Development of Intelligent Music using a Hierarchical Composition System (DIMHCS)

Jakob Schmid

30th April 2005

Faculty of Engineering and Science Aalborg University

Department of Computer Science

Title:

Development of Intelligent Music using a Hierarchical Composition System (DIMHCS)

Project Period:
DAT2,
2/2 to 27/5 - 2004

Project Theme:
Compilers and Languages

Project Group:
Jakob Schmid

Group Members:
Jakob Schmid

Supervisor:
Linas Bukauskas

Number of copies: 4

Number of pages: 68

Abstract:

This report documents the development of a music composition language called the "Development of Intelligent Music using a Hierarchical Composition System" (DIMHCS) language. Furthermore, it documents the implementation of an interpreter for DIMHCS programs. The interpreter reads a file containing DIMHCS language source code and outputs a WAV file containing the finished music composition.

Jakob Schmid

Contents

Introduction

The system described herein could be used by composers of electronic music who wants to go beyond the limits of existing music software. The system is perfect for loop-oriented music.

Development of this system was partly motivated by the lack of an existing tool with suitable flexibility and ease of use.

Terminology

A few musical and music technological terms will be used in the report. Some of these are considered general knowledge and are not explicitly defined. To avoid confusion, a few of the less known terms are defined here:

pitch the frequency of a sound

volume the loudness of a sound

note a named set of frequencies. For instance, the note A consists of the frequencies with frequency $440 \cdot 2^n, n \in \mathbb{Z}$.

chord a combination of 3 or more pitches at once.

well-tempered tuning a historically and culturally (western) defined way of adjusting the frequencies that can be produced by an instrument

part a time-limited subset of a musical composition. To exemplify, a song could consist of a verse and a chorus. The verse and chorus would then be two different parts.

timbre a common property of a selection of sounds, enabling the listener to recognize that they all emenate from the same source. For example, a drum has a different timbre than a saxophone.

beat pattern a specification of a rhythmic figure, usually used for controlling virtual drum-kits.

white noise a random signal, consisting of all frequencies at once. A radio set between station frequencies picks up white noise.

Chapter I

Analysis

In Section 1, the current state-of-the-art music composition software tools are described. These tools fall into three categories: *sequencing applications*, *modular composition systems*, and *software synthesis languages*. Sequencing applications will be described in detail in Section 1.3, modular composition systems in Section 1.4 and software synthesis languages in Section 1.5. The tools have one similarity: they operate on two forms of data, *waveforms* and *Musical Instrument Digital Interface* data. Waveforms will be explained in Section 1.1 and the Musical Instrument Digital Interface in Section 1.2. The need for a new language for composition of music is described in Section 1.6.

Section 2 defines requirements to such a language. Requirements concerning flexibility are described in Section 2.1 and requirements concerning usability are described in Section 2.2.

Section 3 defines requirements to the language interpreter.

1 State of the Art

1.1 Waveforms

Waveforms can be created by a *sampling* process. Sampling is done by connecting a microphone to the input of an *Analog-to-digital converter* (AD-converter) chip. The membrane of a microphone converts changes in air pressure to voltage values. The AD-converter converts the voltage values to *samples*, which are ordinary signed integers. These integers have a fixed size for a single sampling process. The industry standard sample size is 16 bit at the time of writing. This conversion from voltage to sample is performed at a fixed frequency, called the *sampling frequency*. The industry standard sampling frequency of most music software at the time of writing is 44100

samples per second or 44100 *Herz* (Hz). The samples thus recorded by the sampling process are stored in a list called a *waveform*. If the user wants to record 10 seconds of sound (l) and the sampling frequency is 44100 samples per second (f_s), the number of samples in the waveform (s) is:

$$s = l \cdot f_s = 10 \text{ s} \cdot 44100 \text{ samples/s} = 441000 \text{ samples}$$

The samples in a waveform can be converted back to voltages by a *Digital-to-Analog converter* (DA-converter) chip. This conversion from sample to voltage is also performed at a fixed frequency. The voltages output by the DA-converter can be used to directly control the membrane of a loudspeaker. If conversion is done at the sampling frequency, the sound thus produced by the speaker is similar to the original sound recorded. This similiarity is determined by the sampling frequency as well as the size of the signed integers of the samples. The larger the sampling frequency and the more bits in the integers, the greater the similiarity. This similiarity is denoted *sound quality*.

Alternatively, instead of sampling a sound using a microphone and an AD-converter, waveforms can be created artificially by an algorithm. The proces of creating artificial sounds is called *synthesis*.

1.2 Musical Instrument Digital Interface

The Musical Instrument Digital Interface (MIDI) is a protocol designed for communicating data between *MIDI controllers*, *MIDI sequencers*, and *MIDI instruments*. The interface between such devices is a serial cable connection[8].

A MIDI controller is an input device that generates MIDI *messages*. These messages are typically *note on messages*, *note off messages*, or *controller messages*. Note on and off messages consist of a binary value which is either *note on* or *note off*, a *MIDI note number* which is a 7-bit unsigned integer in the range 0-127, and a *velocity* value, which also is 7-bit and ranges from 0 to 127. Controller messages are general purpose data, which consist of a *controller ID* and a *controller value*. The controller ID defines a register to change in the data of the receiver and the controller value defines the new value to be assigned to said register.

A MIDI sequencer can store the aforementioned messages paired with a timestamp for each message. The set of message-timestamp pairs constitute a *sequence*. The sequence can be replayed by sending the stored messages precisely at a time relative to the corresponding timestamps.

A MIDI instrument reacts to a note on message by making a sound.

MIDI instruments usually use the MIDI note number and the velocity value stored in the note message to alter the sound correspondingly. This means that a MIDI instrument will create a high frequency sound as a response to a large MIDI note number. Similarily, the MIDI instrument will create a low frequency sound as a response to a small MIDI note number. Also, the MIDI instrument will react to a MIDI message with a high velocity value by playing a loud sound, and a low velocity value will create a quiet sound. MIDI instruments also reacts to note off messages, either by simply stopping the sound or by gradually reducing the volume of the sound until it is silenced. MIDI instruments are *synthesizers*, devices capable of synthesis.

1.3 Sequencing Applications

Sequencing applications are software-based MIDI sequencers and operate on a sequence. They are operated in different *modes* by the user. The operation mode determine the significance of one axis of a 2-dimensional visualization, while the other axis signifies time. 2 of the most important modes are described below.

The *arrange* mode visualizes *tracks* as a function of time. A track designates either a connection to a MIDI instrument, an output to a DA-converter, or a virtual output managed by the sequencing application. The user operates on waveforms or on *blocks*, which are time-limited subsets of a sequence. The user can change the relative time at which a waveform or a block will be replayed by moving the block or waveform along the axis denoting time. The user can also move the block or waveform to a different track, thus changing either the receiving MIDI instrument or the DA-converter output. Most sequencing applications also offers the possibility of *looping* a waveform or a block, which automatically repeats the waveform or block.

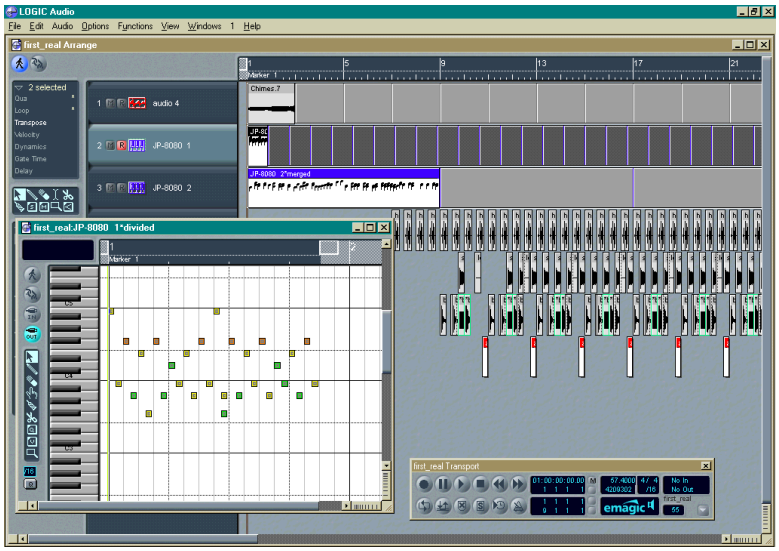
The *matrix* mode gives the user the ability to edit blocks in detail. Blocks consist of MIDI messages. These are arranged in a visualization where MIDI note numbers are displayed as a function of time. Usually, velocity is also displayed using different colours for different velocity values. MIDI messages can be created or changed, by entering or moving them in the visualization.

A screenshot of one of the most popular sequencing applications, Logic [2], is shown in Figure 1 on the facing page.

The main advantage of sequencing applications is that the arrange mode enables the user to easily gain overview of the sequence. The blocks and waveforms are easily copied and moved, which is a powerful editing technique.

The disadvantages of sequencing applications are mainly of cost and flexibility:

♭ **Figure 1:** Logic 4.7



The Matrix Editor is at the bottom-left and behind it is the Arrange Editor

- The cost of the software and the necessary computer hardware is considerable. If the user uses MIDI instruments, the cost is further increased.

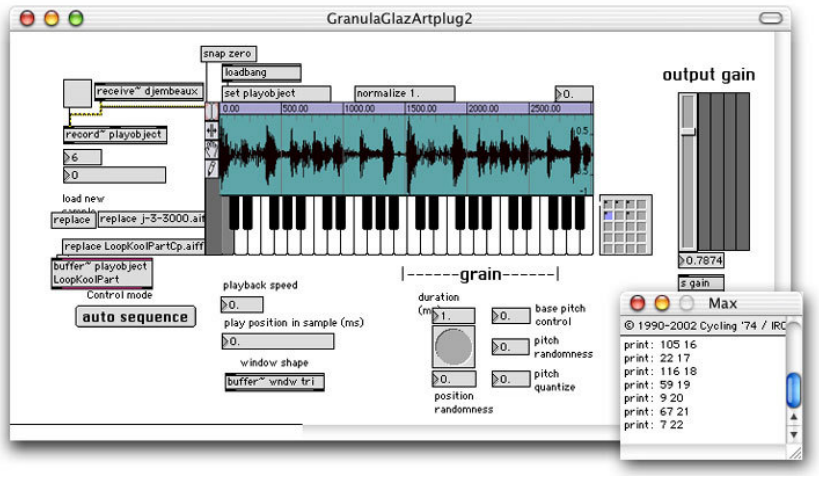
- When using MIDI instruments, the user must obey the boundaries of the MIDI message standard. This means that the user only can use 128 different values for pitch or velocity.

- Waveforms are not dynamically changeable, they basically sound the same each time they are replayed.

1.4 Modular Composition Systems

Modular Composition Systems (MCS) use mathematical functions which are displayed as boxes in a graphical user interface. The boxes can be interconnected by virtual *cables*, which connects the output of one function to the input of another function. The output of these functions can also be connected to a box designated "output" or a similar name. The user can activate the system, which then continously calculates the values of the functions at a fixed sampling frequency. The values entering the "output" box can be converted to samples and sent directly to the DA-converter, thus producing a sound. The way the functions are interconnected determine every aspect of the sound.

♭ **Figure 2:** Max



screenshot by Ircam (copyright 2003)

MCS's can also be made to output MIDI messages, which can be output to a MIDI instrument.

A screenshot of one of the most popular MCS's, Max[3], is shown in Figure 2.

The advantages of MCS's are:

- MCS's are highly flexible. They have the possibility for changing every aspect of the synthesis and sequencing.
- The user can create his own synthesizer in a MCS

The disadvantages of MCS's are:

- It is very difficult to compose music with melodies and rhythms in a MCS.
- They depend on relatively fast computer hardware

1.5 Software Synthesis Languages

Software synthesis languages[1] are similar to MCS's in that the user control every aspect of the synthesis. They are regular programming languages with variables and control structures. There are of course different approaches to the design of software synthesis languages. The most popular approach uses

[1]The term *software synthesis language*, which might be a bit redundant, is used in [10].

2 separate source file formats, a *score file* and an *orchestra file*. The score file is simply a sequence written in ASCII text format. The orchestra file defines *virtual instruments* to be used by the score file.

Virtual instruments are remeniscent of real instruments, because they define a *timbre* (see Terminology on page 8). All notes played by a real instrument will have the same timbre regardless of other parameters such as pitch. Similarily, a virtual instrument can be used to create *virtual notes* that output waveforms which share a common timbre. This idea can be described in object-oriented terminology. The virtual instruments could be viewed as classes, and the virtual notes could be viewed as instances of that class. This terminology will prove useful in the rest of the report, and therefore virtual notes are denoted *virtual instrument instances*.

In some software synthesis languages, each virtual instrument has 3 different categories of variables, *instrument time* variables, *control time* variables, and *sample time* variables. The variable categories differ in the time of evaluation.

Sample time variables are handled like variables in an average procedural language. Each virtual instrument instance produce a special "output" variable. This variable is converted to a sample, and the resulting samples are stored in a waveform. Each rendition of a single sample is called a *sample cycle*.

Control time variable assignments are evaluated every nth sample cycle, where:

$$n = \frac{sampling\ frequency}{control\ rate}$$

The control rate is measured in Hz and is determined by the user.

Assignments to instrument time variables are only evaluated once per virtual instrument instance, during the first sample cycle. This is analogous to the object-oriented concept of the initialization of variables occuring in a constructor of a class.

Music

One of the earliest software synthesis languages was the *Music III* language developed in 1960 by Max V. Mathews and colleagues at Bell Telephone Laboratories. This language has been revised several times and has been used for many years[10].

It employed the method of using a score file and an orchestra file explained above. The orchestra language was similar to assembly code and used variables called *signals*, which could be assigned values from *wave*-generating

functions. A standard example of a wave-generating function is the sine function.

Because of the similarity to assembly code, the language is not very readable.

Csound

Another popular language was *Csound*. Csound was named after the fact that it was implemented in ANSI C, which made it portable between different hardware platforms. Software synthesis languages was rarely portable in 1986, when the language was created. The name would suggest similarities to C, but Csound is even more akin to assembly code than the Music language. Csound used the concept of a score file and an orchestra file, as well as the concept of instrument, controller, and sample time variables.

The readability is like the Music language.

MPEG4 Structured Audio Language

A more modern descendant of Csound is the *MPEG4 Structured Audio language* (MP4-SA). It was created in 1998 by the Machine Listening Group at MIT Media Laboratory as a radical new standard for the *Moving Pictures Expert Group* (MPEG) audio file format[5]. The basic idea of MP4-SA was to contain virtual machine code instead of audio data in a MPEG file.

The concept of the MP4-SA language is very similar to that of Csound. It has the same three categories of variables and is also divided into a score file and an orchestra file. However, the *MP4-SA orchestra language* (SAOL) syntax is very similar to C and contains a lot of built-in functionality to help the programmer perform difficult tasks. The built-in functionality can also be bypassed, and the expert programmer can control everything down to the output of individual samples.

Unfortunately, the *MP4-SA score language* (SASL) is rudimentary, containing only the basic sequence in ASCII format, just like Music and Csound.

The main advantage of software synthesis languages is that they are as flexible and offer as much control as MCS's.

The disadvantages of software synthesis languages are:

- It is difficult to gain overview of a composition by looking at the source code
- The source code does not provide the user with an intuitive sense of what the resulting music will sound like after compilation

1.6 The Need For a New Software Synthesis Language

The power and flexibility of software synthesis languages makes them the tool of choice for users who wants to go beyond the limits of other types of music software. However, the current software synthesis languages have limitations which hinders the creative process. This will be shown using the work process of the language MP4-SA as an example.

Virtual instruments and a sequence definition is needed to create a composition in MP4-SA.

The virtual instruments are defined in a SAOL file. An example of such a file is shown in Example 1 on the following page. This program defines a single virtual instrument called "bass". If more instruments were to be used, much of the code defining "bass" would be reused in the definitions of those virtual instruments.

The sequence is defined in a SASL file. It is unfeasible to write the sequence for a complete composition by hand. Fortunately, the sequence can be created by way of a scripting language. Such a script, written in Python[9], is shown in Example 2 on page 19, and an excerpt of the SASL output is shown in Example 3 on page 19.

As the example shows, even rudimentary compositions require quite a few lines of code. This is detrimental to the user's overview of the composition. Furthermore, the compiling process becomes lengthy due to combined interpretation of a script language and compilation of the results.

A new language should retain the flexibility of the mentioned languages. It should also reduce code size and only employ a single type of source file containing both the virtual instrument definitions and the sequence, to facilitate the user's overview of the composition. Furthermore, the creation of sequences should be made easier by enabling the user to structure, embed and loop parts of his composition. Finally, interpretation should be possible using only a single command.

The users of the new language could be interested in creating professional audio for electronic music. The language might also be used for rendering sequence files with virtual instruments programmed by the user.

♮ **Example 1:** SAOL program (some code omitted)

```
global
{
    srate 44100;
    krate 1050;

    table bass(sample, -1, "/home/schmid/samples/bass.wav");

    route(tonebus, bass);
    send(mixer; 2, .2; tonebus);
    outchannels 2;
}

instr bass(midinote)
{
    imports table bass;
    asig out, ptr;          // 'asig' type is sample time
    ivar freq, tickstep;    // 'ivar' type is instrument time
    ksig samplestep, tick;  // 'ksig' type is control type

    // envelope settings
    ivar atime;   // attack
    ivar rtime;   // release

    // internal env state
    ivar attack;
    ivar release;
    ivar sustain;
    asig env;     // env output

    // calculate sample step value
    samplestep = cpsmidi(midinote) / cpsmidi(60);

    // ENVELOPE COMPUTATION ____________________________________
    atime = 0.01; // attack time (s)
    rtime = 0.1; // decay time (s)

    // computes envelope state
    if (dur > atime + rtime)
    {
        attack = atime;
        release = rtime;
        sustain = dur - (atime + rtime);
    }
    else
    {
        attack = dur/2;
        release = dur/2;
        sustain = 0;
    }
    // compute env from envelope generator
    env = kline(0, attack, 1, sustain, 1, release, 0);

    // OUTPUT SAMPLE ___________________________________________
    out = tableread(bass, ptr);
    output(out*env);

    ptr = ptr + samplestep; // step through sample
}

instr mixer(rt60, wetdry)
{
    // ...
}
```

♮ **Example 2:** Python program that generates the output in Example 3

```
#!/usr/bin/env python

import random

# SUGGESTED INSTRUMENT CLASS ----------------------------------------
class Instrument:
    def __init__(self, name):
        self.name   = name

    def make(self, frombar, tobar):
        print "\n// Instrument.make: ---------------------------------"
        print "//\tname:", self.name
        print "//\tfrom bar", frombar, "to", tobar

# STEP SEQUENCER ----------------------------------------------------
class StepSequencer(Instrument):
    def make(self, frombar, tobar):
        Instrument.make(self, frombar, tobar)

        arp = [ 60,0,0,60, 0,0,0,70, 0,70,0,0, 0,0,0,60,
                0,0,0,0,   0,0,0,0,  0,0,0,0,  63,62,61,0 ]
        for bar in range(frombar, tobar, 2):
            for step in range(len(arp)):
                if arp[step]!=0:
                    stepstart = bar * 4 + step * 0.25
                    print stepstart, self.name, 8.0/len(arp), arp[step]

# MAIN PROGRAM ------------------------------------------------------
bars = 58
random.seed(711)

# song globals
print "0 tempo 80"
print bars*4+3,"end"

bass = StepSequencer("bass")
bass.make(1,bars)
```

♮ **Example 3:** Excerpt of SASL score generated by Python program

```
0 tempo 80
235 end

// Instrument.make: ---------------------------------------
//        name: bass
//        from bar 1 to 58
4.0 bass 0.25 60
4.75 bass 0.25 60
5.75 bass 0.25 70
6.25 bass 0.25 70
7.75 bass 0.25 60
11.0 bass 0.25 63
11.25 bass 0.25 62
11.5 bass 0.25 61

etc...
```

2 Language Requirements

2.1 Flexibility

Flexibility is deemed the most important property of a new software synthesis language. A few of the most important aspects of flexibility are listed below:

Instrumentation The user should be able to create virtual instruments of all sorts. The synthesis should therefore be customizable down to the output of individual samples.

Reusability With Variations Parts of a composition should be reusable, with the posibility of changing certain aspects. For instance, a part could be reused, but with a different virtual instrument.

Effects An effect is a filter (or a function, for the mathematically inclined) with samples as input and output. Typical effects are delay and reverb, which simulate acoustic properties, but any algorithmically describable filter should be creatable.

Randomization It should be possible to randomize every aspect of a composition, including structure, pitch and synthesis.

2.2 Usability

Often, composers (including myself) lose patience with musical composition tools when the creative process becomes too difficult. Therefore, it is very important that a software synthesis language is easy to use.

Learning Curve Composition of simple music should be easy. The user should be able to make his first composition in a few minutes, encouraging him or her to try creating more complex compositions early in the learning process. The language should also report detailed programming errors, to teach the user the rules and constraints of the language as early as possible.

Writability The language should encapsulate tedious tasks into short statements. For example, the usage of waveform files should be made easy and transparent. Also, creating melodies and repeating parts should be accomplished with as few statements as possible.

Readability An experienced user should be able to estimate how a part of a program will sound when rendered without actually invoking the interpreter.

Pattern Sequencing A new software synthesis language should contain a matrix-editing facility for creating *beat patterns* (see Terminology on page 8) and melodies.

Hierarchial Composition A new feature of a software synthesis language would be the ability to embed parts of a composition into larger parts in a hierarchical fashion. This would aid the user in gaining overview of a composition and potentially shorten the source code.

3 System Requirements

Speed The interpretation process must be fast, as not to block the creative process.

Portability The interpreter should be independent of the operating system, that is, compilable on all main platforms.

Selective Interpretation It should be possible to interpret a selected part of a musical piece, as to quickly evaluate how it sounds.

Input and Output The interpreter should read and write the two main uncompressed audio file formats, AIFF and WAV. Furthermore, it should give the user the option to output a MPEG-3 file.

Single Source File The user should be able to create a composition in a single source file, to keep overview of the composition.

Chapter II

Design

I introduce *Development of Intelligent Music using a Hierarchical Composition System* (DIMHCS)[1].

In Section 1, the syntax of DIMHCS is described. In Section 1.1 the grammar is defined, along with operator precedence and associativity rules. An example of an Abstract Syntax Tree (AST) created from the rules in the grammar is given in Section 1.2.

In Section 2, the semantics of the language are explained. Section 2.1 explains the intuitive idea behind the language. In Section 2.2, the basic components of the language are described, including the datatypes and built-in functions. In Section 2.3, the evaluation of expressions is explained, followed by a presentation of control structures in Section 2.4. At this point, the reader is well prepared to investigate how musical compositions are constructed in Section 2.5.

In Section 3, the contextual constraints are described. Section 3.1 explains the type system and Section 3.2 covers the scope rules.

1 Grammar

1.1 Grammar Definition

The terminal symbols of DIMHCS are listed below. Related terminals are placed in the same line. Terminals are separated by whitespaces.

[1]The term "intelligent music" is derived from a genre of music called *intelligent electronica*. The music of this genre is known for complex and ever-changing structure and patterns.

```
block  instrument  dsp
init:  run:
=
while  if  else
tempo  output
route  ->
addwave  default
matrix  select  pitch  velocity  offset  wave  transpose  loop
hold  rest  keep
'
;
[  ]  {  }  (  )
and  or
==  !=  <  >  <=  >=
+  -  *  /  ^
:  note  octave
sin  cos  ?
true  false
```

The following is a definition of the grammar for DIMHCS. The grammar syntax is based on Extended Bachus-Naur Form[4]:

- the EBNF metacharacters are `::= () | * []` and `+`, and they all have their standard meaning
- x_0 `..` x_n means x_0 `|` x_1 `|` ... `|` x_n, where the x's are in lexicographic order. For instance `'a' .. 'c'` means `'a' | 'b' | 'c'` .

```
program                 ::= (  block_definition | instrument_definition
                             | dsp_definition )*

block_definition        ::= 'block' [ identifier ':' ] identifier
                            '{' class_body matrix_definition '}'

instrument_definition   ::= 'instrument' [ identifier ':' ] identifier
                            '{' class_body '}'

dsp_definition          ::= 'dsp' [ identifier ':' ] identifier
                            '{' class_body '}'

class_body             ::= statement_sequence |
                            'init:' statement_sequence
                              [ 'run:' statement_sequence ]

statement_block         ::= '{' statement_sequence '}'
                          |  statement

statement_sequence      ::= statement*

statement               ::= identifier '=' expression ';'
                          |  identifier '[' a_expression ']' '=' expression ';'
```

```
                        |  'if' '(' expression ')' statement_block ';'
                        |  'if' '(' expression ')' statement_block
                               else statement_block ';'
                        |  'while' '(' expression ')' statement_block ';'
                        |  'route' '->' expression ';'
                        |  'output' expression ';'
                        |  'tempo' expression ';'
                        |  'addwave' ( identifier | 'default' ) expression

matrix_definition       ::= ( 'matrix' expression '{' matrix_contents '}' )*

matrix_contents         ::= 'select' matrix_expression optional_list

optional_list           ::= (   'pitch'     matrix_expression
                              | 'velocity'  matrix_expression
                              | 'offset'    matrix_expression
                              | 'wave'      matrix_expression
                              | 'transpose' matrix_expression
                            )*

expression              ::= expression ( 'and' | 'or' ) expression
                        |  a_expression
                               ( '==' | '!=' | '<' | '>' | '<=' | '>=' )
                               a_expression
                        |  a_expression

a_expression    ::= a_expresssion ( '+' | '-' | '*' | '/' | '^' ) a_expression
                |  a_expression ':' ( 'note' | 'octave' )
                |  identifier ':' identifier
                |  a_expresssion '[' ( a_expression | '?' ) ']'
                |  identifier
                |  literal
                |  builtin_function
                | '(' a_expression ')'

builtin_function        ::= 'sin' '(' a_expression ')'
                        |  'cos' '(' a_expression ')'
                        |  '?'   '(' a_expression ',' a_expression ')'

literal                 ::= pitch_literal | numeric_literal | list_literal
                        |  matrix_control_literal | boolean_literal | string

identifier              ::= lowercase_letter identifier_char*

pitch_literal           ::= uppercase_letter pitch_char* digit+

numeric_literal         ::= digit+ | digit+ '.' digit* | digit* '.' digit+

list_literal            ::= expression_list | short_list

expression_list         ::= '{' a_expression ( ',' a_expression )* '}'

short_list              ::= '[' short_list_content* ']'

short_list_content      ::= identifier | numeric_literal | string | pitch_literal
                        |  '.' | "'" | '-'

matrix_control_literal   ::= 'hold' | 'rest' | 'keep'

boolean_literal         ::= 'true' | 'false'

lowercase_letter        ::= 'a' .. 'z' | 'æ' | 'ø' | 'å'
```

```
uppercase_letter        ::= 'A' .. 'Z' | 'Æ' | 'Ø' | 'Å'

digit                   ::= '0' .. '9'
```

The operators in expressions have precedence and associativity according to Table 1. Higher precedence operators are listed at the top of the table and vice versa. The precedence and associativity of the operators in DIMHCS is largely based on those of operators in C.

♯ **Table 1:** Operator Precedence

Operators	Associativity (When Applicable)
`( )`	-
`[ ]`	-
`:`	-
unary `+ -`	-
`^`	right-to-left
`* / %`	left-to-right
`+ -`	left-to-right
`== != < > <= >=`	-
`and or`	left-to-right

Furthermore, DIMHCS use the same commenting scheme as C++ and Java.

1.2 Abstract Syntax Tree Example

A DIMHCS program constructed from the rules in the grammar can be converted into an *Abstract Syntax Tree* (AST) by a parser. Example 4 on the next page shows an example of this process.

♮ Example 4:

```
instrument bleep
{
    sample = sin(lifetime / 100);
    output sample;
}
```

This is the example code, which is converted into the AST below. Note that the AST is rotated 90 degrees compared to standard notation[1]. Also, terminal symbols are enclosed in '' and identifier names and literals are enclosed in ('').

```
- program
    `- instrument_definition
        |- 'instrument'
        |- identifier ('bleep')
        |- '{'
        |- class_body
        |    `- statement_sequence
        |        |- statement
        |        |   |- identifier ('sample')
        |        |   |- '='
        |        |   |- expression
        |        |   |   `- a_expression
        |        |   |       `- builtin_function
        |        |   |           |- 'sin'
        |        |   |           |- '('
        |        |   |           |- a_expression
        |        |   |           |    |- a_expression
        |        |   |           |    |    `- identifier ('lifetime')
        |        |   |           |    |- '/'
        |        |   |           |    `- 'a_expression
        |        |   |           |         `- literal
        |        |   |           |             `- numeric_literal ('100')
        |        |   |           `- ')'
        |        |   `- ';'
        |        `- statement
        |            |- 'output'
        |            |- expression
        |            |   `- a_expression
        |            |       `- identifier ('sample')
        |            `- ';'
        `- '}'
```

2 Semantics

2.1 Intuition

The basic idea of DIMHCS is as follows: the user creates a source file containing definitions of *Virtual Instruments* (VI's), *blocks* that control the VI's, and possibly *Digital Signal Processors* (DSP's), through which the output of blocks can be routed. The user then activates the DIMHCS interpreter,

which performs analysis of the source code, possibly reads waveform files, and outputs a new waveform file containing the finished composition. The creation of the new waveform file through interpretation of the source code is called *rendering*.

Blocks can define a sequence for controlling VI's through a *matrix definition*. The matrix definition in DIMHCS is similar to matrix editors used in sequencing applications, and blocks in DIMHCS are also similar to those of sequencing applications (see Section 1.3 on page 12). Matrix definitions can even control other blocks, which makes a hierarchical structure possible in a composition. The possibility of routing the output of a block through a DSP is remeniscent of MCS's (see Section 1.4 on page 13).

DIMHCS is based on an object-oriented idea, in the sense that VI's, blocks and DSP's all have definitions not unlike classes. Furthermore, these definitions are *instantiated* during the rendering process, in which the output waveforms are calculated. However, DIMHCS is not an object-oriented language, since the definition of new class types isn't possible.

The purpose of DIMHCS is to render an output waveform. This is done by rendering single samples and storing them in a list. The rendering of each sample is called a *sample cycle*. As all VI's , blocks and DSP's are essentially independent, output waveforms can be rendered separately for each VI, block and DSP and mixed together afterwards. The output waveform is supposed to be replayed at a fixed frequency, the *global sampling frequency*. For now, DIMHCS is designed to output waveforms at a fixed sampling frequency of 44100 Hz. This could easily be made customizable in a later version.

2.2 Components

Before investigating the informal semantics of the components of DIMHCS, a few definitions are needed. $\mathbb{Q}$ is the set of rational numbers and $\mathbb{Z}^*$ is the set of non-negative integers in the following:

$\mathbb{Q}_{float} \subset \mathbb{Q}$	is the set of all numeric values
$\mathbb{B} = \{$ `true` , `false` $\}$	is the set of all boolean values
$P = \{0, 1, ..., 11\} \times \mathbb{Z}^*$	is the set of all pitches
E	is the set of all DIMHCS expressions

The function **V** returns the value of non-list DIMHCS expressions (see Example 5 on the following page). It will be implicitly defined in the following sections. Its domain and range are defined below:

$$\mathbf{V} : E \to \mathbb{Q}_{float} \cup \mathbb{B} \cup P$$

♮ **Example 5:** The value function **V**

Note that `2 + 2` are 3 DIMHCS language tokens, not a mathematical expression.

$$\mathbf{V}[\![\ \texttt{2 + 2}\ ]\!] = 4$$

An *atomic* datatype is an indivisible datatype. It cannot be described as a composition of other types. DIMHCS has the following atomic datatypes: *boolean*, *numeric*, *string*, and *pitch*.

Numerics, Booleans and Strings

The simple atomic datatypes of DIMHCS are:

numeric A numeric value is a signed floating-point number. Its literals are written in the same syntax as floating-point numbers in C or Java.

boolean A boolean value is either `true` or `false`

string The string type is not atomic in most languages, but DIMHCS offers no possibility of changing the characters inside a string, nor of reading individual characters from a string. Thus, strings are *constant* in DIMHCS, which justifies grouping them with the atomic datatypes.

Example 6 shows assignments to variables using numeric, boolean and string literals.

Pitches

A pitch value describes a frequency using a pair of a *note* value and an *octave* value. This concept is derived from music theory. The note and octave parts

♮ **Example 6:** Numeric, Boolean, and String Literals

```
b = true;       // an assignment to a boolean literal
n = 3.14159265; // an assignment to a numeric literal
s = "hello";    // an assignment to a string literal
```

♮ **Example 7:** Pitch literals

```
p1 = C3;         // assignments to pitch literals
p2 = D#0;
p3 = Fb16;
```

are of numeric type and can be extracted using 2 *access* operators, `:note` and `:octave`.

The first part, the note name, is written as an uppercase letter followed by an optional `#` or `b`. The uppercase letter must be one of `C`, `D`, `E`, `F`, `G`, `A`, `B`/`H`[2], otherwise an error is issued. The note names represent a note number. They are mapped according to Table 2. The mapping is based on music theory and will not be explained here.

♯ **Table 2:** Note Names

Note Names	Note Number
C, B#, H#	0
C#, Db	1
D	2
D#, Eb	3
E, Fb	4
F, E#	5
F#, Gb	6
G	7
G#, Ab	8
A	9
A#, Bb, Hb	10
B, H, Cb	11

Directly following the note name is the octave number, which is written as a non-negative integer (see Example 7 for example pitch literals).

The *well-tempered tuning* (see Terminology on page 8) allows a precise calculation of a frequency (f) from a note number (n) and an octave (o):

$$f(n, o) = f_b \cdot 2^{n/12+o}$$

f_b (the *base frequency*) is the frequency of note number 0, octave 0. In the case of DIMHCS, this note is named `C0`. `C0` traditionally (see [12]) has

[2] `B` and `H` are aliases. This is due to an old discrepancy between danish / german and international music theory. Danish and german composers use `H` instead of `B`.

the frequency:

$$f_b = \sqrt[4]{2} \cdot 27.5 \text{ Hz}$$

With this knowledge, the frequency of any note/octave pair can be calculated by:

$$f(n, o) = \sqrt[4]{2} \cdot 27.5 \text{ Hz} \cdot 2^{n/12+o}$$

Example 8 shows calculation of a frequency from a pitch literal.

♮ **Example 8:** Frequency Calculation

We will calculate the frequency of the pitch `A3`. The note number for `A` is 9 according to Table 2 on the page before, so we have:

$$\begin{aligned} f(9,3) &= \sqrt[4]{2} \cdot 27.5 \text{ Hz} \cdot 2^{9/12+3} \\ &= 27.5 \cdot 2^{1/4} \cdot 2^{3/4+12/4} \text{ Hz} \\ &= 27.5 \cdot 2^{1/4+3/4+12/4} \text{ Hz} \\ &= 27.5 \cdot 2^4 \text{ Hz} \;=\; 27.5 \cdot 16 \text{ Hz} \;=\; 440 \text{ Hz} \end{aligned}$$

◇ **Implementation Status Note:** *The original DIMHCS design featured the possibility of combining several pitches into one to create a* chord *(see Terminology on page 8). This feature was skipped due to lack of time. Also, the mapping from note names to numbers was not implemented before the report deadline.*

Lists

DIMHCS has a list datatype. The entries in a list can be of any type, including other lists.

Lists dynamically expand to the demands of the program, including assignment to an element beyond the current limit of the list. When such an assignment occurs, there is possibility for one or more unassigned entries between the previously last element in the list and the new last entry. Such unassigned entries may not be accessed, and any attempt to do so results in an interpretation error.

List literals can be written in 1 of 2 different styles, *expression lists* and *short lists*. Expression lists are written using `{` and `}` for delimiters and a `,` between each entry. They can contain *expressions* (see Section 2.3). Short lists are written using `[` and `]` for delimiters and no additional tokens between the actual entries. They may only contain identifier or literal entries.

Example 9 shows assignments to the 2 different kinds of lists literals.

♮ **Example 9:** List literals

```
pitch_identifier = C3;

short_list       = [ pitch_identifier Eb3 F3 ];
                                        // this is a list assignment using the
                                        // short list syntax.

expression_list  = { 3, 2+2, 10/2 };    // this is a list assignment using the
                                        // expression list syntax
```

Example 10 shows a syntax error, where an expression is used inside a short list literal.

♮ **Example 10:** Short list syntax error

```
// sl = [ 2+2 ];  // this is a syntax error: expressions are not allowed in short
                  // lists!
```

◇ **Implementation Status Note:** *Interpretation of lists wasn't implemented in time for the report deadline.*

Class Types

The 3 class types in DIMHCS are VI, block and DSP. Class definitions of these types can be written by the user. They can all contain statements such as assignments. No statement can be written outside a class definition. Classes are usually *instantiated* many times during interpretation of a DIMHCS program. Each instance of a class has its own *environment*, defining identifiers accessible from the statements in the class definition. Special features common for the 3 class types are *sections* and *inheritance*.

The functionality of having time-categorized variables discussed in Section 1.5 on page 14 is achieved in DIMHCS using a different approach. The statements in class definitions are divided into 2 sections, the *init section* and the *run section*. The statements inside the init section will be evaluated in the

beginning of the first sample cycle of a class instance. The statements inside the run section will be evaluated during every sample cycle. This approach is inspired from the object-oriented concept of a constructor method. The sectioning is indicated by the `init:` and `run:` keywords.

Example 11 shows the syntax of init and run sections.

♮ **Example 11:** Class Sections

```
block main
{
init:
    x = 0;      // This is only evaluated during the first sample cycle

run:
    x = x + 1;  // This is evaluated during every sample cycle
}
```

This block sets x to 0 when it is instantiated. During the first and every following sample cycle, it will be incremented. This means that x is 1 after the evaluation of the first sample cycle, 2 after the second, etc.

If no sections are specified, all statements are considered part of an implicit run section. Example 12 shows a definition of a block with an implicit run section.

♮ **Example 12:** Implicit Run Section

```
block main
{
    x = 2;  // This is evaluated during every sample cycle
}
```

This block sets x to 2 during every sample cycle.

◇ **Implementation Status Note:** *The init- and run section functionality wasn't implemented in time for the report deadline. Every statement is considered part of an implicit run section.*

DIMHCS also provides a simple but usable inheritance mechanism for class types. Any class can define a *parent class*. The class and the parent class must have the same type. The semantics for the inheritance are represented by this very straightforward algorithm:

```
To evaluate the first sample cycle of a class instance:

1. Evaluate first the statements of the init section of the parent class,
   then the statements of the init section of the class itself.

2. Proceed by evaluating the statements of the run section of the parent
   class, then the statements of the run section of the class itself.

To evaluate the subsequent sample cycles of a class instance:

1. Evaluate the statements of the run section of the parent
   class, then the statements of the run section of the class itself.
```

Example 13 shows the syntax of class inheritance.

♮ **Example 13:** Class Inheritance

```
instrument bad_instrument
{
init:
    x = 0;
run:
    x = x + 0.01; // increment x during each sample cycle
                  // unfortunately, this virtual instrument defines no output
}

instrument better_instrument : bad_instrument // inherits from bad_instrument
{
    output sin(x); // big improvement: the virtual instrument outputs samples
}
```

`better_instrument` inherits the init section and the run section from `bad_instrument`. So, upon reaching the `output sin(x)` statement in the implicit run section, `x` is defined and will be increased like in `bad_instrument`.

◇ **Implementation Status Note:** *The inheritance functionality wasn't implemented in time for the report deadline.*

Virtual Instruments

A VI definition can contain assignment statements, control strucures, as well as the `output` and `addwave` statements. A VI definition is written using the `instrument` keyword. VI statements are written between the curly brackets `{` and `}` of the VI definition.

A VI has a set of *waveforms* associated with it. These waveforms are

specified with the `addwave` statement. It has 2 arguments, a identifier naming the waveform and a string. The string specifies the filename of a loadable WAV file. `addwave` statements are written in one of the init or run sections, but the waveforms are loaded before the interpretation stage. Thus, the waveforms are *static* and *constant* and associated with a VI, not a VI instance.

A VI instance has a selected waveform, which is represented in the environment of the VI instance by the `selected_wave` identifier. `selected_wave` has numeric list type. It also has identifiers called `selected_pitch` (pitch type) and `selected_velocity` (numeric type) defined in the environment.

The number of sample cycles during which a VI instance has existed is represented in its environment by the numeric `lifetime` identifier. The environment of a VI instance also contains the `length` identifier, which contains the total number of sample cycles the VI instance will exist before it is un-instantiated.

The purpose of a VI is to output samples. This is accomplished by the `output` statement, which has a numeric argument. This argument is the output sample of the VI instance.

Example 14 shows a definition of a very simple VI.

Example 14: Very Simple VI Definition

```
// very simple virtual instrument definition
instrument white_noise
{
    output ?(0,1); // output a random sample with a value between 0 and 1
}
```

This VI outputs *white noise* (see Terminology on page 8). The noise is generated by the random function.

Example 15 on the next page defines 2 VI's that output sine waves.

♮ **Example 15:** Sine Wave VI's

```
// another very simple virtual instrument definition
instrument simple_bleep
{
    output sin(lifetime / 100);
}
```

This VI outputs a sine wave. The argument of the sine function determines the frequency of the output wave. `lifetime` increases with 1 for each sample cycle. Each sample cycle will take up 1/44100 of a second given a global sampling frequency of 44100 Hz.

```
// virtual instrument that output a 440 Hz sine wave
instrument bleep
{
    output sin(lifetime * 440 * (2 * pi) / 44100);
}
```

This code defines an instrument that always outputs a sine wave of frequency 440 Hz (`A3`), given a global sampling frequency of 44100 Hz.

Example 16 shows a definition of a VI that use the value of `selected_pitch` to generate a frequency corresponding to that pitch. It employs the formula from page 28 to calculate a frequency from a note/octave pair.

♮ **Example 16:**

```
// virtual instrument that output a sine wave with frequency corresponding
// to selected_pitch
instrument pitched_bleep
{
init:
    freq = 27.5 * 2^(1/4) * 2 ^ (selected_pitch:note / 12 + selected_pitch:octave);

run:
    output sin(lifetime * freq * (2 * pi) / 44100);
}
```

Example 17 on the next page shows a VI that replays a waveform.

♮ **Example 17:**

```
// virtual instrument that replays a waveform
instrument sample_player
{
init:
    sample_number = 0;
    addwave piano_wave "piano.wav";

run:
    output piano_wave[sample_number];
    sample_number = sample_number + 1;
}
```

The perceived frequency of a waveform should be changable during playback. To perform this task, *resampling* is used. A basic example of resampling :

> We have a waveform consisting of 1000 samples recorded at the sampling frequency 44100 Hz. The waveform was a recording of a sine wave at 440 Hz. From this waveform we extract all the odd samples. We now have a waveform of 500 samples. This waveform is replayed at a sampling frequency of 44100 Hz. This of course causes the produced sound to play for hals the duration of the original waveform. Furthermore, the new sine wave sounds as if *it were a recording of a 880 Hz sine wave.*

Resampling is used by all standard sound cards and most synthesizers. Almost all systems that use resampling to change pitch, use interpolation algorithms that enhance the result of the resampling. Such algorithms are easily created in DIMHCS, but this is beyond the scope of this report.

Example 18 on the facing page shows a VI called `default_instrument` that uses resampling. The `default_instrument` itself cannot be used directly, but VI's can inherit from it. It is automatically included in source code format by the parser and inserted before the source code of the user. This approach is superior to a "hard-coded" version created in the global environment, because users easily can enhance the default instrument without having to recompile the entire DIMHCS system.

♮ **Example 18:**

```
// default instrument
instrument default_instrument
{
init:

    // A easily customizable version of the frequency formula
    scale = { 0, 1/12, 2/12, 3/12, 4/12, 5/12,
                6/12, 7/12, 8/12, 9/12, 10/12, 11/12 };
    basenote_frequency  = 27.5 * 2^(1/4); // C0
    octave_base         = 2;
    freq = basenote_frequency * octave_base ^
              (scale[selected_pitch:note] + selected_pitch:octave);

    ptr  = 0;
    wave = selected_wave;

run:
    output selected_velocity * selected_wave[ptr];

    // increment sample number
    ptr += freq / basenote_frequency;
}
```

◇ **Implementation Status Note:** *The ability to inherit from the default instrument was not implemented in time for the report deadline.*

Blocks and Matrix Constructs

Blocks are responsible for instantiating VI's. Valid block statements include assignment and control structures, as well as the `tempo` and `route` statements.

Instantiating VI's is done via a *matrix definition* and a *tempo definition*. A matrix definition is a defined by a *subdivision* value and a set of lists.

The tempo definition is done via the `tempo` statement and sets a numeric value measured in *beats per minute* (BPM). BPM is the default tempo unit of rhythmic music. The term *beats* is in DIMHCS a synonym for 1/4 *whole note* (WN), the basic time unit of DIMHCS. The subdivision value is also measured in WN. If the block in question is instantiated by another block, it may omit the tempo definition, as the tempo will be the same as that of the instantiating block by default.

When the block is rendered, the matrix definition serves as a schedule for the rendering of VI's or other blocks. Informally, a matrix definition is the DIMHCS equivalent of a sequence, it decides which VI or block should play at what time and how.

Example 19 shows a block definition. The tempo definition sets the tempo to 60 BPM. The matrix definition sets the subdivision value to 1/8 WN. It selects the `pitched_bleep` VI and selects 3 pitch values `C3`, `D3`, and `E3` to be played. When rendered, the waveform resulting from this block will sound like the 3 notes `C3`, `D3`, and `E3` played by the `pitched_bleep` VI. Note that although the matrix definition is placed beneath the `run:` section, it is not a part of this section. The matrix definition is not a statement.

♮ Example 19:

```
block simple
{
init:
    tempo 60;   // tempo definition

run:
    matrix 1/8  // matrix definition - '1/8' is the subdivision
    {
        select [ pitched_bleep ]
        pitch  [ C3 D3 E3 ]
    }
}
```

The schedule defined by a matrix definition is divided into *time frames* with length (l) depending upon the tempo definition (tmp), the subdivision value (s), and the global sampling frequency (f_S).

The length (measured in samples) of a time frame in a matrix definition can be calculated by:

$$l \;=\; \frac{s}{tmp \cdot 1/4 \text{ WN/beats}} \cdot 60 \text{ s/m} \cdot f_S \text{ samples/s}$$

The list set in the matrix definition always contains one or more *select lists*, selecting a sequence of blocks or VI's to be instantiated. The select list may contain either all blocks or all VI's, not a mixture of the two. Each select list is followed by one or more of these lists:

pitch list If the select list selects VI's, the pitch list defines the `selected_pitch` in the environments for those VI's.

velocity list If the select list selects VI's, the velocity list defines the `selected_velocity` in the environments for those VI's.

wave list If the select list selects VI's, the wave list defines the `selected_wave` in the environments for those VI's.

transpose list If the select list selects blocks, the transpose list defines values that will be added to the note number of all pitch values occuring in that block.

offset list The offset list defines values to be added to the start and end of the time frames for the selected VI's or blocks.

If the user specifies a pitch / wave / velocity list for a block or a transpose list for a VI, an error is issued.

During the rendering process, each select list is rendered separately, and the resulting waveforms are mixed together.

Actually, the matrix definition of Example 19 on the facing page shows that for the second and third time slot in the matrix, the select list defines no instrument to be played. This is not necessary, the `pitched_bleep` will "stay selected".

◇ **Implementation Status Note:** *Interpretation of blocks and matrix definitions wasn't implemented in time for the report deadline. Some features of algorithms for rendering matrix definitions are not documented here.*

Digital Signal Processors

A DSP defines a filter through which all the output of a block can be routed. This is done in the block with the `route` statement. A DSP can contain assignment statements and the `output` statement.

DSP's are only instantiated if a block is routed through them. As each block will be rendered individually, only one input block exist for each DSP instance. The input from a block is represented in the environment of a DSP by the `input` identifier.

Example 20 on the next page shows a simple DSP, which acts as a filter removing high frequencies from the input block.

♮ **Example 20:**

```
block simple
{
init:
    tempo 60;                   // tempo definition
    route -> lowpass_filter;    // route output through 'lowpass_filter'

run:
    matrix 1/8
    {
        select [ pitched_bleep ]
        pitch  [ C3 D3 E3 ]
    }
}

dsp lowpass_filter
{
init:
    buffer = 0;

run:
    output input + buffer / 2;
    buffer = input;
}
```

`lowpass_filter` outputs the average of the current input sample and the previous input sample. This filters out output high frequencies in the output.

◇ **Implementation Status Note:** *Interpretation of DSP's wasn't implemented in time for the report deadline.*

Built-in Functions

Although DIMHCS does not allow the user to create new functions, a few built-in functions has been provided. Any number of built-in functions could easily be supported in later versions of DIMHCS, but the current implementation has the functions `sin()`, `cos()`, and `?()`.

`sin()` and `cos()` are standard trigonometric functions, returning the sine and cosine function of their numeric argument. The argument is expected to be measured in radians.

The `?()` function is also quite standard, besides its somewhat strange appearance. It takes 2 numeric arguments and returns a random numeric value, such that:

$$a \leq \texttt{?(}\ a\ \texttt{,}\ b\ \texttt{)} \leq b$$

Example 21 on the facing page shows usage of the built-in functions.

Example 21: Built-in functions

```
angle = ?(0, 2 * 3.141592);   // 'angle' is assigned a random value between 0 and
                              // ca. 2 pi
x = cos(angle);               // 'x' is assigned the cosine of that angle
y = sin(angle);               // 'y' is assigned the sine of that angle
```

2.3 Expressions

DIMHCS evaluates expressions similarly to C or Java. This includes the use of parentheses, which is not examined further. The semantics of the operators in DIMHCS are now investigated, grouped after the type of the operands.

Numeric Operators

Numeric operators are used on numeric type operands and they include the usual binary operators `+ - * /`. Unary `+ -` also work. The binary comparison operators `== != < > <= >=` operate on numeric subexpressions in the usual way and return a boolean value.

The operators listed above are all fairly standard in languages with expressions. Apart from these, DIMHCS employ an exponentiation operator `^` and a floating point modulus operator, `%`. Modulus is traditionally undefined for non-integers, but DIMHCS numeric values are all floating point numbers. The semantics of this operator is given below. First we define `MAX_DOUBLE` to be the maximum floating point number representable on the computer platform. Then we define a number N:

$$\exists\, n \in \mathbb{Z}^+ (N = 10^n) \wedge N > \texttt{MAX\_DOUBLE}$$

The semantics for the `%` operator are then:

$$\mathbf{V}[\![\ a \,\%\, b\ ]\!] = \frac{aN \bmod bN}{N}$$

where 'mod' is the standard integer modulus operator. These semantics are similiar to the `%` operator of Python[9].

The operators are listed in Table 3 on the next page. Unless specified, the operators have the same semantics as C or Java.

Boolean operators

Boolean `and` and `or` are used on boolean operands.

♯ **Table 3:** Numeric Operators

Operators	Semantics
unary `+ -`	`-` means negation, `+` is only included for completeness
`^`	exponentiation ($\mathbf{V}[\![\ a\ \texttt{\^{}}\ b\ ]\!] = a^b$)
`* / %`	multiplication, division and modulus
`+ -`	addition and subtraction
`== != < > <= >=`	binary comparison operators

♯ **Table 4:** Boolean Operators

Operators	Semantics
`and`	$\mathbf{V}[\![\ a\ \texttt{and}\ b\ ]\!] \equiv \mathbf{V}[\![\ a\ ]\!] \wedge \mathbf{V}[\![\ b\ ]\!]$
`or`	$\mathbf{V}[\![\ a\ \texttt{or}\ b\ ]\!] \equiv \mathbf{V}[\![\ a\ ]\!] \vee \mathbf{V}[\![\ b\ ]\!]$

The operators are listed in Table 4.

Pitch operators

Pitch subexpressions can use the unary access operators `:note` and `:octave`, which return a numeric value. `==` and `!=` also work on pitch operands. Apart from this, pitch subexpressions can use the `+` and `-` operators for *transposition*, which means adding a numeric value to or subtracting a numeric value from the note number of the pitch subexpression. Example 22 demonstrate this feature.

The operators are listed in Table 5 on the facing page.

List operators

Lists can use the unary postfix subscript operator `[x]`, where `x` is a numeric index value (starting with 0 as the first value) or `?` for a random index. As

♮ **Example 22:**

```
p = C3 + 2;
```

This example transposes the pitch `C3` up by 2. $\mathbf{V}[\![\ \texttt{C3}\ ]\!] = (0, 3)$. The value of `p` is changed to $(2, 3)$. This value is equal to the value of the pitch literal `D3`

♯ **Table 5:** Operator Precedence

Operators	Semantics
`+ -`	transposition
`:note`	note number (numeric)
`:octave`	octave number (numeric)
`==`	$\mathbf{V}[\![\ a\ \texttt{==}\ b\ ]\!] \equiv (\mathbf{V}[\![\ a\ \texttt{:note}\ ]\!] = \mathbf{V}[\![\ b\ \texttt{:note}\ ]\!]) \wedge (\mathbf{V}[\![\ a\ \texttt{:octave}\ ]\!] = \mathbf{V}[\![\ b\ \texttt{:octave}\ ]\!])$
`!=`	$\mathbf{V}[\![\ a\ \texttt{!=}\ b\ ]\!] \equiv \neg(\mathbf{V}[\![\ a\ \texttt{==}\ b\ ]\!])$

♯ **Table 6:** List Operators

Operators	Semantics
`[ ]`	subscripting
`+`	concatenation
`*`	concatenation with itself

the index value is a floating point number, the actual index is rounded to the nearest integer before indexing. This is an important feature when selecting a sample from a waveform at a different sample rate than the recorded (see the dicussion of resampling on page 36). The binary operators `+ *` does concatenation operations (see Example 23).

♮ **Example 23:**

```
list  = [ 1 2 3 ];
x     = list[2];   // y is 3
x     = list[?];   // x could be 1, 2 or 3
list2 = list * 2;  // equals [ 1 2 3 1 2 3 ]
```

♯ **Table 7:** Instrument Operators

Operators	Semantics
:	access waveform in instrument

♮ **Example 24:** If ... Else Control Structure

```
if(2 == 2) x = 4;;        // note the extra semicolon - the syntax requires it

if(x > 3)
{
    x = x + 1;
    pitch = C3;
};

if(pitch != D3) x = 2; else x = 3;;

if(x >= 2) pitch = C4;
else
{
   pitch = D4;
   x = 10;
};
```

Instrument operator

A waveform can be selected from an instrument by the binary : accessor operator. The left subexpression must be an instrument and the right a waveform declared with `addwave` in that instrument. Note that since waveforms are constant, they may not be changed by assignment.

◇ **Implementation Status Note:** *The instrument operator wasn't implemented in time for the report deadline.*

2.4 Control Structures

If ... Else

DIMHCS uses an `if` ... `else` control structure similar in functionality to that of C or Java. However, the syntax is slightly different, because of the different structure of statement blocks in DIMHCS (see Grammar Definition on page 23)

The semantics of the `if` ... `else` control structure are the same as that of `if` ... `else` control structures of C or Java, and it will not be discussed further.

Example 24 shows usage of the `if` ... `else` control structure.

♮ **Example 25:** While Control Structure

```
x = 0;

while(x <= 10) x = x + 1;

while(x >= 0)
{
    y = x / 2;
    x = x - y;
}
```

While

DIMHCS uses a `while` control structure similar in functionality to that of C or Java. The syntax for the `while` control structure is slightly different from C or Java (see Grammar Definition on page 23).

The semantics of the `while` control structure are the same as that of `while` control structures of C or Java, and it will not be discussed further.

Example 25 shows usage of the `while` control structure.

2.5 Constructing a Composition

As the components of DIMHCS have been presented, an example of how to construct an actual composition in DIMHCS is in order. A composition consists of one or more block definitions and one or more VI definitions. Furthermore, the blocks can be routed through DSP's, which also should be defined.

We will start by defining the VI's. We'll inherit from `default_instrument` as explained on page 36:

```
instrument bass : default_instrument
{
    addwave default "elecbass.wav";
}

instrument drumkit : default_instrument
{
    addwave bassdrum  "bdrum.wav";
    addwave snaredrum "snare.wav";
    addwave hihat     "hihat.wav";
}
```

All the VI definitions do is define waveform files. These waveform files can then be selected from a matrix definition through the wave list.

We will define 2 blocks that has matrix definitions that instantiate the VI's. No tempo is set, as these blocks will be instantiated from another block.

```
block bassline
{
    matrix 1/8
    {
        select [ bass ]
        pitch  [ F#2 C#2 E2 F#2 E2 C#2 H1 C#2 ]
    }
}

block beat
{
    bd = drumkit:bassdrum;
    sn = drumkit:snaredrum;
    hh = drumkit:hihat;

    matrix 1/8 // this matrix definition defines 2 sequences, that are to be
               // rendered into one rhythm waveform. The first 'hh' is
               // mixed together with the first 'bd', etc.
    {
        select [ drumkit ]
        wave   [ hh ] * 8         // a simple hihat rhythm
        select [ drumkit ]
        wave   [ bd . sn . ] * 2  // alternating bass drum and snare drum
    }
}
```

A DSP filter is also included. We'll use the filter from Section 2.2 on page 39.

```
dsp lowpass_filter
{
init:
    buffer = 0

run:
    output input + buffer / 2;
    buffer = input;
}
```

We'll route the "bassline" block through the filter:

```
block bassline
{
    route -> lowpass_filter; // <- this line was added

    matrix 1/8
    {
        select [ bass ]
        pitch  [ F#2 C#2 E2 F#2 E2 C#2 H1 C#2 ]
    }
}
```

Usually, any DIMHCS composition has a block called "main", which is rendered automatically by the DIMHCS interpreter. Here, the main block, which instantiate the other blocks, must have a tempo defintion. The "main" block is listed below:

```
block main
{
    tempo 110;

    matrix 1
    {
        select [ . . bassline - - - ]
        select [ beat ] * 8
    }
}
```

◇ **Implementation Status Note:** *The current version of the interpreter cannot render this example.*

3 Contextual Constraints

3.1 Type System

Formal Type System

DIMHCS has the following basic datatypes: boolean, numeric, pitch, string, list, instrument, block, and dsp.

Formally, the basic datatypes is defined as a set T_{basic}:

$$T_{basic} = \{\text{boolean}, \text{numeric}, \text{pitch}, \text{string}, \text{list}, \text{instrument}, \text{block}, \text{dsp}\}$$

Lists have entries of a subtype. A type description that adequately describes lists is element in the recursive set T:

$$T = T_{basic} \times T$$

Elements in T are pairs (t, t_{list}). The field t_{list} is undefined for all values of t except list (see Section 2.2 on page 30). For simplicity, only the first element in the pair is written for all non-list types.

Expressions (which includes identifiers and literals) and nodes of an AST can have a type. If E is the set of all expressions, a function $\mathbf{T}$ is defined, that returns the type of a given expression:

$$\mathbf{T} : E \to T$$

Example 26 on the following page shows some example output of the function $\mathbf{T}$.

♮ **Example 26:** The type of expressions

$$\begin{aligned}
\mathbf{T}[\![\ \texttt{2.718281}\ ]\!] &= \text{numeric} \\
\mathbf{T}[\![\ \texttt{E2}\ ]\!] &= \text{pitch} \\
\mathbf{T}[\![\ \texttt{2 + 2 == 5}\ ]\!] &= \text{boolean} \\
\mathbf{T}[\![\ \texttt{"piano.wav"}\ ]\!] &= \text{string}
\end{aligned}$$

Of course, $t = \text{list}$ for all lists. t_{list} is the type of the entries in the list. All entries must have the same type, otherwise a type error is issued. An example of list type:

$$\mathbf{T}[\![\ \texttt{\{ 1 \}}\ ]\!] = (\text{list}, (\text{numeric}, \textit{undefined}))$$

For simplicity, this is written as:

$$\mathbf{T}[\![\ \texttt{\{ 1 \}}\ ]\!] = (\text{list}, \text{numeric})$$

Example 27 shows types of lists. Example 28 shows a list assignment that results in a type error.

♮ **Example 27:** the type of lists

```
numeric_list    = { 1, 2 };          // 'numeric_list' is a list of numeric entries
pitch_list      = { C3, D3 };        // 'pitch_list' is a list of pitch entries
list_list  = { numeric_list, { 1 } }  // 'list_list' is a list of (numeric) list entries
```

$$\begin{aligned}
\mathbf{T}[\![\ \texttt{numeric_list}\ ]\!] &= \mathbf{T}[\![\ \texttt{\{ 1, 2 \}}\ ]\!] = (\text{list}, \text{numeric}) \\
\mathbf{T}[\![\ \texttt{pitch_list}\ ]\!] &= \mathbf{T}[\![\ \texttt{\{ C3, D3 \}}\ ]\!] = (\text{list}, \text{pitch}) \\
\mathbf{T}[\![\ \texttt{list_list}\ ]\!] &= \mathbf{T}[\![\ \texttt{numeric_list}\ ]\!] = \mathbf{T}[\![\ \texttt{\{ 1 \}}\ ]\!] \\
&= (\text{list}, (\text{list}, \text{numeric}))
\end{aligned}$$

♮ **Example 28:** lists type errors

```
error_list1   = { 1, true };          // this is an error, because the entries
                                      // have different types
error_list2   = { 1, { 2 } };         // also an error, because entry 1 is
                                      // numeric and entry 2 is a list
```

Type Inference

To minimize the debugging time for the programmer, it has been deemed important to provide precise type error information before the interpretation stage begins. This advocates a static type system. Static type systems often requires explicit type declarations before using a variable, but since musicians often work very intuitively, type declarations in DIMHCS seem unnecessary.

Instead, DIMHCS employs an automatic *type inference* system, thus avoiding the need for explicit type declarations.

The parser recognizes the type of literals. From the literals, the type of expressions without identifiers can be inferred, as each operator in an expression returns a specific type based on the type of the operands (see Section 2.3 on page 41.

The type of identifiers is determined the first time they are used. There are 2 possible situations in which an identifier can occur for the first time:

- class definitions
- assignments

In the case of class definitions, one of the keywords `block`, `instrument`, or `dsp` is present, defining the type of a class identifier. Example 29 shows class definitions in use.

♮ **Example 29:** class definitions

```
block      my_block { }
instrument my_instrument { }
dsp        my_dsp { }
```

The types of the 3 classes are:

$$\begin{aligned} \mathbf{T}[\![\ \texttt{my\_block}\ ]\!] &= \text{block} \\ \mathbf{T}[\![\ \texttt{my\_instrument}\ ]\!] &= \text{instrument} \\ \mathbf{T}[\![\ \texttt{my\_dsp}\ ]\!] &= \text{dsp} \end{aligned}$$

In the case of assignments, the new identifier is simply assigned the type of the expression occuring on the right-hand side of the equal sign. When an identifier is assigned a type, the identifier retains that type until it is out of scope (see Section 3.2 on the next page). If an identifier which already has been assigned a type is assigned to an expression of a different type, a type error is issued. This concept is demonstrated in Example 30 on the following page.

♮ **Example 30:**

```
pitch_identifier = C3;  // the expression 'C3' has pitch type, and
                        // 'pitch_identifier' is therefore a pitch identifier

numeric_identifier = pitch_identifier:note;
                // the ':note' operator takes a pitch operand and returns a
                // numeric type value. 'numeric_identifier' is therefore a numeric
                // identifier

pitch_identifier = numeric_identifier;  // this statement throws a type error, because
                                        // 'pitch_identifier' and 'numeric_identifier'
                                        // have different types
```

The type inference of these 3 assignment statements can be written as follows:

$$\begin{aligned} \mathbf{T}[\![\ \texttt{pitch\_identifier}\ ]\!] &= \mathbf{T}[\![\ \texttt{C3}\ ]\!] = \text{pitch} \\ \mathbf{T}[\![\ \texttt{numeric\_identifier}\ ]\!] &= \mathbf{T}[\![\ \texttt{pitch\_identifier:note}\ ]\!] \\ &= \text{numeric} \\ \mathbf{T}[\![\ \texttt{pitch\_identifier}\ ]\!] &\neq \mathbf{T}[\![\ \texttt{numeric\_identifier}\ ]\!] \end{aligned}$$

3.2 Scope Rules

The scope rules for the language are simple:

- all variable identifiers only have scope inside the class definition in which they are declared. There are no global variable identifiers.
- class identifiers all have global scope.
- wave identifiers have local object scope, but can be accessed using the accessor operator, ':'.

The rules are illustrated by Example 31 on the next page.

♮ **Example 31:** Scope Rules

```
block main // block identifier (global)
{
    // implicit declaration of dsp variable 'd' references 'delay',
    // which has global scope
    d = delay;

    // declaration of wave variable w, references wave declared inside
    // piano definition, using the accessor operator
    w = piano:w1;
}

dsp delay // dsp identifier (global)
{
    // declaration of numeric value does not interfere with 'd' in
    // 'main' block
    d = 100;
}

instrument piano // instrument identifier (global)
{
    // declaration of wave identifier (local)
    addwave w1 "piano1.wav";
}
```

Chapter III

Implementation

Section 1 explains the choices of tools used for creating the DIMHCS interpreter. Section 1.1 documents the choice of C++ as implementation language. Section 1.2 documents the choice of SableCC[13] as compiler-compiler tool.

Section 2 explains the structure of the DIMHCS interpreter. The object-oriented design of the interpreter is presented in Section 2.1. The *syntactic analysis* phase is documented in Section 2.2. The creation of *default environments* is described in Section 2.3. The *contextual analysis* phase is divided into two subphases, the *object identification* subphase which is explained in Section 2.4, and the *scope and type checking* subphase which is covered in Section 2.5. The *interpretation* phase is described in Section 2.6. Finally, rendering of WAV-files is briefly summarized in Section 2.7.

Section 3 documents the error handling capabilities of the DIMHCS interpreter.

1 Tools

1.1 Implementation Language

The choice of language for implementing the interpreter is based on the popular programming paradigms of the time of writing and the requirements for the interpreter itself.

The current trend in interpreter design favours using a object-oriented language, such as Java, C++, C#[7] or Python[9] as implementation language. This also benefits the selection of the language creation tools, as the latest tools are written for the object-oriented languages.

The choice between the object-oriented languages were in part based on

a single but very important requirement for the interpreter: **speed**. Java was discarded on this account, because of the overhead of interpretation, garbage collection and array index checking. Microsofts C# wasn't practically available to the programmer at the time of writing. Python was also considered, but the available language creation tools for Python were not deemed adequate for this purpose.

C++ was selected due to having the appropriate language creation tools and the the possibility of compiling the DIMHCS system into an executable file with sufficient execution speed (see Section 3 on page 21).

1.2 Compiler-Compiler

As the implementation language was selected, the choice stood between 2 C++-compatible compiler-compiler tools, *lex / yacc* and SableCC. Lex and yacc output C code, which of course isn't object-oriented. The output of SableCC can be C++[6], which is the reason why it was selected.

2 Interpreter Structure

2.1 Object-Oriented Design

The implementation of the DIMHCS interpreter is divided into 3 main components: the parser generated by SableCC, an *AST analysis component*, and a *language component*.

Generated Parser Component

The parser is separated from the other components by use of Étienne Gagnons *Extended Visitor Design Pattern*[13]. It allows the rest of the system to *traverse* the AST generated by the parser *depth-first* by inheriting from the `DepthFirstAdapter` class. Thus, the actual traversal algorithm is hidden from the language implementor.

AST Analysis Component

Figure 3 on page 55 shows the AST analysis component of the system.

The class `ASTAnalysis` inherits from the generated `DepthFirstAdapter`. `ASTAnalysis` adds the functionality for using decoration tables and identification tables from within the analysis classes (see below). The classes `ObjectIdentification`, `ContextualAnalysis` and `Interpreter` inherit from `ASTAnalysis`, thus giving them the tree traversal functionality.

These 3 classes are responsible for the object identification, scope and type checking and interpretation, respectively. Each of the classes has a corresponding `Exception` class, which is *thrown* in the case of errors.

The `Renderer` class handles the actual rendering of waveforms, by using the `Interpreter` to evaluate blocks and VI's.

The `ContextualAnalysis` class use the `DefaultEnvironment` class to set up the default global environment as well as default environments for the blocks, VI's and DSP's.

The utility class `TreePrinter` adds functionality to print an ASCII representation of an AST, much like the one in Example 4 on page 26. It inherits from `ReverseDepthFirstAdapter`, which is generated by SableCC. It has the same functionality as `DepthFirstAdapter`, only it traverses the AST from right to left. The `DecoratedTreePrinter` class prints a tree of the same type as `TreePrinter`, only the type of each node of the AST is printed as well.

Language Component

Figure 4 shows the language component of the system.

`DecorationTable` is used to store information associated with each node of the AST. It employs a hash mapped list, using a function of the memory address of nodes in the AST as hash function. The actual information is stored in the entries of the decoration table, instances of the `Decoration` class. It stores type and value information.

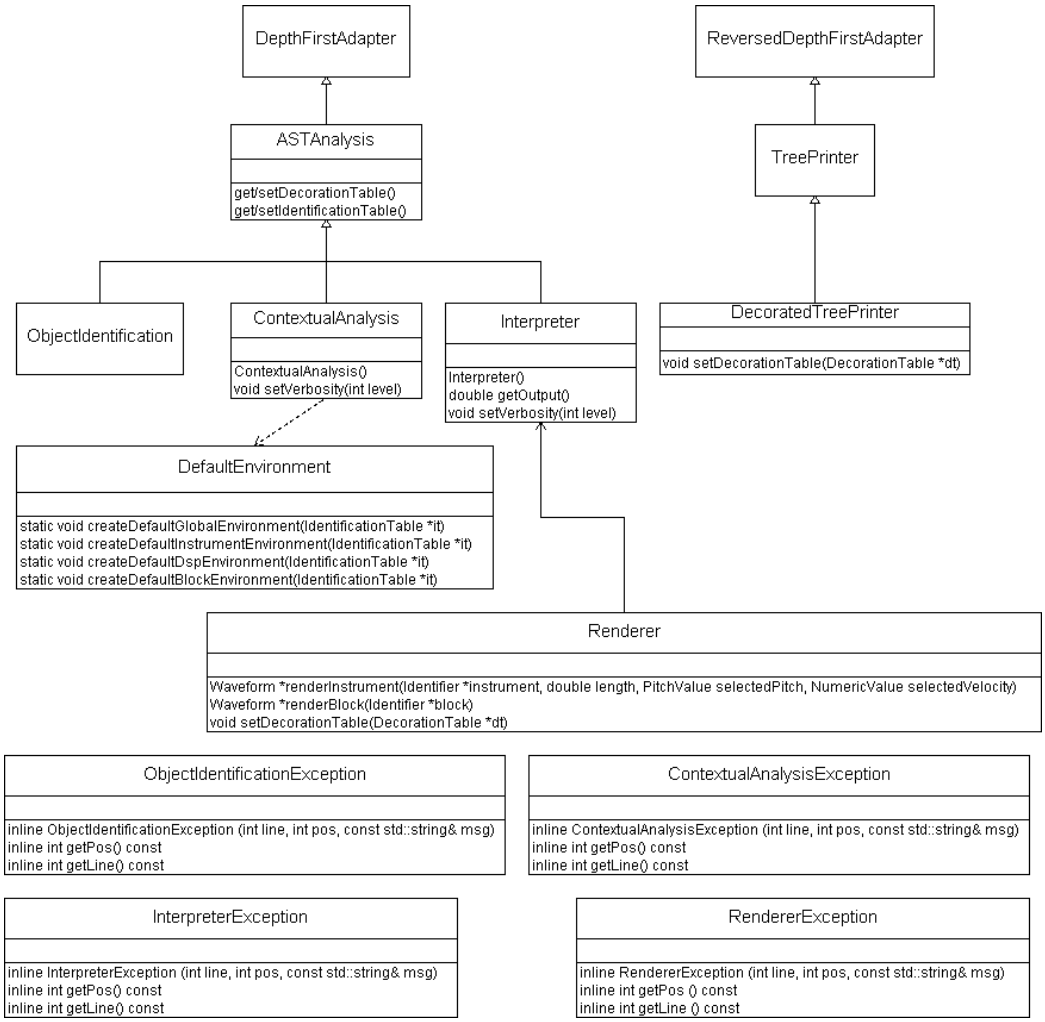
`IdentificationTable` is used for storing similar information as `DecorationTable`, only the information is associated with the names of identifiers. It is implemented similarly to `DecorationTable`, using a hash function of the identifier name string. The entries of the identification table are of class `Identifier`, and they store type and value information, as well as AST node and scope information.

Type information is stored in instances of the `Type` class, which contain the type information described formally in the Type System section on page 47.

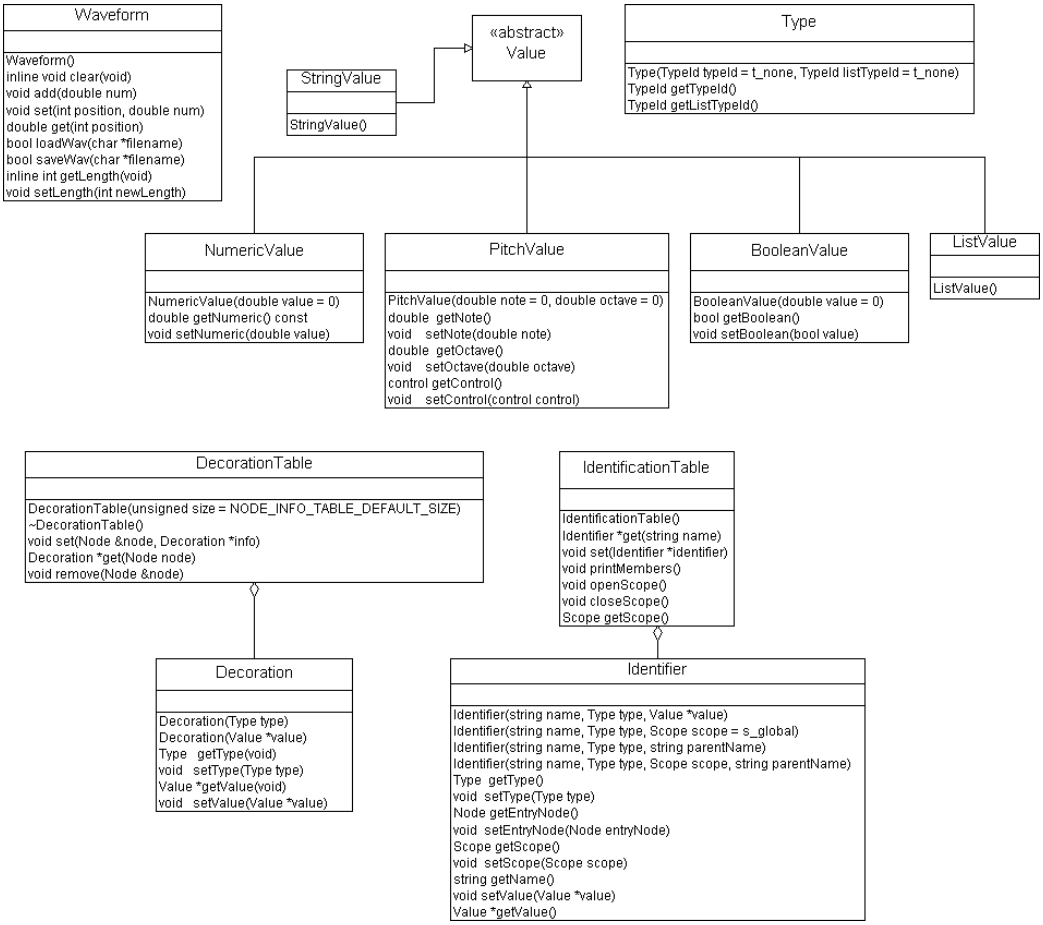
◇ **Implementation Status Note:** *Storage of the recursive List types was not fully implemented in time for the report deadline.*

The `Value` abstract class is used indirectly for storing value information in `DecorationTable` and `IdentificationTable`. The `NumericValue`, `PitchValue`, `BooleanValue`, `StringValue` and `ListValue` classes inherit from this base class. They are used polymorphically by methods in

♭ **Figure 3:** Class Diagram: AST Analysis Component



♭ **Figure 4:** Class Diagram: Language Component



`DecorationTable` and `IdentificationTable`. In C++, polymorphism is used by dynamic casting of references to class instances with a common base class using the `dynamic_cast<>()` operator.

◇ **Implementation Status Note:** *The StringValue and the ListValue classes were not fully implemented in time for the report deadline.*

Furthermore, a utility class called `Waveform` is implemented. It has functionality for loading and saving WAV files and converting them to numeric lists.

2.2 Syntactic Analysis

Syntactic analysis is provided by two mechanisms, a *lexer* and a *parser*.

The lexer reads *tokens* from the input source file, and recognizes terminal symbols. It is defined in the `Lexer` class, which is generated automatically by SableCC. If it reads a token that isn't specified in the grammar, it issues an error.

The parser recognizes the phrase structure of the output of the lexer, and determines if it is syntactically valid. It is defined in the `Parser` class. The output of the parser is an AST.

SableCC generates a parser that recognizes the *Look-Ahead LR(1)* (LALR(1)) class of languages. The LALR language class is a subset of the *LR* language class. The name LR is derived from the fact that LR parsers read their input from **L**eft to right and produce **R**ightmost derivations. The (1) denotes that the parser "looks ahead" on 1 extra input symbol in the process of recognizing a production. The parsing algorithm is an implementation of a *Deterministic Finite Automaton* (DFA). [11][13].

Lexing and parsing is used by the main compiler function simply by creating an instance of the `Lexer` class and supplying it a reference to a file input buffer. Then an instance of the `Parser` class is created with a reference to the Lexer instance. Finally, the parser is activated, returning the root node of an AST.

The parser generates classes for traversing the AST. Example 32 on the next page shows a few methods of the `DepthFirstAdapter` class.

♮ Example 32:

The production:

```
expression = {multiplicative} multiplicative_expression                  |
             {add}            expression plus multiplicative_expression  |
             {sub}            expression minus multiplicative_expression ;
```

results in SableCC generating the following methods in `DepthFirstAdapter`:

```
virtual void inAMultiplicativeExpression   (AMultiplicativeExpression node);
virtual void caseAMultiplicativeExpression (AMultiplicativeExpression node);
virtual void outAMultiplicativeExpression  (AMultiplicativeExpression node);
virtual void inAAddExpression   (AAddExpression node);
virtual void caseAAddExpression (AAddExpression node);
virtual void outAAddExpression  (AAddExpression node);
virtual void inASubExpression   (ASubExpression node);
virtual void caseASubExpression (ASubExpression node);
virtual void outASubExpression  (ASubExpression node);
```

For example, when interpreting a addition expression in a program, the method `inAAddExpression()` will be called, then the parsing method `caseAAddExpression()` will be called, and on completing that, the `outAAddExpression()` method will be called.

2.3 Default Environment

The next step is to create the default global environment. This defines a few default identifiers in the identification table.

At this point, a global identification table has already been created by instantiating the `IdentificationTable` class. In this, all identifiers can be stored, together with their type, value, scope and some extra data. The default environment is simply created in that identification table with global scope by the static method `DefaultEnvironment::createDefaultGlobalEnvironment()`.

2.4 Object Identification

The DIMHCS language design influenced the structure of the interpreter. Because the class type definitions can be referenced before they are defined, an extra pass over the generated AST was required. This pass is considered a subphase of the *contextual analysis* phase and is named *object identification*.

The object identification is the first subphase of the contextual analysis

phase, as well as the first complete traversal of the AST generated by the syntactic analysis phase. It enters global identifiers into the global identification table.

The object identification performs a simple depth-first traversal of the AST. An `ObjectIdentification` instance is created for this purpose. The `ObjectIdentification` class inherits from the `ASTAnalysis` class, which has the tree-traversal functionality. The `ObjectIdentification` methods enters global identifiers (and waveforms) and a reference to their node in the AST into the identification table.

2.5 Scope and Type Checking

The second subphase of the contextual analysis phase is the *scope and type checking*. The scope and type checking is performed after the object identification, checking scope of identifiers and inferring types of all nodes in the AST as well as all local identifiers.

◇ **Implementation Status Note:** *The scope and type checking class is mislabeled ContextualAnalysis. It should have been named ScopeAndTypeChecking, but this wasn't corrected in time for the report deadline.*

This second depth-first traversal of the AST is handled by the class `ContextualAnalysis`, which is instantiated for this purpose. The `ContextualAnalysis` class has the same heritage as the `ObjectIdentification` class, which gives it the same tree-traversal functionality. This subphase makes use of the decoration table, contained in the `DecorationTable` class, which makes it possible to add extra information to a node. This is used for inferring the type of expressions, which again can be used for inferring the type of identifiers. All type information is stored in instances of the `Type` class.

Scope checking is performed simply by marking all local identifiers in the identification table as having local scope, and when the scope is left, all identifiers in the identifiers table with local scope are removed.

2.6 Interpretation

The interpretation phase is activated by an instance of the `Renderer` class. Interpretation does not entail a single, complete pass over the AST, but rather a large number of selective passes over nodes in the AST corresponding

to VI's, DSP's and blocks.

Interpretation is handled by an instance of the `Interpreter` class. For tree-traversal functionality, this class is inherited from `ASTAnalysis` like `ContextualAnalysis` and `ObjectIdentification`. It uses type information stored in the decoration table to handle calculation and storage of data.

Runtime Organization

Each data type is represented by a corresponding class inherited from the `Value` class. These are `BooleanValue`, `NumericValue`, `PitchValue` and `ListValue`. When an identifier is created in the program, a Value subclass instance and a Type instance is allocated by the interpreter and these are entered into the identification table.

2.7 Rendering

The rendering phase is handled by the `Renderer` class.

It was designed to render the block named `main`, recursively rendering all other blocks and the VI instances. Unfortunately, implementation of this phase was not completed before the deadline of the report. To make testing of the interpretation phase possible, a simple version of the `renderInstrument()` method was implemented. This method renders a single instance of the VI named "test", using values for `length`, `selected_pitch` and `selected_velocity` supplied as parameters.

3 Error Handling

Each phase and subphase of the DIMHCS system has an exception class associated with it. Table 8 on the facing page lists the exceptions for each phase and subphase. Note that though lexing and parsing strictly isn't subphases due to their execution being interleaved, they are still listed as such for the sake of simplicity. These classes are used for error handling and error message output.

♯ **Table 8:** Exception Classes

Phase / Subphase	Exception Class
Lexing	`LexerException`
Parsing	`ParserException`
Object Identification	`ObjectIdentificationException`
Scope and Type Checking	`ContextualAnalysisException`
Interpretation	`InterpreterException`
Renderering	`RendererException`

◇ **Implementation Status Note:** *The RendererException is unused in the implementation at the time of writing.*

The exceptions can be used to print the line and character number of the token estimated to be the first source of the error. Examples 33, 34, 35, 36, 37, and 38 show the error handling.

♮ **Example 33:** Lexer Exception

```
block main
{
    p = Cx; // Cx is not a valid pitch literal
}
```

DIMHCS interpreter output:

```
PARSING ==========================================================
lexer error in line 3 pos 9: [3,9] Unknown token: Cx;
```

♮ **Example 34:** Parser Exception

```
block main
{
    if(2 == ) y = 2;;
}
```

DIMHCS interpreter output:

```
PARSING ==========================================================
general parser exception: [3,13] expecting: '-', '+', '(', '{', '[', string, '?',
'sin', 'cos', score control literal, boolean literal, pitch literal, num literal,
identifier
```

Note that the default parser exception outputs all the valid tokens that *could* have been placed where the error occurred

♮ **Example 35:** Object Identification Exception

```
block main { }
block main { }
```

DIMHCS interpreter output:

```
PARSING ============================================================
OBJECT IDENTIFICATION ==============================================
object identification error in line 2 pos 7: identifier 'main' defined twice
```

♮ **Example 36:** Scope and Type Checking Exception: Scope Error

```
block main
{
    matrix 1/8
    {
        select [ bleep ]
    }
}
```

DIMHCS interpreter output:

```
PARSING ============================================================
OBJECT IDENTIFICATION ==============================================
CONTEXTUAL ANALYSIS ================================================
contextual analysis error in line 5 pos 18: unknown identifier: bleep
```

♮ **Example 37:** Scope and Type Checking Exception: Type Error

```
block main
{
    p = C3;
    n = 2;
    x = p * n;
}
```

DIMHCS interpreter output:

```
PARSING ============================================================
OBJECT IDENTIFICATION ==============================================
CONTEXTUAL ANALYSIS ================================================
contextual analysis error in line 5 pos 11: incompatible types for multiplication
```

♮ **Example 38:** Scope and Type Checking Exception: Type Error

```
block main
{
    p = C3;
    p = { C3 };
}
```

DIMHCS interpreter output:

```
PARSING ============================================================
OBJECT IDENTIFICATION ==============================================
CONTEXTUAL ANALYSIS ================================================
contextual analysis error in line 4 pos 7: incompatible types for assignment
```

This error shows that the type inference and subsequent checking works correctly; an identifier cannot have 2 different types in the same scope.

◇ **Implementation Status Note:** *Note that though the errors in the examples listed are detected correctly, the error handling isn't finished at the time of writing, so some tests of error handling may yield unexpected results.*

Chapter IV

Evaluation

Section 1 evalutes the choice of C++ as implementation language for the interpreter. Section 2 evalutes the DIMHCS system.

1 Implementation Language Evaluation

In retrospect, C++ was an appropriate implementation language, yet not perfect for the task. The SableCC output was adapted to Java output, and Java has easier access to polymorphism and references. The implementation in C++ is very similar to that of a Java implementation, but the code would look more complex due to casting and reference operators. This difference is demonstrated by Example 39 on the next page

♮ Example 39:

This example shows an excerpt of the implementation of a trivial interpreter method, `outAAddExpression()`, which computes the result of arithmetic addition. First, the C++ version:

```
void Interpreter::outAAddExpression(AAddExpression node)
{
    // get value from subexpressions
    Decoration *d1 = dt->get(node.getExpression());
    Decoration *d2 = dt->get(node.getMultiplicativeExpression());
    Decoration *result = dt->get(node);

    // numeric addition
    if(result->getType() == t_num)
    {
        double v;
        v = dynamic_cast<NumericValue *>(d1->getValue())->getNumeric()
             + dynamic_cast<NumericValue *>(d2->getValue())->getNumeric();
        result->setValue(new NumericValue(v));
    }
}
```

Then the same method in Java. It is obviously a bit easier to write, which means fewer errors and shorter development time:

```
void outAAddExpression(AAddExpression node)
{
    // get value from subexpressions
    Decoration d1     = dt.get(node.getExpression());
    Decoration d2     = dt.get(node.getMultiplicativeExpression());
    Decoration result = dt.get(node);

    // numeric addition
    if(result.getType() == T_NUM)
    {
        double v;
        v = d1.getValue().getNumeric() + d2.getValue().getNumeric();
        result.setValue(new NumericValue(v));
    }
}
```

2 System Evaluation

At the time of writing, the system is unfinished and cannot be fully tested. However, the syntactic and contextual analysis phases are functional. The error handling for these phases also works very well. The system can render a single VI instance and store the output in a WAV file. The output sounds as expected.

The system was succesfully compiled and tested on 2 different machines:

Name:	**huba**
Hardware Architechture:	233 MHz Pentium MMX
Operating System:	Gentoo Linux 2.6.5-r1
Name:	**homer**
Hardware Architechture:	2.80 GHz Intel Xeon
Operating System:	Linux 2.4.20-31.9smp

The test included speed of interpretation using the following DIMHCS program:

```
instrument test
{
init:
    freq = 27.5 * 2^(1/4) * 2 ^ (selected_pitch:note / 12 + selected_pitch:octave);

run:
    output sin(lifetime * freq * (2 * pi) / 44100);
}
```

The premature version of the system renders the "test" instrument in a time frame of 10000 samples. The test results are shown in Table 9:

♯ **Table 9:** Test Times

Machine	Interpretation Time
huba	5 m 20 s
homer	23 s

Bibliography

[1] David A. Watt & Deryck F. Brown. *Programming Language Processors in Java: Compilers and Interpreters*. Prentice-Hall, 2000. ISBN: 0-130-25786-9.

[2] Emagic Soft- und Hardware GmbH. Logic audio, 2004. http://www.emagic.de/products/index.php?lang=EN.

[3] IRCAM. Max, 2003. http://www.ircam.fr/produits/logiciels/max-e.html.

[4] ISO. ISO/IEC 14977:1996 – information technology – syntactic metalanguage – Extended BNF, 1996.

[5] MIT Media Laboratory Machine Listening Group. MPEG-4 Structured Audio, 2000. http://web.media.mit.edu/~eds/mpeg4.

[6] Indrek Mandre. Alternative output sablecc, 2004. http://www.mare.ee/indrek/sablecc.

[7] Microsoft. C ♯, 2004. http://msdn.microsoft.com/vcsharp.

[8] MIDI Manufactorers Association. Complete MIDI 1.0 detailed specification, 1996. http://www.midi.org/about-midi/specinfo.shtml.

[9] Python Software Foundation. Python, 2004. http://www.python.org.

[10] Curtis Roads. *The Computer Music Tutorial*. MIT Press, Cambridge, Mass., 1996. ISBN: 0-252-18158-4.

[11] Various sources. Wikipedia, the free encyclopedia, 2004. http://www.wikipedia.org.

[12] Unknown Author. Tutorials for midi users, 2004. http://www.borg.com/~jglatt/tutr/miditutr.htm.

[13] Étienne Gagnon. Sablecc, an object-oriented compiler framework. Master's thesis, School of Computer Science McGill University, Montreal, March 1998. http://www.sablecc.org.