4. A musical program auralisation tool

*To obtain the value of a sound, a movement, measure from zero. (Pay attention to what it is, just as it is.)*

—John Cage: *Four Statements on the Dance*

4.1. Introduction

Program auralisation involves the mapping of run-time events in a program to sound patterns and several systems that do this were reviewed in chapter 2. However, none of these systems attempted to define the auralisations within a formal musical framework. We have argued in chapter 3 and elsewhere ([7, 152]) that music promises much as a communication medium to the process of human-computer interaction with its ability to carry complex information in parallel.

To determine whether music offers advantages to program auralisation (and by extension to visualisation) it was necessary to construct a new software tool that would enable the auralisation of programs within a musical context. The approach taken was incremental. First, a prototype auralisation system was constructed and tested with a rudimentary set of motifs. Then, the motifs were reworked using a set of structured design principles, but without focusing on the cognitive and music-theoretic aspects. These motifs were evaluated in two pilot studies.
4.2. The prototype CAITLIN system

4.2.1. Musicality of auralisations

While previous attempts at program auralisation have proved that it is possible to represent program run-time state in sound, issues of musicality have not been addressed. The extent of the music content of auralisations has been to map certain program events and data values to pitches without regard to the musicality of those pitch selections. In effect, pitch mappings have been made as a function of data values without any constraints being imposed by a music syntax filter. Thus, no heuristics for governing the design of musical auralisations have yet been developed. This meant that any musical auralisation tool would have to be designed from first principles without reference to previous experience.

Consequently, the approach taken was first to develop a prototype system and then to subject it to experimentation to ascertain whether the auralisations were comprehensible. The lessons learnt from the initial experiments were used in the redesign of the system, this new version being the one used in the full study.

The system constructed is the Computer-Audio Interface To Locate Incorrect Nonsense, or CAITLIN [7, 152]. CAITLIN is a non-invasive pre-processor that allows a novice programmer to auralise a program written in Turbo Pascal®.

An important consideration for any auralisation system is that it be easy to use. Therefore, it is desirable that as far as possible the process is automated and that the user be protected from having to compose any music. We criticised the existing auralisation systems for requiring users to specify their own auralisations (whether by using a scripting language or a component-based visual language). For such a system to be useable by non-musically-educated programmers and novice programmers in particular, the details of auralisation specification and design need to be as removed as possible. Therefore, we decided to pre-define the auralisations in the CAITLIN system so to make the auralisation step as simple as the compilation step.

4.2.2. Technical information

CAITLIN is a small program (approximately 12,000 lines of code) implemented in Borland® Pascal 7.0 using the Turbo Vision extensions for the interface. The ex-
ecutable is an MS-DOS Protected-Mode program. The MS-DOS platform was chosen so that an interface of the same type as the Borland® Integrated Development Environment (IDE) could be given to the system. This allows CAITLIN to have similar interface characteristics to the compiler environment familiar to the subjects in the studies. This obviates the need for the novices to learn a new environment simply to use the system. Figure 4.1 depicts the basic architecture of the system showing the various functional units and how they interact.

**Figure 4.1 CAITLIN’s architecture**
This diagram shows the main components of the CAITLIN program. In reality, the Formatter is external to the program and need not be invoked, allowing the auralised source to be compiled without formatting. As one would not normally need to look at the auralised source code, the formatting option can be turned off. Also, the Compiler is also external to the system and is, in fact, the Turbo Pascal command-line compiler (BPC.EXE).

CAITLIN’s musical output is achieved by sending MIDI data to a multi-timbral synthesiser via the MIDI port on a Sound-Blaster®-compatible sound card. Wherever possible, the output conforms to the General MIDI [26] standard commonly employed by many of the newer tone generators.

Smith criticised MIDI for being too inflexible for music research work as it was designed primarily for the production of pop and rock music [146]. Whilst this may be a restriction where the auditory display requires any kind of signal processing, such as effecting changes in timbre in real-time, it would not appear to pose a prob-
lem for this work. This is because CAITLIN relies on musical structure to convey information, different timbres being used to aid disambiguation. The kind of musical data required is easily communicated by the MIDI language.

CAITLIN is a portable system. By using a MIDI interface and relying on music rather than signal processing to carry information, the system can be implemented on a relatively modest (and increasingly more common) hardware platform comprising a personal computer and sound card with a General-MIDI-compatible instrument set.

The auralisations employed by CAITLIN are exclusively musical for the reasons discussed in chapter 3. Musical structures can be devised to provide a grammar by which information can be unambiguously presented. Taking this further, we can make use of the common musical device known as the leitmotif\textsuperscript{12} (usually shortened to motif) to represent recurring features. A motif is generally understood "as a core of pitch and rhythmic information which may be subjected to variation by a range of musical transformations" [96]. Motifs in auditory display have previously been suggested and discussed by Sonnenwald et al [148].

CAITLIN is non-invasive. That is, the source code that provides its input is left unchanged by the pre-processing phase. The auralisations are brought about by adding calls to library routines to a copy of the program held in memory. It is this elaborated copy of the source program that is compiled to produce an executable image that incorporates the auralisation routines.

One restriction CAITLIN places on the programmer is that it will only accept a source program that is free of syntax errors, that is, a program that successfully compiles. The reason for this is simple: CAITLIN is designed to assist with debugging executable programs and not to help the programmer compile code.

On running CAITLIN (see Figure 4.2) the user is presented with a screen similar in concept and layout to that of the Turbo Pascal Integrated Development Environment (IDE). A menu option allows the user to load a source program into memory which is then parsed and stored in token form in memory. Once a program has been loaded and parsed, the user may opt, via a menu (see the open menu on Figure

\textsuperscript{12} One well-known musical work that employs this device is Prokofiev's "Peter and the Wolf".
4.2), to auralise and then compile and run the auralised program, or musicode. Compilation is achieved by writing the amended program to a file and invoking the compiler in command line mode.

![Figure 4.2 CAITLIN—main screen](image)

Here we see a program that has been loaded into CAITLIN. The Program menu allows the program to be auralised, compiled, and then run without having to leave CAITLIN. The other menu options allow various system settings to be changed. Notice how the interface looks like the Turbo Pascal IDE.

Auralisation is carried out at the code construct level. That is, a WHILE loop is auralised in one way and REPEAT, FOR, CASE, IF...THEN...ELSE and WITH constructs in others. For each construct the user may elect to change various aspects of its auralisation, such as instrument, note length, MIDI channel, loudness and, for FOR loops, the musical scale type (see Figure 4.3).

The speed at which the music is heard is controlled by a user-definable tempo variable. Tempo is defined as the number of beats, or quarter notes, sounded per minute. All options can be saved to a configuration file.
The Setup menu option allows the user to alter various parameters of the construct auralisations. In this example, the user can select the timbre, note duration, scale type, MIDI channel, and velocity (volume) for the \texttt{FOR...TO} and \texttt{FOR...DOWNTO} loops.

4.2.3. Points of interest

For each construct the auralisation comprises three basic parts: a musical signature tune (motif) to denote the commencement of the construct, a musical structure representing the execution of the construct and a signature to signal exit from the construct.

The contents of the musical structure within the construct will depend upon the construct's characteristics. Different constructs have different features which will be represented in various ways. To this end we have introduced the notion of the point of interest (POI). A point of interest is a feature of a construct the details of which are of interest to the programmer at execution. It is similar to, but not the same as, DiGiano and Baecker's audiopoints [53] and Mathur's possible positions for an auralisation [20, 21, 111]. For example, the \texttt{FOR} construct possesses three POIs:

1. entry to the loop;  
2. execution of the iterated statement;  
3. exit from the loop.

Similarly, in its basic form the \texttt{IF} construct has four POIs:

1. entry to the \texttt{IF} construct;  
2. evaluation of the conditional expression;  
3. execution of selected statement;  
4. exit from the \texttt{IF} construct.
We can see, depending on the result of the conditional expression, that POI-3 of the IF statement, may not be heard. We can also see that for each construct type the first and last POIs always denote entry to and exit from the construct respectively.

To enable the listener to distinguish POI-1 of a FOR loop from POI-1 of a WHILE loop we have defined a short signature tune for each construct type. Thus, when a program is auralised the tune associated with FOR statements is inserted prior to each FOR loop and so on. A construct's last POI is auralised by playing a variation of its signature tune (such as playing it in reverse).

By defining a program in terms of its points of interest we can build up an understanding of how each program element and hence, by extrapolation, the whole program should sound. For example, we know that each FOR loop will be heard as a sequence of:

- playing of signature tune, followed by
- repetition of music denoting iterated statement execution, followed by
- playing of modified signature tune.

```
PROGRAM Demo;
VAR
  cnr1, cnr2 : Integer;
BEGIN { Demo }
  FOR cnr1 := 1 TO 7 DO
    FOR cnr2 := 15 DOWNTO 1 DO
      Writeln (cnr1, ':', cnr2) ;
END { Demo }.
```

**Figure 4.4 Source code example**

This is illustrated by Figure 4.4 and Figure 4.5. Figure 4.4 is a listing of a simple program employing two FOR loops. Figure 4.5 shows the auralised version of this program. We can see that the auralisation employed in this example is quite simple in nature. Code has been inserted such that each iteration of the outer loop is represented by a pitch of an ascending scale, the scale type being selected by the user (see Figure 4.3). Likewise, the inner loop plays a descending scale as the loop counter in this case is decremental rather than incremental. The sophistication of the various auralisations will increase as research and experimentation determine what musical structures convey information the best.
Chapter 4. A musical program auralisation tool

Figure 4.5 Auralised code example
CAITLIN has added the necessary code to allow the program to be auralised. Careful inspection reveals the original code is still intact, and that extra code has been inserted. In normal operation the user would have no need to look at this auralised code.

It should be borne in mind that the source version of the musicode (Figure 4.5) is not intended to be examined by the programmer. In the same way that intermediate object code is generated by compilers as input to the linker prior to generation of an executable image, the source musicode is an intermediary between CAITLIN and
the language compiler. It must be stressed that the programmer using CAITLIN will only have access to the original and unchanged source code.

With suitable tuition in the auralisations employed by CAITLIN the programmer can thus develop a hypothesis of the music to be produced by his program. When the auralised program sounds different than expected then two possibilities are indicated: either the programmer must refine his hypothesis or a bug exists in the program. The point at which the auralisation deviates from the expected pattern should indicate the region of code in which a bug resides.

4.2.4. Earcons and program auralisation

As earcons are designed around short musical motifs, the question arises as to their suitability for use in auralisation systems. Earcons have been used successfully to communicate information to users about system objects and state changes. Interaction tasks, such as drag-and-drop, have been shown to be made more efficient by the addition of earcons [30]. The length of earcons can be reduced by means of sounding the component parts of an interaction in parallel. For instance, for the file-create action, instead of sounding the file earcon followed by the create earcon, the same information can be presented using a parallel earcon that plays both component earcons at the same time.

The characteristics of earcons may be summarised as:

- **Atomic**—When representing events, those events do not persist over time, and are thus indivisible.

- **Change of state**—They signal the occurrence of an event (change of state) but they do not signal persistence of state.

- **Sequential**—Earcons are designed to be played sequentially and not in a nested fashion. By this we mean that when an earcon is heard to start we do not expect to hear another start until the present one is ended. This applies equally to compound and parallel earcons.

These features make earcons unsuitable for use in program auralisation. Whilst it is true that certain program domain events could be mapped successfully onto earcons, other aspects of running programs make earcon use problematic. The Pascal constructs all persist over time. It is not sufficient to represent a loop construct
using an earcon because all the contextual and timing information is lost. The earcon would describe the occurrence of the loop construct as an atomic event that has no duration. However, as programmers we are interested in knowing when the loop begins, in the execution of its body (and evaluation of the controlling condition) and in its eventual termination. Furthermore, it is usual for constructs to have other constructs nested within them. To make sense of the auralisations listeners need to know that the loop currently being heard is nested within an outer loop that is, of necessity, still active.

The basic problem with earcons for this application then, is that they do not persist over time and thus allow nesting. The approach we have taken of mapping points of interest to individual motifs which are harmonically and rhythmically related solves this problem. The use of a drone to signal presence in a construct body supplies the added context needed to be able to understand the relationships of nested constructs.

4.3. Testing the concept

Before embarking on a thorough design of auralisation motifs a small experiment was conducted to test whether the basic concept of musical auralisation made sense to programmers. The prototype auralisation motifs used in this study were not designed with regard to the structural categorisations discussed above. For each construct a unique motif was arbitrarily assigned, each of the motifs having no structural resemblance to the others of different sub-classes. A metaphoric mapping was used for the IF and IF...ELSE statements. A pitch bend was applied to mimic the rising and falling inflection of the human voice when posing and answering questions. In fact, as several of the subjects in the experiment observed, this ended up sounding like a comical ship’s fog horn.

The auralisations were as follows:

- **REPEAT** – twinkle twinkle little star
- **WHILE** – a plagal cadence (the amen at the end of a hymn)
- **FOR** – three notes on a piano that say ‘dah dah dit’ followed by an upwards or downwards scale depending on the direction of the loop followed by the closing ‘dit dit dah’.
• IF & Else – the ‘fog horn’
• CASE – not used in this study

The purpose of these rough and ready auralisations was to determine whether the musical auralisation approach of mapping points of interest within constructs to musical events could be used by programmers to infer information about executing programs.

4.3.1. Subjects

This first experiment [152] involved eight subjects, all of whom were lecturers at Liverpool John Moores University involved in the teaching of programming. None of the subjects reported any special musical skills or experience other than the general enjoyment of music through listening to the radio and other media. Two subjects even claimed to be tone deaf, by which it is assumed they meant that they cannot sing well or at all.

4.3.2. Procedure

The experiment was conducted in two parts, a familiarisation session followed by a test session. The familiarisation involved presenting to the subjects auralisations of ten short Pascal programs. Each example was accompanied by a narrative description of the program and the program’s source code was available on request. Each of the ten sample auralisations could be repeated as many times as required. The subjects had no interaction with CAITLIN, as the auralisations were presented by playing back recordings taken from CAITLIN prior to the session.

Following the familiarisation session the subjects were presented with nine test programs which contained a total of fourteen individual construct auralisations. For each exercise they were asked to describe the structure of the program it represented by identifying the constructs present and, if more than one construct was present, whether those constructs were nested or sequential. Only audio cues were available; the output of the programs was not shown. Also, no facility was provided

13 http://www.cms.livjm.ac.uk/caitlin/tutorial.htm provides access to these examples and the nine auralised programs used in the test.
for changing any of the system parameters (such as instrument used etc.). The entire process took around 25 to 30 minutes.

4.3.3. Scoring

Examination of the Pascal constructs reveals a set of hierarchical relationships (see Figure 4.6). For example, all loops belong to the iteration class, and the `FOR...TO` and `FOR...DOWNTO` loops are siblings and belong to the bounded iteration class. It is possible to then specify a construct at three levels: its class (iteration or selection), its sub-class (e.g. bounded vs. unbounded iterations) or its specific identity (e.g. a `FOR...TO` loop, or an `IF...ELSE` statement). Therefore, the subject responses were scored using these categories. If a subject correctly identified a construct, then this was considered to be a correct identification at the specific identity level. If they either confused a construct for its sibling or gave the name of the sub-class, then this was identification at the sub-class level. If constructs were confused for other members of their class (iteration/selection), or a response was only given in terms of the class then a class-level identification was made.

![Figure 4.6 Pascal constructs](image_url)

The Pascal constructs can be classified hierarchically. At the highest level we have two construct classes, Iteration and Selection. There are two types (sub-classes) of iteration, the bounded (count-controlled) and the unbounded loops. There are two bounded loops in Turbo Pascal, `FOR...TO` and `FOR...DOWNTO`. The unbounded loops are `REPEAT` and `WHILE`. There are two selection sub-classes, the `IF` statements and the `CASE` statements. Each of these has simple and extended forms. The extended forms include an `ELSE` path.
4.3.4. Results

The results suggested that, on the whole, subjects were able to visualise program structure using only the auralisation as a clue (see Chart 4.1). Where subjects did not correctly give the specific identity of a construct, their response was checked to see whether they had, at least, managed to identify the construct’s sub-class or class.

![Chart 4.1 Preliminary experimental results by construct](image)

**Chart 4.1 Preliminary experimental results by construct**

In this preliminary study there was one WHILE loop, eight IF/ELSE statements, two FOR loops and three REPEAT loops. The abscissa shows each of the constructs with the ordinate showing how many subjects identified the construct at each of the three levels of identification. For instance, the first construct (WHILE) was correctly identified by six subjects whilst the remaining two subjects identified it as either a REPEAT or an unbounded loop.

For instance, we observe from Chart 4.1 that the first construct (a WHILE loop) was correctly identified by six of the eight subjects; the other two thought it was a REPEAT loop. The average score at the specific identity level was 78% (min. 36%, max. 100%). Sub-class identifications accounted for a further 14% of responses on average (min. 0%, max. 43%) with class-level responses accounting for only 1%.

Of interest is construct 5 (a FOR...TO loop) which was mistaken for its sibling (FOR...DOWNTO) as many times as it was correctly identified. However, the same construct was correctly identified six times earlier on. The confusion may have arisen because this occurrence of the loop was nested within the REPEAT loop (construct 4), the added complexity of which may have affected subjects’ performance.
Chapter 4. A musical program auralisation tool

Chart 4.2 Responses by subject

This chart shows the results for the same experiment, but summarised by subject. For each subject (along the abscissa) we can read how many constructs were identified at each of the three levels.

Chart 4.2 shows how the individual subjects performed on the test. Subject 4 had a markedly lower score than the others. It is worth noting that this subject claimed to have little experience of western music. This is potentially disturbing given the prior assertion that western music is a widely understood medium. However, it should be noted that the auralisation for the \texttt{REPEAT} loop involved an extract from the tune to ‘Twinkle, twinkle, little star’. The subject commented that he was not familiar with that tune whereas the seven other subjects immediately recognised it. It is possible that this recognition provided the subjects with an advantage in recognising the structure.

Programs 8 and 9 (Figure 4.7 & Figure 4.8 below), which correspond to auralisations 11 & 12 and 13 & 14 respectively, are worth consideration. Although both programs look similar, program 8 has a nested structure whilst program 9 has a sequential structure. Subjects had great difficulty in deciding whether the second \texttt{IF} in program 8 was nested within the \texttt{ELSE} path of the other \texttt{IF}. Likewise, several subjects had difficulty deciding whether the two \texttt{IF}s in program 9 were sequential, or nested.
The only difference between these two programs is that the first has an IF...ELSE construct whereas the second has two simple IFs. A well-specified auralisation should enable this difference to be heard clearly.

The reason the confusion arose here is that an auralisation for the IF construct’s final point-of-interest (POI) was inadvertently omitted from the prototype system. Thus, it was impossible to identify in all but a very few cases the point at which an IF statement terminated.

There were two aural cues to indicate that program 8 involved a nesting of two IF statements. First, an auralisation was given to indicate the presence of an ELSE path. Secondly, CAITLIN used octave increases in pitch to denote a nesting of constructs; that is, an inner selection would be sounded an octave higher than the selection within which it was nested. Thus, it was theoretically possible to distinguish between programs 8 and 9. However, although program 9 contained a sequence of two IF statements, and so possessed neither an ELSE path signature nor an octave differentiation between the two selections, ambiguity arose because the auralisations lacked a signature for the selections’ final POI (exit). A further possible reason for the poor scoring on these two programs is that no example of nested selections was given in the familiarisation session.

This deficiency in the auralisation of selection constructs has other consequences. Program 7 (Figure 4.9) contains a REPEAT loop nested within an IF statement (constructs 9 and 10). In the prototype, octave separation was only provided for like constructs. That is, two nested selections would be played an octave apart but a loop nested within a selection (as in program 7) did not benefit from this. When combined with the missing POI signature for the selection it was impossible to state, by listening to the auralisation alone, whether program 7 contained a sequence of an IF...ELSE construct.
Chapter 4. A musical program auralisation tool

IF followed by a REPEAT or whether, as was the case, it contained a REPEAT nested within an IF.

```
PROGRAM test07;
VAR
    a      : Boolean;
    b      : Integer;
BEGIN { test07 }
    a := True;
    b := 1;
    IF a THEN
        REPEAT
            Writeln (b);
            Inc (b);
        UNTIL b > 5;
END { test07 }.
```

Figure 4.9 Test program 7

There was one further omission from the IF construct’s auralisation. If an IF..ELSE construct is auralised but the conditional expression yields a result of true then there was no indication in the auralisation that the ELSE path existed. Hence, an IF statement yielding true would sound the same as an IF..ELSE yielding true. Again, this caused uncertainty amongst the subjects’ responses. For an example of this problem see the second IF statement in Figure 4.7.

4.3.5. Conclusions

In general, programmers understood what CAITLIN was doing and could follow the execution of simple programs. However, the ambiguity surrounding the IF statement shows that it is important to auralise exit from a construct as well as entry to it. The auralisation of the first and last POIs helped subjects to differentiate nested and sequential program structures. The background drone provided during execution of WHILE and REPEAT loops also assisted with this. Thus, it may be concluded that the mapping of points of interest to musical motifs does assist users to identify constructs and to identify whether multiple constructs are nested or sequential and, therefore, that the musical auralisation approach does have merit.

Instrument selection was seen to be very important. Subjects commented that it was easy to deconstruct auralisations in the mind when the timbres used for the various constructs were markedly different.
Careful attention must be paid to signature tune construction. One subject was unable to distinguish between the entry and exit signatures of the \texttt{FOR} loop. Although this subject gave fourteen correct responses, more complex examples may well have caused confusion, especially when such loops are nested. The signature tune used for the \texttt{REPEAT} loop was more intricate than other signatures and did appear to confuse several subjects.

A proportion of the non-correct responses appear to be caused by subjects incorrectly remembering what each auralisation represented. One subject identified one auralisation as both a \texttt{REPEAT} loop and as a \texttt{WHILE} loop in consecutive test programs. He described his uncertainty as being caused by not remembering which tune was which. A longer familiarisation session may have improved his score.

The original motifs were chosen purely to aid discrimination between them. The tunes used served to provide a primitive musical environment for initial testing purposes. It was recognised that to be accessible, no reliance should be placed on the user having to be familiar with the tunes used in the motifs; that is, the system should not expect users to know ‘\textit{twinkle, twinkle little star}’ or any other such tune to be able to use the system.

Having seen that the system could be used to generate auralisations that could be interpreted by programmers the next step is to apply the features of the construct taxonomy to the design of a proper set of auralisation motifs.

\section*{4.4. Hierarchical motif design}

Our perception of music is primarily temporal, that is, we more readily perceive those features between which there are temporal relationships \cite{159}. It has been observed that we tend to perceive music not as an arbitrary sequence of note durations but as a temporal structure in which notes are grouped into various kinds of units \cite{97}. Two features of temporal musical structure, \textit{succession} and \textit{overlap} \cite{159} have analogues in the program domain: sequence and construct nesting. Also, as the structure of music (like that of programs) is multi-levelled \cite{159} and given that the events of an executing program occur within a time-ordered framework, it would seem sensible to attempt to map program events to musical ones. For instance, the \textit{exposition}, \textit{development} and \textit{recapitulation} of the sonata form could map to the ini-
tialisation, executable-body and finalisation sections of a typical program and its subprograms.

Programming languages offer the programmer a range of tools for achieving similar ends. Pascal provides three iteration constructs — `WHILE`, `REPEAT` and `FOR`. Each allows iterative execution of code but differs from the others in the way the looping is controlled. So, the three loop constructs are different but share certain characteristics. Similarity is also found in the selection statements. `IF`, `IF..ELSE`, `CASE` and `CASE..ELSE` provide for selective execution of statements but use different mechanisms to accomplish it.

So, there is a taxonomy of constructs implicit in the language; it is possible to model this hierarchy by a musical framework [6]. In so doing there would be a theme denoting iteration and another theme to represent selection. Within each, variations of the theme would be used to represent the individual constructs. All selections would thus sound similar to each other but entirely different from the loop constructs. Figure 4.10 shows how the selection and iteration constructs of Pascal can be viewed and how this structure might be modelled in a hierarchical motif design.

![Figure 4.10 Taxonomy for motif design](image)

Onto the construct classification we can overlay a motif hierarchy. By defining a motif for the two construct classes and then defining variations of the motifs for the lower nodes on the tree we can create a musical equivalent of the construct hierarchy.
Therefore, a construct may be specified at any of three levels of identification:

- **Class**—a vague description where we think of constructs in terms of their generic type (iteration or selection).

- **Sub-class**—identification of constructs as groups of siblings. That is, we talk about the `IF` statements, the `CASE` statements, the bounded loops (`FOR` statements) and the unbounded loops (`WHILE` and `REPEAT`).

- **Specific identity**—an exact description, corresponding to the bottom level of the tree in Figure 4.10. For example, here we view an `IF` statement as different from its sibling, the `IF...ELSE` statement.

A set of auralisation motifs designed to reflect this structural hierarchy of constructs should, therefore, allow constructs to be identified at their specific identity level as well enabling users to make higher-level abstractions by identifying them at their sub-class and class levels.

Having achieved a measure of success with the simplistic musical devices used in the preliminary experiment, the next step was to focus on the details of motif design. Attention was focused on formulating some common design principles for the motifs. Hitherto, the motifs relied on fairly distinctive but arbitrary musical sequences (e.g. *Twinkle, twinkle little star*) with no underpinning design methodology. The next stage was to attempt to construct new motifs according to more formal rules. These rules would be derived by examining the similarities and differences between the individual Pascal constructs and mapping these onto a musical framework.

The preliminary experiment discussed in section 4.3 and elsewhere [152] was used solely as a rough guide to whether experienced programmers could (in principle) grasp the practice of communicating program structure by music.

### 4.5. Design of auralisation motifs

When moving from the ad-hoc prototype motifs to a set of more formally structured motifs we initially distinguished between the selection and iteration classes by using a chord-based motif for the iterations and a melodic device for the selections. In this way it was anticipated that the two construct classes would be quite different
in their respective auralisations, yet the principle of construct similarity (in which all selections are related and all iterations are related) is maintained.

We stated in section 4.2.3 that for the two classes of construct there are four fundamental points of interest that require auralisation, viz.:

1. entry to the construct;
2. evaluation of controlling conditions;
3. execution of the construct’s body;
4. exit from the construct.

The first and last POIs bracket the construct and are thus clearly related. Therefore, these two POIs are modelled by related motifs where the exit motif provides closure for the entry theme.

![Figure 4.11 Generic selection theme (Trk 02)](image)

**Figure 4.11 Generic selection theme (Trk 02)**

This is the foundation on which the individual selection construct motifs will be built. One would never hear this high-level motif, but its features will be inherited and transformed by each of the lower level motifs.

The generic selection theme is presented as Figure 4.11. Metaphorically, a selection poses and then answers a question, and this was represented by a rising scale signifying entry to the selection construct (which when played sounds like a question being asked—just as we upwardly inflect our voices when asking questions) and a descending scale denoting exit (showing that the question has been answered). The underlying theme was reworked for each of the individual selection constructs by changing the rhythmic and harmonic structures.

The generic iteration motif is given as Figure 4.12, which shows a simple chord progression of I-I\(^6\)\(^3\) to denote entry to and exit from the loop construct.

---

\(^{14}\) I.e., tonic to first inversion. A first inversion (a chord with the mediant in the bass) is often called a chord of the \( \frac{4}{3} \) because the two notes above the bass are respectively, a third and a sixth from the bass (e.g. EGC in C major).
Again, this idea was reworked for each of the loop constructs by employing various chord sequences for entry and exit all of which began and ended on the tonic.

4.5.1. Construct bodies

The principle of construct similarity was again used in the design of motifs for the construct body points of interest. Selections involve evaluation of Boolean expressions and conditional execution of statements. Iterations use the value of a Boolean expression to control the repeated execution of a statement block. Boolean evaluations were modelled by using motifs in major keys for True results and minor keys for False results. The justification for this choice is that the ordinal set of Boolean values (False and True) can be mapped quite conveniently onto the major and minor modes.

A potential danger lies in our tendency to associate music in major keys with happiness and the minor modes with sadness; we may subconsciously equate a Boolean True (major) as being ‘good’ (or even a success) whilst seeing the False as bad (minor). But, for the continued execution of the REPEAT loop, the Boolean expression must be False and so Boolean False/minor key does not mean that a test is unsuccessful in the terms we might think of it. However, if the major mode is used as the assumed default then this accords with the tendency of adult western listeners to default to a major mode in the absence of information to the contrary [158]. The fact that the REPEAT loop requires a Boolean False (or diatonic minor) to continue its iteration merely serves to highlight the difference in logic between it and the WHILE construct. The mapping of True to major and False to minor was incorporated in the various points of interest of the constructs.

A loop construct implies persistence over time (usually several subordinate actions will be carried out within the body of the loop) and so a background drone was added to the iterations’ bodies to reinforce in the listener’s mind that everything that is happening is doing so within the loop. We need to know when the individual
loop iterations occur and when the controlling Boolean expression is evaluated. For the **REPEAT** and **WHILE** loops a simple major/minor chord device is used when the loop condition is tested; this would be heard immediately after entry to the **WHILE** loop, but after the iterated statements of the **REPEAT** loop. Each time one of these chord devices is heard we know that the loop has reached its decision point. The null **WHILE** loop (where the terminating condition is true upon entry) would thus be heard as a sequence of entry motif followed by minor chord motif for condition evaluation followed by the exit motif.

The **FOR** loops are bounded (count-controlled), a loop invariant taking incremental steps from a starting value to an end value. To denote this stepping up (or down) of the invariant, the pitch of the drone in the **FOR** loop was increased (or decreased for the **FOR...DOWNTO**) by one diatonic step with each repetition\(^\text{15}\).

### 4.5.2. Variations on a theme

In musical terms, this design principle would be thought of as variations on an original theme. This ensures that all selections sound like each other but can be distinguished by their individual mutations of the class motif. The reason for doing this is not simply one of organisational convenience, although such categorisation can be useful cognitively; rather, it provides us with a means of program comprehension at different levels of abstraction and also with a way of conveying spatial information temporally. The abstraction is achieved as one can choose to listen to a program’s auralisation in terms of its overall structure (e.g. a selection followed by a loop etc.) or in terms of its details (e.g., an **IF...ELSE** followed by a **WHILE**). Further abstractions could be achieved by providing selective auralisations in terms of:

- classes of construct to auralise;
- number of iterations of a loop;
- nesting depth of constructs;
- aural presentation of temporal and spatial information.

\(^{15}\) The exact interval of each step depends on the musical scale assigned to the construct by the user. It is possible to have the **FOR** loops mapped to any scale and not just the seven-step major scale (e.g. ten note blues, although the blues scales are not strictly diatonic as they rely on accidentals).
4.5.3. Temporal and spatial mappings

The categorisation of constructs also allows for a musical portrayal of spatial program features. One of the motivating factors behind research into program auralisation is that sound is a temporal medium and program execution is a temporal phenomenon and, therefore, it makes sense to explore the possibilities of mapping the latter to the former. Through auralisation we can listen to the execution of a program and make inferences about its state. But restricting an auralisation to temporal detail alone may lead to a loss of quality.

4.5.3.1. IF statements and ELSE paths

Consider the code fragment in Figure 4.13. If all occurrences of IF statements sounded alike and if the value of variable a were greater than 3 then it would be impossible to determine, from the auralisation alone, whether one is hearing a simple selection (first IF in Figure 4.13) or one with an ELSE path but where the ELSE part was not followed (second IF). By fully classifying the constructs and building this into the theme tunes such ambiguity can, in theory, be avoided.

```
(* IF without ELSE *)
IF a > 3 THEN
  Writeln ('a > 3') ;

(* same test, but with ELSE *)
IF a > 3 THEN
  Writeln ('a > 3')
ELSE
  Writeln ('a <= 3') ;
```

Figure 4.13 Anticipation of structure

The second selection in this program contains an ELSE path. But if the condition \( A > 3 \) is True then a trace of the program would reveal two identical traces for the two selections. However, by signalling the presence of an ELSE path in the auralisation, even if it is not followed, one is aware that the branch is there.

An advantage of this approach is that when an IF.ELSE occurs and the initial condition is false the listener would not be caught unawares by the subsequent occurrence of an ELSE structure in the auralisation. By setting up this anticipation\(^\text{16}\) of possible future events on the part of the listener, CAITLIN creates a sort of construct footprint which shows not just where the program has been but also where it might go. Such an auralisation would be able to capture the spatial information re-

---

\(^{16}\) See Robert Jourdain’s ‘Music, the Brain and Ecstasy’ for an interesting description of musical anticipation [87].
lating to the presence of an ELSE path; the difference between the two constructs in Figure 4.13 is readily apparent in such an auralisation.

4.5.3.2. CASE statements

The CASE statement (Figure 4.14) is another example of how an auralisation in CAITLIN conveys spatial information within a musical framework.

```
CASE x OF
  1..4 : Writeln ('Between 1 and 4') ;
  5..7 : Writeln ('Between 5 and 7') ;
  ELSE   Writeln ('No match') ;
END ;
```

Figure 4.14 CASE statement

The CASE constants are laid out spatially in a program listing. Therefore, we think in terms of matches of the CASE selector with the first, second, or third CASE constants. By creating an auralisation that sounds each of the constants in turn and whether or not that constant produces a match, we are able to preserve this spatial understanding in the auralisation.

Like IF..ELSE, CASE allows for alternative courses of action depending on an expression's value. Of interest to the programmer is which instance of the CASE constants (if any) produces a match with the CASE selector (x in Figure 4.14). Unlike the IF..ELSE which carries out its comparisons of the various (nested) conditional expressions sequentially, no such ordering is implied by the CASE. However, it is convenient for us to think of the selector as matching in turn the first, second, third etc. instance of the CASE constant lists. In Figure 4.14, if x had the value 6, then we would say that the second CASE constant list produced a match.

This can be regarded as a spatial judgement because the second constant is lower down the list than the first. This can be communicated by signalling the presence of each constant in turn (by a percussive sound, in the CAITLIN system). If a particular constant produces a match then this too can be mapped to a musical event (e.g. a major chord). The resultant auralisation would give the effect of the program stepping over each constant list in turn until the end (or ELSE part) is reached or a match occurs.

4.5.3.3. Nested constructs

Another spatial element that can be mapped to sound is construct nesting. Programmers show this visually by indenting the code for each level of nesting. It was decided to model this in CAITLIN by increasing the pitch of nested construct motifs by one octave. However, it only takes five or six levels of nesting before the pitch be-
comes so high that differences between notes in the melodies become very hard to
discern. Other possible mappings include position within the stereo field, reducing
the size of the pitch increase (though this causes problems with harmonic discords)
or using background drones for each construct, the pitch of which can be varied with
less difficulty.

4.6. New motifs

A new set of motifs based upon the notion variation of class themes and the
other ideas discussed above was constructed. These are discussed below.

4.6.1. Selection motifs

The redesigned motifs for the simple $\text{IF}$ statement (i.e., an $\text{IF}$ without an $\text{ELSE}$
path) are given as Figure 4.15 and Figure 4.16. We observe the same basic melodic
theme as in the generic motif but with a modified rhythm. Also, notice the device
showing evaluation of the Boolean expression. Where the expression yields a $\text{True}$
result (Figure 4.15) a musical device in the $\text{major}$ key is used. Where the expression
yields $\text{False}$ a $\text{minor}$ key is employed (Figure 4.16). Triplet rhythms help to keep the
motifs short by putting three notes into the space of two. Triplets also give a pleas-
ant-sounding feel to the music.

![Motif Diagram]

**Figure 4.15 Simple $\text{IF}$ statement yielding $\text{True}$ (Trk 04)**

This auralisation is of an $\text{IF}$ statement whose condition yields a $\text{True}$ result. Notice how the basic structure
of the generic selection motif is preserved; transformations in terms of contour and rhythm make the mo-
tif unique. Track 4 on the accompanying CD contains this auralisation.

The music in Figure 4.15 would be generated by the following Pascal code:

```pascal
a := 10;
IF a = 10 THEN
  Writeln ('Condition is True');
```
Figure 4.16 Simple IF yielding False (Trk 05)
In this example the simple IF now results in a Boolean False. Notice how the conditional evaluation is now represented by a minor key.

An example of Pascal code that would result in the auralisation of Figure 4.16 is:

```pascal
a := 20;
IF a = 10 THEN
  Writeln('Condition is True');
```

The auralisations for IF...ELSE statements yielding True and False are now given as Figure 4.17 and Figure 4.18 respectively. Notice the semi-quavers in the opening and closing that signal the presence of the ELSE path.

Figure 4.17 IF...ELSE yielding True (Trk 06)
The IF...ELSE motif is very similar to that of the simple IF. The differences lie in the rhythmic structures of the first and last points-of-interest (bars 1 and 3).

Figure 4.18 IF...ELSE yielding False (Trk 07)
Again, the conditional evaluation is represented by a minor key as the condition is now False.

Like the IFs the CASE statement is also divided into two sub-classes: the simple CASE and that which contains an ELSE part\(^{17}\). The CASE operates by searching lists of CASE constants for a match with the CASE selector. This was auralised by simulating

---

\(^{17}\) It should be noted that the CASE..ELSE construct is peculiar to Turbo Pascal. Standard Pascal does not provide for an ELSE branch in the CASE statement.
the \texttt{CASE} statement stepping through each constant in turn until either the list is exhausted or a match is found. Comparison of the selector with each constant was signalled by sounding a cow-bell timbre as each constant was encountered. When the selector matches a value in the constant list then a major chord is also sounded alongside the cow-bell. Exit from the \texttt{CASE} statement is signified by playing the exit tune in a major key if a match was found or in a minor key if no match was made.

For example, the following code fragment would result in the auralisation shown in Figure 4.19.

\begin{verbatim}
a := 10;
CASE a OF
  2   : Writeln ('Match first instance') ;
  5   : Writeln ('Match second instance') ;
  10   : Writeln ('Match third instance') ;
END ;
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.19.png}
\caption{CASE with a match (Trk 08)}
The \texttt{CASE} is differentiated from the \texttt{IFs} by rhythmic changes and a modified melodic contour. Also, the stepping through of the \texttt{CASE} constants is signalled by a hit on a cowbell. A match causes a major chord to be played too.

If the value of \texttt{a} in the previous example were 11 rather than 10 then there would be no match with any of the \texttt{CASE} constants. The resulting auralisation is given as Figure 4.20.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.20.png}
\caption{CASE with no match (Trk 09)}
This \texttt{CASE} construct generated no match with any of the constants and so no major chord is heard. Also, the motif ends in a minor key to reinforce this lack of a match.
The CASE..ELSE always results in a statement being executed. If there is no match with a CASE constant then the statement belonging to the ELSE part is followed. Consequently, the CASE..ELSE always exits in a major key. The cow-bell was used again to signal the testing of the CASE selector against each constant list in turn, and a match was indicated as before. If no match is found then a minor-key is used to denote execution of the ELSE part. For example, the auralisation given as Figure 4.21, would be generated by the following code fragment which shows a CASE..ELSE with a match of the selector against the third constant:

```pascal
a := 10;
CASE a OF
  2 : Writeln ('Match first instance');
  5 : Writeln ('Match second instance');
 10 : Writeln ('Match third instance');
 20 : Writeln ('Match fourth instance');
ELSE : Writeln ('No match found');
END;
```

Figure 4.21 CASE...ELSE with a match (Trk 10)
The CASE..ELSE is differentiated from its sibling by more complex rhythm and melodic contour. The cow-bell and major chord for a match is the same as for the simple CASE.

Again, if the variable a has the value 11 then no match is made and CAITLIN would generate the auralisation given as Figure 4.22.

---

18 With hindsight this is not, perhaps, an intuitive mapping. Major/minor devices are used to signal the results of Boolean evaluations. Because the CASE..ELSE still tests for a match with the CASE constants, then its exit tune should reflect the result of this test rather than its present function which is to signify the execution of a statement within the construct.
4.6.2. Iteration motifs

The iteration motifs were redesigned around the class theme of a tonic-to-tonic chord progression. An example of the WHILE loop's motif is given as Figure 4.23. This score shows a WHILE loop of two iterations.

Figure 4.23 WHILE loop with 2 iterations (Trk 12)
The WHILE loop inherits the basic form of the generic iteration construct. Additional chords are used to represent evaluation of the controlling condition. When the condition becomes False then the motif moves into C minor.

Entry to the loop is denoted by the repeated chord in the first bar. The second bar represents the evaluation of the Boolean expression controlling entry to the loop body. In this example we see that this expression yields a result of True twice (signified by the F-major chords). The third time the expression is evaluated it yields False (shown by F-minor chords) which results in the construct terminating. The major chord at the end is slightly misleading. It is meant to signify successful exit from the construct, but it could be argued that it should be in a minor mode to reinforce the idea that the WHILE’s condition has given a False value and so the construct has ended. The note C that is sustained across the first four bars is a drone that is used to indicate to the user that the program is executing the body of a loop construct. As the body of the loop could contain many further constructs, the drone was intended to assist the programmer in recognising that the loop has not yet terminated.
Like the **WHILE** loop, this motif maintains the features of the generic iteration but modifies them for its own purposes. Because the loop iterates when the condition yields *False*, the minor key device will be heard prior to the execution of each iteration. When the condition becomes *True* a major key will be heard and the motif will end.

Figure 4.24 shows a **REPEAT** loop with two iterations (that is, two executions of the iterated statement). The chords in bars 1 and 2 denote entry to the construct and its body. In the third bar there is an F-minor to C-minor cadence telling the listener that the condition at the bottom of the loop has yielded *False* and so the loop body is executed once more. The second time that the Boolean expression is evaluated (bar 4) it yields *True* (that is, the **UNTIL** condition is satisfied) and so a major cadence is played and the loop exits. Again, a drone was used\(^{19}\).

The **FOR**...**TO** loop’s variations are given as Figure 4.25 and Figure 4.26. We see the overall progression of the generic signature tune (Figure 4.12) retained but with some additional chords and rhythmical variations. Figure 4.25 shows a **FOR**...**TO** loop with ten iterations; a **FOR**...**DOWNTO** loop of ten iterations is given as Figure 4.26. The opening phrase of the **FOR** statement is dependent on the loop being either incremental (Figure 4.25 which has a continually rising note sequence in bar 1) or decremental (Figure 4.26 which reverses the order of the third and fourth chords resulting in a drop in pitch of the top note). As the **FOR** loops are bounded (count-controlled), the continuous drone was modulated in pitch with each iteration. The drone goes up in pitch each time around the **FOR**...**TO** and down in pitch in the **FOR**...**DOWNTO**.

\(^{19}\) Although the **REPEAT** uses the same pitch for its drone as the **WHILE**, the two constructs are played using different timbres.
Chapter 4. A musical program auralisation tool

The FOR...TO statement uses an upwards scale to represent each iteration of the loop. On the final iteration the drone has a rhythmic variation.

In all cases it is important to try to keep the motifs as short as possible while maintaining the ability to generate expectancies [56] of resolution in the mind of the listener.

4.7. Pilot Study 1

Using the redesigned hierarchical motif structures, a short experiment was carried out to see whether programmers could understand the approach. There were six subjects in this study all of whom were lecturers in computing at Liverpool John Moores University involved in the teaching of programming at the undergraduate level. Three of the subjects (subjects 1, 2 and 3 also participated in the preliminary experiment described in section 4.3). Subject 1 was the person who did not recognise *twinkle, twinkle little star* in the earlier experiment.

The experiment comprised four tests (labelled 1a to 1d) performed sequentially over the space of about an hour. Several short Pascal programs were auralised and the results recorded onto audio tape. All subjects heard the auralisations at the same time via a single stereo tape deck. The auralisations were selected so as to give an example of each possible permutation of POIs for each construct. Tests 1a and 1b involved ten selection auralisations and eleven iteration auralisations; tests 1c used twelve selections and nine iterations and test 1d employed eleven selections and
thirteen iterations. Tests 1a to 1c employed twenty-one construct auralisations overall and test 1d used twenty-four. The format of each test was:

- First, present exemplar auralisations (chosen randomly from the available set);
- For each example in a randomly chosen subset of the available auralisations: play the auralisation, ask subjects to identify it and then give subjects the correct answer;
- Finally, present the subjects with all the auralisation examples (randomly ordered) and ask them to identify each one in turn.

### 4.7.1. Tests 1a to 1d

In test 1a, subjects were presented with twenty-one auralisations and were required to categorise each one as either ‘selection’ or ‘iteration’. The purpose of this test was simply to see whether the two motif classes were suitably distinct, or whether there were any areas of concern.

Test 1b involved the same auralisations as test 1a, but this time subjects were required to specify the name of the construct represented by each auralisation (e.g., `FOR...TO`, `IF...ELSE`, `FOR...DOWNTO` etc.). In both tests 1a and 1b, each auralisation represented only a single construct.

Test 1c required subjects to identify individual constructs again, but the differences from test 1b were:

1. the set of auralisations was different;
2. each auralisation contained either two or three sequential constructs (i.e., no construct nesting was employed).

The final test, 1d, used its own set of auralisations and this time each auralisation contained a number of constructs which were combined in a mixture of sequential and nested structures (seven sequences and seven nestings in total).

### 4.7.2. Scoring

In test 1a, a response was judged to be correct if the subject correctly identified the construct class. In tests 1b to 1d, as in the preliminary experiment, scores were given for constructs that were correctly identified at the specific identity, sub-class
and class levels. Additionally, a score was kept of the subjects’ identification of nested and sequential structures in test 1d.

4.7.3. Results

Chart 4.3 shows the average correct response (specific identity for tests 1b through 1d) for construct identification in tests 1a to 1d. Unsurprisingly, there is a sharp drop off in accuracy between 1a and 1b as subjects had to switch from identification at the class level to identification at the individual construct level. It is encouraging to see that identification improved over the course of the experiment.

![Chart 4.3 Pilot 1—average correct responses](image)

For each of the four tests (abscissa) the ordinate shows the average correct identification of constructs. Note, that for test 1a, subjects were only required to identify the class (iteration or selection) of the constructs. For tests 1b to 1d, subjects were asked to identify the individual constructs.

Chart 4.4 shows how construct classes were misidentified during the experiment. On average, in test 1a 30% of the iterations were misidentified as selections. The best score here was a misidentification rate of 0%; the lowest was the subject who identified 55% of the iterations as selections. We also observed that 18% of selections were misidentified as iterations.
At first sight it may just seem that when a subject was unsure of an auralisation’s type, then the tendency was to guess it to be a selection by default. However, by the end of test 1d we observe that class-error rate has dropped from 30% to 10% for iterations, and from 18% to 7% for selections which may suggest that although iterations are hard to discriminate at first, with practice accuracy improved.

Chart 4.5 shows the relative levels of correct identification for each test. Test 1a was straightforward because subjects were required only to distinguish between two auralisation classes (iteration and selection) and so answers were either wholly correct or incorrect. However, tests 1b to 1d required subjects to identify each auralisation as a specific construct. Constructs possess three levels of identity: their specific identity (e.g. a `FOR...TO` loop), a sub-class identity (e.g., `FOR...TO` and `FOR...DOWNTO` belong to the sub-class `FOR` loop) and finally their class identity (e.g., `FOR`, `REPEAT` and `WHILE` all belong to the iteration class). Therefore, it was possible for subjects to incorrectly identify an auralisation’s specific identity whilst still placing the construct in the correct class or sub-class. The results from tests 1a to 1d are given as Chart 4.5.
Chart 4.5 Pilot 1—Identification of motif class

On this chart we can see how subjects managed to identify motifs overall. For each test, the average identification at the three levels of identity is shown. For instance, for test 1b we observe that, on average, 38% of constructs were correctly identified, a further 28% were identified at their sub-class level, with 16% being at least correctly identified as iterations or selections. Note that for test 1a, only class-level identifications were required.

We notice that at the finest level of accuracy (specific identity) the number of correct responses was initially low (38% for test 1b, min. 24%, max. 48%) but that it rose to 66% in test 1d (min. 54%, max. 79%). It should be noted at this point that test 1d involved a level of detail not present in the preceding tests—constructs were nested in half the cases. Interesting is that although some subjects had difficulty discriminating between nested and sequential structures the added complexity did not affect the upward trend in correct identification of constructs (average number of nestings misidentified as sequences was 21%, (subjects ranging in accuracy from 57% to 100%); the average number of sequences misidentified as nestings was 11%, ranging in accuracy from 71% to 100%).

Chart 4.5 also shows that of those constructs that had their specific-identity misinterpreted, subjects still managed, on average, to correctly identify the sub-class of those constructs in 28%, 24% and 24% of cases in tests 1b to 1d respectively. Furthermore, when taken together with the number of constructs that were incorrectly identified at the specific-identity and sub-class levels but that were placed into the correct class (16%, 2% and 2% in 1b to 1d respectively) we observe two things:

- Correct identification at the class level rose across tests 1a to 1d.
• Accuracy at the specific-identity level rose across tests 1b to 1b at the same time as misidentifications of class and sub-class fell.

4.7.4. Discussion

Subjects appeared to find it harder to identify iterations as iterations than selections as selections, though on the whole, identification of constructs at the class level was quite successful. (This is reversed in the second pilot study where selections were misidentified more often than iterations—see section 4.8 ff.)

Initial identification of constructs was poor but this improved over the course of the tests. This suggests that a learning effect may be at work and that with practice the auralisations are durable and learnable.

Different subjects used different cues to identify auralisations. Some used the contour of the music to assist them (see Figure 4.27 which shows one subject’s rough work). Others relied on the timbre of the auralisations but some of these had difficulty in discriminating between the string sound used for WHILE loops and the synthesised brass sound used for REPEAT loops.

Although the subjects were not required to state the results of any Boolean evaluations heard during the experiment, comments made by them afterwards suggested that they did recognise such occurrences. For instance, subjects referred to the happy and sad tunes they heard when conditions were being tested within the auralised programs.
Figure 4.27 Melodic contour mapping
This subject drew contours of the motifs to assist in identification. The subject had no formal musical training and did not play any musical instruments.

4.8. Pilot Study 2

The second pilot study was procedurally identical to the first. Changes were made to the set of programs used in the test and to the motifs used to auralise the programs. The program set was amended to give a better balance of construct types across the tests. The motifs were changed for reasons discussed below in section 4.8.1.

Eight subjects took part in this test. The subjects were a mixture of full-time lecturing staff, post-graduate research students and research assistants from Loughborough University’s Department of Computer Science. The subjects each had between five and fifteen years experience of programming.
The purpose of this study was simply to gather more data with which to inform the process of CAITLIN’s redesign. However, because some deficiencies were spotted in the motifs used in the first pilot study, this second study was also used to see if a different approach to motif design would yield any better results.

4.8.1. Changes to the motifs

As stated in section 4.4 the intention of the motif design was that the hierarchical structure of the Pascal construct types should be captured and reflected in the auralisations. An early design decision was to distinguish between selection and iteration constructs by using a motif based upon a simple melody for the former and one based on a simple chord progression for the latter. After using the hierarchically-designed motifs it became clear that this separation between melody and chord progression did not work well in practice. For example, the motif for an IF..ELSE (see Figure 4.17) contains a chord device to signal the Boolean evaluation, while the FOR loops (Figure 4.25 and Figure 4.26), end up possessing a distinctive melody which emerges from the chord progression employed. Upon further reflection it seemed inadvisable to rely on recognition of chords and their inversions as the primary means of identification for the average, non-musically trained user. Also, using chords as a building block adds to the number of concurrent musical events and so the harmonic environment becomes richer and the probability of misleading harmonic interactions increases.

![Figure 4.28 Melodic contour for selection class](image)

The melodic contour for the selection class is intended to mimic the rising and falling intonation of the human voice when asking and answering a question.

The motifs for REPEAT, WHILE and FOR loops were rewritten so as to follow the same design principle as used for the selections, that is, variations on a simple theme. The selection theme was described as mimicking a natural language ques-
tion-answer dialogue. This gives rise to a melodic structure which can be represented pictorially as Figure 4.28.

![Figure 4.28 Melodic structure](image)

**Figure 4.29 Melodic contour for iteration class**
The contour for the iteration class is supposed to represent repeated action.

The melodic contour for the iteration class is given as Figure 4.29. The notion of statements being executed repeatedly was loosely modelled by choosing a theme which has a repeating contour (in this case looking like two cycles of a cosine function). Close examination of the iteration motifs (see below) reveals this contour in each case (although it is inverted in the case of the `FOR...DOWNTO` loop). The redesigned `WHILE` motif is presented as Figure 4.30.

![Figure 4.30 Melodic motif for WHILE loop](image)

**Figure 4.30 Melodic motif for WHILE loop (Trk 16)**
As before, the major/minor keys were used for conditional evaluations. Notice how the melody of the construct auralisation follows the basic pattern shown in Figure 4.29. Percussive cues have been added to signal evaluation of the controlling condition. The motif begins in a major key and ends in the minor key as the controlling condition is `False` upon exit.

Observe the use of rhythmic sounds at the point of evaluating Boolean expressions. This mimics the `CASE` motif that uses a cowbell when evaluating the `CASE` constants. In the case of the loop constructs, the sound of drumsticks being hit together signals a decision point in the loop. When the loop condition yields a result that means the loop must terminate (`False` for `WHILE`, `True` for `REPEAT` and the when the
loop invariant equals its final value in the FOR constructs) then the sound is changed to a sleigh-bell.

Another point to note in the new WHILE motif is that entry is played in a major key but exit is played in a minor key. This reinforces the characteristic of the WHILE loop which is that it finishes when its condition is False. Similarly, the REPEAT loop (Figure 4.31) now begins in a minor mode and ends in the major.

![Figure 4.31 Melodic REPEAT motif (Trk 17)](image)
The contour of the generic iteration is maintained in the new REPEAT motif. The auralisation begins in a minor key and ends in the major as the controlling condition is True upon exit.

The new FOR...TO and FOR...DOWNTO loops are given as Figure 4.32 and Figure 4.33 respectively. Note, no drumstick sound was used when looping in the FOR constructs as the pitch modulation of the drone serves the same purpose.

![Figure 4.32 Melodic FOR...TO motif (Trk 18)](image)
Observe in the two FOR loops the inversion of melodic contour. The FOR...TO begins with an up-down melody and ends with a down-up whilst the FOR...DOWNTO reverses this pattern. The reason for this is simple: one loop counts up whilst the other counts down and so this was modelled in the entry and exit motifs. As the FOR...TO is the incremental loop, the opening motif was given the up-down shape so as to lead more smoothly from the last note of the entry motif to the first note of the drone.
Likewise, the `FOR...DOWNTO`s drone descends and so the down-up device was employed to give a smoother transition from entry motif to drone\(^{20}\).

![Figure 4.33 Melodic FOR...DOWNTO motif (Trk 19)](image)

### 4.8.2. Results

Chart 4.6 gives the average correct response for construct identification in tests 1a to 1d. Unlike the previous experiment (see Chart 4.5) there was actually a decline in accuracy across the four tests. Consequently, the misidentification of construct classes increased (see Chart 4.7), this time with selections having a higher misidentification rate than iterations.

![Chart 4.6 Pilot 2—Average correct responses](image)

**Chart 4.6 Pilot 2—Average correct responses**

This chart shows the average correct identification of individual constructs across the four tests. As in the previous experiment, only class-level identification was required for test 1a.

---

\(^{20}\) Comments from subjects after the test indicated that the smooth transition was less important than the relative intuitiveness of the mapping. Confusion arose when trying to distinguish between the two `FOR` loops as the opening sequence of the `FOR...TO` descended in pitch whilst it ascended for the `FOR...DOWNTO`. 
Chart 4.7 Pilot 2—Misidentification of class
In contrast with the earlier experiment, this chart shows that selections were misidentified more than iterations.

Chart 4.8 shows the aggregate responses across the four tests in the experiment. The overall decline in accuracy is clearly seen here and it appears to be due entirely to the misidentification of constructs at their generic class level; that is, if a specific construct was misidentified, there was no increased compensatory correct identification of its sub-class or class across the four tests. This may suggest that the differences between the iteration and selection motifs actually became less clear over time.

Chart 4.8 Pilot 2—Identification of motif class
This chart shows the combined correct responses at the three levels of identification across the four tests. Test 1a required only class-level identification.
4.8.3. Discussion

The results of this experiment were disappointing given that the iteration motifs were reworked with the express purpose of making them easier to identify. However, it should be noted at this point that the misidentification of iterations was lower than that of selections which is in contrast to the previous experiment.

Subjects commented that the shapes of the FOR loop motifs were counter-intuitive as the incremental FOR started with a descending pattern whilst the decremental loop began with an ascending pattern. As in the first pilot study, subjects also commented that the two timbres used for REPEAT and WHILE loops were difficult to tell apart.

4.9. Conclusions

The three quantitative experiments yielded some useful information about program auralisation. First, the earlier tests indicated that auralisation can be used in communicating information to programmers. Comments received after the experiments showed that subjects were generally receptive to the idea of program auralisation and believed that it could be made to work. Useful information regarding selection of timbre, complexity of melodic structures and intuitiveness of mappings was gleaned from these tests. It should be noted at this point that all three experiments required identification to be made solely on the basis of aural cues. In a real debugging situation programmers would obviously have detailed knowledge of the program being auralised.

Between the three experiments we observe an overall decline in the ability of subjects to correctly identify Pascal constructs from the auralisations alone. The first two experiments showed an increase in correct responses across the duration of the tests; the third experiment showed a decrease in response. We also observe that whilst the second experiment showed an improvement in construct identification across its four tests, the overall accuracy for the study was lower than for the preliminary experiment. We must not discount the effect of programming experience on these results. In the preliminary study, all subjects were university lecturers with substantial programming experience. In the first pilot, again all subjects were lecturers, but three of them were very new in post and had less experience of pro-
gramming and of teaching programming than the others. The second pilot had subjects who were postgraduate students, again with relatively little experience in programming compared with the expert subjects.

A caveat must also be observed to do with sample sizes in these three studies. The largest study had only eight subjects and so not too many definitive conclusions should be drawn from the results.

In the preliminary experiment we observed a generally high level of construct identification with the majority of constructs either being specifically identified or being placed in the correct sub-class. The first pilot study shows an increase in the identification at the class level with the consequent drop in accuracy at the sub-class and construct-identity levels. The second pilot study shows a further drop still.

In summary, we can observe the following trends: Accuracy of construct identification decreased and motif design changed from purely arbitrary mappings in the preliminary experiment to more formalised musical structures in the third.

In other words, as the motifs became more musical, more structurally consistent and more organised, accuracy of identification became worse. From this we draw the tentative conclusion that whilst the organisational principles of motif design provide a useful framework for the representation of constructs, the average user does not discriminate well between subtle musical changes.

It became clear that what the average user needs above all are clear and unambiguous auralisations. Motifs should be constructed in such a way that their organising principles and musical grammar and structure become transparent and so allow the individual features of the different constructs to be made readily apparent.

4.10. Summary

In this chapter we have specified and described the construction of a system for the generation of musical program auralisations. Reasons for adopting a structured musical framework based upon the component parts of Pascal constructs (point of interest) and the structural hierarchy of those constructs were given. A small study was then conducted to test the underlying concepts and the results were encouraging.