voice needing a note;
                THEN Look(Penalty,i,NumParts,Species,Lim,Pens,Is,CurNotes);
              END
          ELSE
!
```

If the current voice is the last voice, then we save the current proposed overall continuation in the *Pens* array, and continue looking for more. When we finish this loop we will have all the best solutions in order in the *Pens* array.

```
;
            BEGIN
            INTEGER x,i;
            x←SaveIndx(Penalty,Pens);
            IF x>0
              THEN
                FOR i←1 STEP 1 UNTIL NumParts DO
                  Pens[x-i]←is[i]
            ELSE Lim←Lim MIN Penalty;
            END;
      END;
    END;
!
```

Next we need a procedure that calls *Look* (given above), and coordinates all the voices as the ensemble grinds toward the cadence.

```
;
    RECURSIVE PROCEDURE BestFitFirst(INTEGER CurTime,CurrentPenalty,NumParts,Species);
    BEGIN
    INTEGER i,j,Curmin,Lim,ChoiceIndex,NextTime,OurTime;
    INTEGER ARRAY Pens[0:FieldsNumFields],Is[1:NumParts],CurNotes[1:MostVoices];
    IF AllDone OR CurrentPenalty>MaxPenalty THEN RETURN;
!
```

AllDone is true after we have found a solution. At that point we exit all the calls currently active and start afresh with a new maximum penalty.

```
;
    ChoiceIndex←EndF;
    AllDone←FALSE;
    ARRCLR(Pens,infinity);
    ARRCLR(Is);
    ARRCLR(CurNotes);
    Branches←Branches+1;
    IF Branches>MaxBranch THEN RETURN;
    IF (Branches MOD 10) =0
      THEN
        MaxPenalty←MaxPenalty*PenaltyRatio;
!
```

If we just let the searcher run until a solution is found, it happens quite frequently that the search gets completely bogged down in a blind alley, spending immense amounts of time beating up against an impossible cadence situation. The branch counter gives us a way to jump out of such a situation. In addition, as we spend more and more time on a given attempt, we gradually reduce the acceptable maximum penalty somewhat like a person getting more and more frustrated as more effort is poured into a fruitless search.

```
;
    CurMin←infinity;
    Lim←BestFitPenalty-CurrentPenalty;
    NextTime←infinity;
    FOR i←1 STEP 1 UNTIL NumParts DO
      BEGIN
      OurTime←Onset[VIndex(CurTime,i)+1,i];
      IF OurTime≠0
        THEN NextTime←NextTime MIN OurTime;
      END;
!
```

We now know what the next note is overall. We set up the *CurNotes* array marking all voices that have an onset at the minimum next time. Each of these voices will then take part in the search for the best overall continuation.

```
;
    FOR i←1 STEP 1 UNTIL NumParts DO
      IF Onset[j←Vindex(NextTime,i),i]=NextTime
        THEN CurNotes[i]←j;
    FOR i←1 STEP 1 UNTIL NumParts DO
      IF CurNotes[i]≠0 THEN DONE;
    Look(0,i,NumParts,Species,Lim,Pens,Is,CurNotes);
!
```

Now we have the *NumFields* best continuations saved in the *Pens* array along with the associated penalties. We go through these continuations one at a time, trying to find one that will get us all the way to a cadence.

```
;
    CurMin←Pens[ChoiceIndex];
    IF CurMin≥Infinity THEN RETURN;
    WHILE NOT AllDone DO
      BEGIN
      IF CurTime<TotalTime
        THEN
          IF (curmin+currentpenalty≥MaxPenalty) THEN RETURN ELSE
      ELSE
          IF (curmin+currentpenalty≥BestFitPenalty) THEN RETURN;
!
```

We are still below the global maximum penalties, so set up the next continuation and give it a whirl.

```
;
      FOR i←1 STEP 1 UNTIL NumParts DO
        IF CurNotes[i]≠0
          THEN SetUs(CurNotes[i],indx[Pens[ChoiceIndex-i]]+Us(CurNotes[i]-1,i),i);
      IF NextTime<TotalTime
        THEN BestFitFirst(NextTime,CurrentPenalty+CurMin,NumParts,Species)
      ELSE                                     ! we've reached the end;
        SaveResults(CurrentPenalty,CurMin,NumParts,Species);
!
```

The variable *ChoiceIndex* points to where we are in the continuations. By decrementing it by field size, we move on to the next best continuation.

```
;
      ChoiceIndex←ChoiceIndex-Field;
      IF ChoiceIndex≤0 THEN RETURN;
      CurMin←Pens[ChoiceIndex];
      IF CurMin=infinity THEN RETURN;
      IF CurTime=0 THEN MaxPenalty←BestFitPenalty*PenaltyRatio;
      END;
   END;
!
```

The only thing left that is of any interest is the code that decides what rhythms to employ in fifth species. For simplicity's sake, we just load up an array with the legal rhythmic patterns and choose among them randomly. This approach obviously leaves much to be desired. Musical styles are differentiated more by rhythmic practices than melodic, and on a smaller level, it is the fluid handling of rhythm that makes a group of sounds a piece of music rather than a pedagogical exercise. However, this entire program ignores issues of phrasing or larger melodic and rhythmic structures. If we decide to carry this effort further in that direction, a much more complex decision and search mechanism will be required. Just as Fux's book was one step on the way toward Parnassus, this program, simple minded, even toy-like in many ways, may lead toward good music somewhere down the road. In any case, messing around with counterpoint is a pleasure in itself, so the author needed no further justification.

```
;
   INTEGER Seed;                 ! random number sequence seed;
   INTEGER ARRAY RhyPat[0:10,0:8],RhyNotes[0:10];
                                 ! array of legal rhythmic patterns;
   SIMPLE PROCEDURE FillRhyPat;
   BEGIN
   INTEGER i;
   RhyPat[0,1]←wholeNote;        ! for first species "rhythms";
   RhyNotes[0]←1;
   FOR i←1 STEP 1 UNTIL 2 DO RhyPat[1,i]←CASE i OF (0,halfnote,halfnote);
   RhyNotes[1]←2;
   FOR i←1 STEP 1 UNTIL 3 DO RhyPat[2,i]←CASE i OF (0,halfnote,quarternote,quarternote);
   RhyNotes[2]←3;
   FOR i←1 STEP 1 UNTIL 4 DO RhyPat[3,i]←quarternote;
   RhyNotes[3]←4;
   FOR i←1 STEP 1 UNTIL 3 DO RhyPat[4,i]←CASE i OF
           (0,quarternote,quarternote,halfnote);
   RhyNotes[4]←3;
   FOR i←1 STEP 1 UNTIL 4 DO RhyPat[5,i]←CASE i OF
           (0,quarternote,eighthnote,eighthnote,halfnote);
   RhyNotes[5]←4;
   FOR i←1 STEP 1 UNTIL 5 DO RhyPat[6,i]←CASE i OF
           (0,quarternote,eighthnote,eighthnote,quarternote,quarternote);
   RhyNotes[6]←5;
   FOR i←1 STEP 1 UNTIL 4 DO RhyPat[7,i]←CASE i OF
           (0,halfnote,quarternote,eighthnote,eighthnote);
   RhyNotes[7]←4;
   FOR i←1 STEP 1 UNTIL 6 DO RhyPat[8,i]←CASE i OF
           (0,quarternote,eighthnote,eighthnote,quarternote,eighthnote,eighthnote);
   RhyNotes[8]←6;
   FOR i←1 STEP 1 UNTIL 5 DO RhyPat[9,i]←CASE i OF
           (0,quarternote,quarternote,quarternote,eighthnote,eighthnote);
   RhyNotes[9]←5;
   RhyPat[10,1]←wholeNote;           ! for first species "rhythms";
   RhyNotes[10]←1;
   END;
   REQUIRE FillRhyPat INITIALIZATION;
!
```

Now a few simple procedures to clear and access the arrays associated with the rhythm finder.

```
;
    PROCEDURE UsedRhy(INTEGER n);
    RhyPat[n,0]←RhyPat[n,0]+1;

    INTEGER PROCEDURE CurRhy(INTEGER n);
    RETURN(RhyPat[n,0]);

    PROCEDURE CleanRhy;
    BEGIN
    INTEGER i;
    FOR i←1 STEP 1 UNTIL 9 DO RhyPat[i,0]←0;
    END;
    INTEGER PROCEDURE GoodRhy;
    BEGIN
    INTEGER i;
    i←10*RAN(seed);
    IF (CurRhy(i)>CurRhy(1 MAX (i-1))) THEN RETURN(1 MAX (i-1));
    IF (CurRhy(i)≤CurRhy(9 MIN (i+1))) THEN RETURN(9 MIN (i+1));
    RETURN(i);
    END;
!
```

The rest of the program merely stuffs the cantus firmus into the *CtrPt* array and sets the starting points of the various voices. We append several examples of counterpoint written by the program. Under each counterpoint note is the associated penalty. Currently the program has no provision for starting a melody with a rest, nor does it reward invertible counterpoint and imitation. It tends to let voices get entangled in each other, and makes no decisions about overall melodic shapes. Certain implied vertical clashes are not noticed (especially those involving tritones). Given these caveats the program does quite well in a reasonably short time.

Acknowledgments

I thank Jonathan Berger for his very valuable advice and criticisms.