Originally published as: Frances K. Dannenberg, Roger B. Dannenberg, and Philip Miller, "Teaching Programming to Musicians," in 1984 Proceedings of the Fourth Annual Symposium on Small Computers in the Arts (October 1984), pp. 114-122.

Teaching Programming to Musicians

Frances K. Dannenberg, Roger B. Dannenberg, Philip L. Miller

Computer Science Department, Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

A new approach has been developed for teaching programming to musicians. The approach uses personal computers with music synthesis capabilities, and students write programs in order to realize musical compositions. Our curriculum emphasizes abstraction in programming by the early introduction of high-level concepts and the late introduction of programming language details. We also emphasize abstraction by relating programming concepts to musical concepts which are already familiar to our students. We have successfully used this curriculum to teach Pascal to children and we are presently using it in a university-level course for composers.

We have developed a new methodology, designed especially to teach programming to music students. Students are challenged to produce artistic works through programming skill, using personal computers with music production capabilities. A key feature of our curriculum is that it allows students to use their existing musical knowledge as a basis for understanding computer programming. We have used this approach successfully to teach Pascal to children ranging in age from 9 to 16 years, and we are now using the curriculum as part of a college-level computer music course.

This approach is unique in several ways: First, it is designed specifically for artists. Programming is viewed as a means of creative expression rather than an abstract skill whose utility may be difficult to justify to an artist. Secondly, our approach is an inherently multi-media one. We have found that "listening" to a program's execution while reading the program is helpful in learning and debugging. Finally, we build upon existing musical knowledge. Musicians are familiar with the concepts of sequence, repetition, conditional selection, and procedural abstraction from the domain of music. We make use of analogy to teach the corresponding programming structures.

In Section 1, we present the origin and goals of this project. Then, in Section 2, we describe some earlier work and experience that guided our curriculum design. In particular, we wanted to teach what we call the *abstractionist methodology*. Section 3 then describes our specific curriculum design for teaching programming to musicians. Our experience with this curriculum is discussed in Section 4, and we present our conclusions in Section 5.

1. Background

We began with the goal of designing the curriculum for a "Computer Arts Summer Program." The program was to be held at the American Center in Paris, and was aimed at 12- to 16-year-olds. We planned to include computer music and computer programming instruction and to provide every student with a personal computer in the style of many "computer camps" held in the United States. It was also decided to integrate the music instruction as much as possible with computer programming.

We considered two approaches to the use of computers for music. First, we could present fixed, menu-oriented programs for drawing, composing, and computer-aided instruction. Rather than writing their own programs,

students would manipulate parameters in existing programs. Alternatively, we could write interfaces to graphics and sound synthesis devices so that students could create music by writing their own computer programs. We decided to concentrate on the latter approach: that is, teaching students how to program in order to produce music.

2. Previous Work

We know of no work that addresses the needs of teaching programming to the musician in particular. However, there is a wealth of literature concerning programming methodology and pedagogy in general. Of particular interest are papers by Perlis[1], Dijkstra[2], and Hoare[3], which discuss the importance of various forms of abstraction to programming. We call the general approach advocated by these authors the *abstractionist methodology*. Because of its importance to our curriculum for teaching musicians, we describe it here in some detail.

2.1. Abstractionist Methodology

We recognize three principal levels of programming abstraction: the control structure level, the procedure level, and the data structure level.

Abstract Control Structures. The most familiar level is that of control structures[4]. This is essentially the structured programming movement of the 1960s, with **do-while**, **if-then-else**, etc. In contrast to the **goto**, which may be used to create arbitrary flow of control, control structures should have single points of entry and exit, and they should indicate the programmer's intention, for example, to iterate a sequence of statements.

Procedural Abstraction. The second level of abstraction is abstraction at the procedural level. The idea is that problems are too complex to be thought about all at once, so we think about them hierarchically. To illustrate, we will borrow from a textbook[5]. Consider the task of grocery shopping. This high-level task can be divided into smaller tasks (subtasks) in many ways. Let's say we decide to decompose it into two subtasks: *generating a shopping list* and *buying all the items on the list.*

We continue now with the subtask of *generating a shopping list.* Likewise, it may be decomposed in a variety of ways. Assume our solution is first to obtain a pencil and paper, followed by examining the kitchen cupboards for some idea of what is needed, and finally, to consult the spouse for a contribution to the list. Generating the shopping list could be done in other ways. For example, one might simply delegate the task to one's spouse, cook, maid, etc.

With our shopping list firmly in hand, we can consider the subtask of *generating the shopping list* completed. We now turn our attention to the other major subtask, *buying the items on the list.* This, of course, could also be done in a variety of ways. Let's say we wish to do it in the following manner: go to the grocery store; collect the items on the list; pay the cashier; and, finally, return home with the groceries.

This leaves out many details of the acquisition phase. For example, we named a subtask *collect items on list*, however, we have said nothing about how this is to be accomplished. How are we going to search the store for the items on the list? Are we going to use a shopping cart,

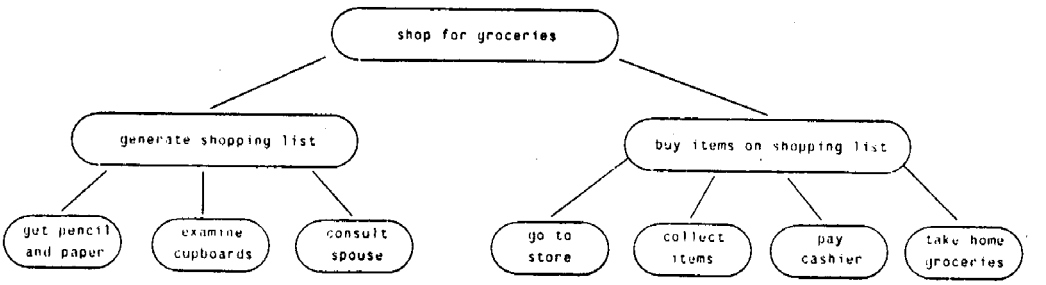


Figure 2-1: Figure Showing The Tree Structure Of Problem Decomposition

or perhaps just a shopping basket? We named another subtask of *paying the cashier.* This might be done with cash. It might also be done with a personal check, or with manufacturers' coupons, a charge card or some combination of these methods.

We see that this technique generates a hierarchical solution to the overall task. Although we've expanded only two levels, it gives enough of the idea for you to see how it is done. In programming, tasks are similarly divided into components which are then written separately.

Data Abstraction. Data abstraction is the business of thinking of a problem as a set of objects and the operations that are performed on those objects. Consider a payroll system. The problem is to maintain information on the employees of a company. The information that is kept on each employee includes such items as name, rate of pay, hours worked, whether or not the employee is participating in the company's group insurance plan. We can think of this information as an index card of information. Because the company has a number of employees, the index cards are arranged alphabetically into a shoe-box of index cards. Thus the abstract structure for the payroll problem is a shoe-box of index cards.

The second aspect of data abstraction concerns the operations on the structures. There must be a way to thumb through the cards, searching for a particular one. There must be way to copy information from a card and a way to change the information that is on a card. A card must be added when a new employee is hired. A card must be removed when an employee is terminated.

In programming, the box of index cards would be represented by a data-structure. It is desirable, in order to reduce program complexity, to confine the details of this data-structure to only a small part of the overall program. This is accomplished by writing procedures for each of the desired operations. If an operation is complex, it might be implemented by a package of procedures.

We find these three levels of abstraction to be at the core of good programming practice. They facilitate design of software that is at once verifiable, implementable, debuggable, and extendible. They dovetail neatly with the ideas of information hiding and strong typing. They represent current thought in software engineering.

2.2. Abstractionist Pedagogy

Recently, a few educators have begun to adopt a pedagogical style that is designed specifically to teach the abstractionist methodology. This style does not have a name in computer science, but we will call it the *abstractionist pedagogy.* Among the people with whom we are familiar, Bob Floyd is credited with the idea, which has since been applied in several textbooks[6, 7, 5, 8].

Among the key features of the abstractionist pedagogy are the early introduction of high-level concepts and the late introduction of programming language details. This encourages a hierarchical approach to problems, beginning with the highest level. Programming instruction begins with the introduction of a handful of pre-written procedures. The student writes his first program simply by calling these procedures sequentially. Next, the student is given a technique for writing new procedures, built from sequences of the primitive procedures mentioned above. Control structures are then introduced, and finally, a full programming language is presented to the student.

Standing head and shoulders above the rest in successfully executing this pedagogy is Richard Pattis. In the marvelous little book, Karel the Robot: A Gentle Introduction to Programming, students learn to manipulate a robot, Karel, using primitive procedures such as *Move* and *TurnLeft.* The robot is simulated on a standard CRT. Tasks are designed for the student, such as programming Karel to step over a hurdle or to escape a maze. As the student learns more powerful techniques of programming, successively more general and elegant programs that control the robot are written.

We believe that this task domain, one that is visual and tactile, is a good one for introducing programming methods. Unlike the domain of numeric calculations (the unfortunate standard fodder for beginning programmers)

the Robot world introduces no intellectual barriers to the student. It provides an environment that is at once intuitive and rich with analogies that can be exploited for introducing and fixing the rudiments of sound programming methodology. The book and the approach are now being adopted in a number of high schools in the U.S. and abroad, in part due to the positive recommendations of the College Board's Advanced Placement Computer Science Development Committee[9], [10].

2.3. Assumptions and Prejudice

We designed our curriculum for musicians with several assumptions in mind:

- The first is that the abstractionist methodology is sound and should be taught to beginning programmers.

- The second assumption is that there is a best way to teach this methodology. The abstractionist pedagogy has been used successfully at Carnegie-Mellon and elsewhere in programming courses.

- The third assumption is that musicians can learn the programming methodology. It is sometimes held that the mathematically-oriented students (engineering and science) are able to learn programming methodology, while artists are either unable or much less able to do so. It is clear, however, that some musicians are excellent programmers. Some are respected computer scientists. We decided that the best approach was to assume that for the purposes of programming, musicians as beginning programming students are no different from any other group of beginners.

- The fourth assumption, as suggested in our description of the course, is that the best way to introduce the concepts of programming methodology is to tie these concepts closely to a knowledge base that is familiar. In introducing a new concept, a successful teaching method is often to explain it by its analogy to some more familiar concept. In our course, we apply this to teaching programming abstraction, explaining programming structures to students by analogy to similar hierarchical organizations in music, with which they are familiar.

Thus far, we have introduced a number of important ideas. We have specified a programming methodology as the correct one to teach. We have talked about how to teach that methodology in terms of subject matter, texts, and software. How this all manifests itself in terms of teaching programming to musicians is the subject of the next section.

3. The Abstractionist Approach in a Musical Setting

When we began to design our programming course for musicians, we looked for musical analogues to the concepts we wanted to teach: sequential execution, procedural abstraction, and control structure abstraction. We were quite pleased to find musically meaningful analogies for all of these concepts. Below, we describe how each concept was presented to our musician/programmers.

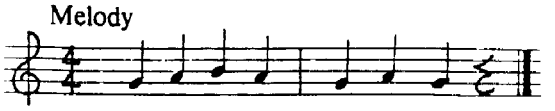
For the introduction of procedural and control abstraction, we wanted to keep programs as simple as possible, avoiding issues such as parameters, input/output, and synthesizer interfaces. Taking Karel the Robot[8] as a model, we defined a set of parameterless procedures to play the notes of an octave scale and to produce silence[1]. A few more procedures were added to produce sound effects, and an *include file* mechanism was used to hide the definitions of all of these procedures. The use of personal computers made it possible for each student to have a machine that could edit, compile, and execute programs using these procedures. Each machine could also synthesize appropriate sounds.

3.1. Sequence/Melody

The first programming lesson consists of a simple melody and an explanation of how to translate the melody into a program. For example, the following melody:

---

[1]The procedures are *PlayDo*, *PlayRe*, *PlayMi*, *PlayFa*, *PlaySol*, *PlayLa*, *PlaySi*, *PlayDo2* (an octave higher than *PlayDo*), and *Rest*. The names of these procedures were chosen to avoid a clash between the Pascal reserved word "**do**" and the solfege syllable "Do".



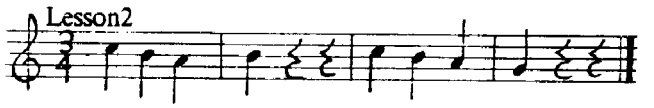
would be translated to:

```
program Melody;
{include definition file here}
begin
    PlaySol;
    PlayLa;
    PlaySi;
    PlayLa;
    PlaySol;
    PlayLa;
    PlaySol;
    Rest
end.
```

Students are encouraged to compose their own melodies and to program the computer to play them.

3.2. Procedures/Phrases

For the next lesson, an example is chosen that includes several occurrences of a musical phrase. The example is translated into Pascal, and it is observed that the program contains a duplicated sub-sequence of commands. Students are shown how to build a named procedure from the sub-sequence. For example, the following melody:



could be rendered as follows, using a procedure to implement measures 1 and 3:

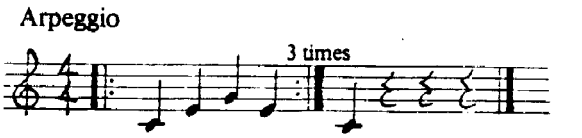
```
program Lesson2;
{include definition file here}

    procedure DoSiLa;
    begin
        PlayDo2;
        PlaySi;
        PlayLa
    end;

begin
    DoSiLa;
    PlaySi;
    Rest;
    Rest;
    DoSiLa;
    PlaySol
end.
```

3.3. Loops/Repetition

After programming a composition using procedures, we turn to basic control constructs. The loop is the first construct considered; its musical analogue is the *repeat.* At this point, we consider only the **for** loop. The next example illustrates the use of the **for** loop to program a musical repeat:



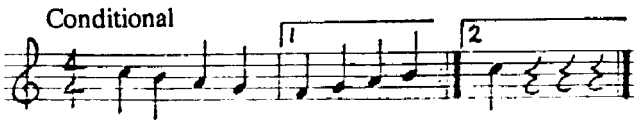

```
program Arpeggio;
var i: integer;
{include definition file here}
begin
    for i := 1 to 3 do
        begin {repeated measure}
            PlayDo;
            PlayMi;
            PlaySol;
            PlayMi
        end;
    PlayDo {the last note}
end.
```

For this lesson, Pascal requires a declaration for the loop control variable. Since variables have not yet been introduced, we avoid the issue by describing the declaration as a "magic incantation" to be explained later. A loop construct that implicitly declares its control variable[11] would be preferable for teaching purposes.

Perceptive students will realize that a musical repeat can be implemented by programming the repeated music as a procedure and calling it several times. An interesting work to discuss at this point is *Vexations*, by Erik Satie, which consists of a short musical statement to be repeated 840 times!

3.4. Conditionals/First and Second Endings

The next lesson concerns conditional execution. The analogue in music is the first-and-second ending notation. Consider the following example:




```
program Conditional;
var i: integer;
{include definition file here}
begin
    for i := 1 to 2 do
        begin
            PlayDo2;
            PlaySi;
            PlayLa;
            PlaySol;
            if i = 1 then
                begin {first ending}
                    PlayFa;
                    PlaySol;
                    PlayLa;
                    PlaySi
                end
            else
                begin {second ending}
                    PlayDo2
                end
        end
end.
```

Students should be encouraged to experiment with conditionals in non-traditional musical structures. For example, conditionals could be used to introduce variations at several points in a repeated note sequence.

3.5. Parameters

Until now, no procedures have been parameterized. This simplifies the presentation of control constructs and procedures, but imposes rather severe limitations on the variety of sounds that can be programmed. In the next several lessons, students are taught how to call parameterized procedures and how to declare them. By this time in the course, students recognize the need for more subtle control over sound, and welcome the introduction of parameters.

Predefined procedures called Note and Rest are used to introduce parameters. The Note procedure takes arguments for frequency, amplitude, and duration; for example Note(440, 100, 50). The Rest procedure has one argument, duration; for example Rest(90).

Students are then taught how to define their own parameterized procedures. At this point, they have the programming skills necessary to create interesting pieces.

Advanced students will want more direct access to the sound generation hardware than that provided by the Note procedure. In our case, we use a fairly sophisticated synthesizer interface capable of independent time-varying frequency, amplitude, and waveform control over 16 oscillators. The synthesizer interface illustrates data abstraction. Procedures are used to manipulate some underlying structure (the synthesizer) in order to hide irrelevant details of the structure. Students are encouraged to develop their own data abstractions at the next higher level in order to obtain a control interface that is appropriate for their composition. For example, a procedure named Gliss could be written in terms of primitive frequency controls in order to implement a musical *glissando.*

4. Results and Discussion

We taught a 15-day course, where students had a total of 4 hours per day for instruction and access to computers. Although the course was intended for 12- to 16-year-olds, the actual range was 9 to 16 years. All of the students were able to develop programs that used procedural abstraction, loops, and conditionals. For example, one student, who had no previous computing experience, wrote an 83-line program to perform a piece with the structure *ABA* (see the appendix). The *A* section was implemented as a procedure with an internal structure of the form *abaca.* This was accomplished using a **for** loop to iterate 3 times, with a conditional to insert *b* after the first iteration, and *c* after the second iteration. This program used a variation of the Note procedure to give control over the rate of attack and decay of each note.

4.1. Music as a Concrete Programming Task

As expected, students understood the programming tasks immediately since they came from familiar intellectual territory. This allowed students to concentrate on the solutions to the problems rather than trying to understand the problems themselves.

4.2. "Listening" to Program Behavior

As with the domain of Karel the Robot, which can be simulated on a CRT, we found music to be attractive for programming because it was possible to follow program behavior quite closely. This was true in part because program behavior was slowed to a musical pace. Also, one could hear the result of each program step; consequently, one did not often need to deduce a program's behavior from its final output. Rather, the entire program execution was transparent, and problems could be isolated without a painful debugging process.

In addition, we found that the music domain has specific advantages over Karel. First, music is an ideal medium for transmitting large amounts of information about program behavior to our musicians. It is also possible to read a program listing visually while simultaneously following program execution aurally. This was valuable in helping students to learn the association between program statements and their actions.

4.3. Motivational Factors

Our students discovered that making music with computers is also fun and exciting. As students completed their assignments, they would perform their pieces for the class, often receiving applause and compliments. Students were highly motivated to finish their assignments!

4.4. Extension to Other Domains

Based on our experience, we feel that other domains could serve as an excellent basis for the abstractionist pedagogical style. The style is appropriate for various types of music synthesizers[2] , but it might also be considered for the new, low-cost speech-synthesis devices. Another interesting domain is that of computer graphics. The "turtle graphics" interface is an example of an appropriate set of primitives[12]. In another application, Harry Holland at Carnegie-Mellon University is using our approach to teach Pascal to artists. His students use a color graphics display and program in terms of primitives like Box, Circle, and Line. Architectural drawing is another possible domain. Finally, a mechanical robot is being constructed at Carnegie-Mellon University, based on Karel, to make the programming task more exciting.

[2]For this reason, we do not describe our lowest level synthesizer interface in greater detail here.

At Carnegie-Mellon University, the programming pedagogy is reinforced not only by the Pattis text and the Miller and Miller text, but also by software that was written with an eye to the abstractionist methodology. GNOME software is built so that procedural and control abstraction are the natural form of program construction. Details of syntax and some details of semantics (e.g. the order of procedure declaration) are issues for the programming environment, not for the programmer.[13]

We are currently using the abstractionist pedagogy as part of a computer-music course for college students. In this course, however, we introduce parameterized procedures at the beginning so that students have more music-making capabilities from the start.

5. Conclusions

We have presented our view of the proper pedagogical style for teaching the abstractionist methodology. The approach has been used successfully at Carnegie-Mellon University and elsewhere.

It was gratifying to discover that the approach can be adapted quite well to the musical domain and that musicians can indeed learn to program with the abstractionist methodology. In fact, music has specific advantages, including familiarity with the domain, program behavior that is audible, and a strong motivation to "compose" programs.

It is interesting to compare our experience teaching grade-school level students to that of teaching university students. Our goal with the grade-school students was primarily to teach programming, while in the university course, programming skills are primarily a means of realizing a composition. One conclusion is that there are limits as to how far one can integrate the teaching of music and programming. For example, the programming tasks described in this paper have little musical value to a university-level course in computer music, but the programming concepts are an important foundation for more sophisticated tasks. The problem is that a "toy" domain like Karel the Robot is ideal for teaching programming, but toy music domains are not attractive to serious musicians. We believe part of this problem can be solved by a better choice of synthesizer interface, and we intend to experiment further in future courses.

The number of musician/programmers is small, but the field of music has already felt their impact. It will be interesting to watch what musicians do with programming skills as they become more widespread.

6. Acknowledgements

It our pleasure to acknowledge a number of people and organizations whose contributions made this project a success. Raj Reddy deserves credit for the concept of a computer arts camp for children. Colette Wilkins was indispensable in teaching as well as in a multitude of other tasks essential to the success of the course. Judith Pisar, Henry Pillsbury, Alex Mehdevi and the staff at the American Center obtained equipment and handled innumerable problems in preparation for the course. Computer equipment was loaned to us by Atari France and Apple. Carnegie-Mellon University's Summer Studies funded the courseware development, which was programmed by Linda Isaacson, Richard Sean Keegan, Robert Rose, Peter Shell, and Mark Wilkins.

References

1. Perlis, Allan, "A First Course in Computer Science," , 1965.

2. Dijkstra, E.W., *Structured Programming*, Academic Press, 1972, ch. Notes on Structured Programming.

3. Hoare, C.A.R., "Proof of a Program: FIND," *CACM*, Vol. 14, No. 1, January 1971, .

4. Dijkstra,E.W., "GOTO Statement Considered Harmful," *CACM*, Vol. 11, No. 3, March 1968, .

5. Philip L. Miller and Lee W. Miller, *Computer Science, The First Course*, Random House, 1985.

6. D. Cooper and M. Clancey, *Oh Pascal*, Norton, 1982.

7. Arthur Keller, *A First Course in Computer Programming Using Pascal*, McGraw-Hill, 1982.

8. Pattis, R., *Karel the Robot, A Gentle Introduction to the Art of Programming*, John Wiley and Sons, 1981.

9. The Advanced Placement Computer Science Committee of the College Board, "Advanced Placement Course Description: Computer Science," , 1984.

10. The Advanced Placement Computer Science Committee of the College Board, "Teacher's Guide to Advanced Placement Courses in Computer Science," , 1984.

11. William Wulf, D. B. Russell, and A. Nico Habermann, "Bliss: A Language for Systems Programming," *CACM*, Vol. 14, No. 12, December 1971, .

12. Harold Abelson and Andrea diSessa, *Turtle Geometry: the computer as a medium for exploring mathematics*, MIT Press, 1980.

13. David B. Garlan and Philip L. Miller, "GNOME: An Introductory Programming Environment Based on a Family of Structure Editors," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM, May 1984, Published as SIGPLAN Notices 19(3) and Software Engineering Notes 9(3).

Appendix

Listing of a Student Program

```
program Param;
var
{note: this is the interface include file: }
(*$ID2:INTER.DEF*)

   procedure Note(Pitch, Attack, Decay,
                              Amplitude: integer);
   const Voice = 0;
   begin
      WaitVoice(Voice);
      FDelay(Pitch, 0, Voice);
      ARamp(Attack, Amplitude, Voice);
      ARamp(Decay, 0, Voice)
   end;

   procedure Bizarre;
   var C: integer;
   begin
      for C := 1 to 3 do
         begin
            Note(700, 100, 100, 100);
            Note(750, 97, 97, 97);
            Note(800, 94, 94, 94);
            Note(850, 91, 91, 91);
            Note(900, 88, 88, 88);
            Note(950, 85, 85, 91);
            Note(1000, 100, 100, 90);
            Note(1250, 90, 90, 100);
            Note(1500, 100, 110, 100);
            Note(2000, 90, 89, 79);

            if C = 1 then
               begin
                  Note(850, 75, 77, 75);
                  Note(825, 78, 77, 78);
                  Note(800, 76, 79, 78);
                  Note(4000, 70, 120, 110);
                  Note(4700, 120, 70, 110);
                  Note(800, 89, 70, 75);
                  Note(730, 95, 84, 77);
                  Note(888, 100, 110, 95);
                  Note(2540, 127, 71, 120);
                  Note(1700, 120, 120, 90);
                  Note(990, 110, 127, 75);
                  Note(4000, 90, 89, 97);
                  Note(4500, 85, 90, 100);
                  Note(4700, 90, 110, 100);
                  Note(5000, 100, 75, 89);
                  Note(500, 120, 75, 100);
                  Note(700, 90, 90, 100);
                  Note(1000, 85, 90, 86);
               end
            else if C = 2 then
               begin
                  Note(5000, 100, 100, 90);
                  Note(9000, 127, 127, 90);
                  Note(8500, 127, 127, 90);
                  Note(8000, 127, 127, 90);
                  Note(7500, 127, 127, 90);
                  Note(7000, 127, 127, 90);
                  Note(2700, 95, 90, 100);
                  Note(2000, 90, 99, 110);
                  Note(900, 85, 90, 90);
                  Note(700, 85, 80, 95);
               end
         end
   end;

begin
   MusIni;
   Bizarre;
   Note(350, 65, 65, 70);
   Note(325, 65, 65, 74);
   Note(300, 65, 65, 77);
   Note(375, 127, 127, 90);
   Note(400, 120, 120, 95);
   Note(385, 100, 100, 85);
   Note(300, 95, 95, 85);
   Note(250, 90, 87, 94);
   Note(215, 70, 78, 66);
   Note(207, 50, 56, 90);
   Note(200, 80, 76, 88);
   Note(189, 79, 87, 85);
   Note(206, 80, 75, 76);
   Note(200, 76, 47, 69);
   Bizarre;
   ARamp(0, 0, 0)
end.
```