μ: A Functional Programming Language for Digital Signal Processing

Matthew Gordon

University of New Brunswick

9 April 2003

CS4997 Undergraduate Thesis
Supervisor: Prof. Brad Nickerson

Contents

List of Figures

1 Introduction

This document describes my undergraduate thesis work at the University of New Brunswick between September of 2002 and April of 2003. During this time I designed a programming language and implemented a compiler for that language. The language is called μ (pronounced 'moo') and was designed primarily with digital signal processing in mind. After designing the language and implementing the compiler, I wrote a number of simple μ programs and evaluated the language based on my experience with these programs.

The field of digital signal processing (DSP for short) involves the creation of signal filters. A filter is anything which a signal passes through which may change the signal. A good introduction to this field can be found in [13]. DSP has a wide range of applications—just about every technology that involves signals involves DSP: telephones, radios, televisions, computer networks and many others. However μ was designed primarily for processing a certain kind of signal: audio signals.

2 Motivation and Inspiration

2.1 Audio Engineering and Filter Plugins

Audio engineering is the task of taking one or more audio sources—instruments, singing or speaking people, or anything else that makes a noise—and reproducing it so that it sounds as good as possible. This is accomplished by applying various filters to the audio sources in order to remove unwanted artefacts or make the sounds more pleasant to listen to. The "sound guy" that runs the mixing board when a band is playing live is an audio engineer, and audio engineering is also a very important part of creating sound for movies, radio, videos and compact discs.

2.1.1 Hard Disk Recording Software

Recording studios traditionally have used reel-to-reel analog tape machines to record music. These machines support multiple tracks of high-quality audio, allowing each part of the music (e.g. each instrument and singer in a band) to be recorded separately on it's own track. After everything has been recorded, filters can be applied to individual tracks or groups of tracks, and finally a mixer is used to mix all the tracks together and produce a stereo output to record onto CDs, Tapes or DVD. This allows the engineer to experiment with using different filters with the different tracks and try different ways of mixing the tracks together.

Recording studios are increasingly moving towards hard disk recording software such as Pro Tools. Instead of recording to analog tape, the tracks are recorded digitally onto computer hard disk. The mixing and processing (filtering) can then be done using software, although hardware filters are still often used.

2.1.2 Filter Plugins

The audio signal filters used with recording software typically take the form of plug-ins—software modules which dynamically link with ("plug into") the recording software using one of a number of standardized interfaces. For example, if a user is unhappy with the equalizer that is included with their recording software they may buy a third-party equalizer plug-in in a format that their software recognizes. After they have installed the new equalizer it becomes available to them as just another part of the recording package.

The filters used in recording music fall mostly into a small number of categories; equalizers, delays, compressors and expanders are the most common. Different examples of a particular type of filter vary with respect to the details of their exact capabilities and interface, along with the "character" that they impart to the sound. Good plug-ins can cost quite a bit of money. One of my goals in this thesis was to work towards an easy way for users to experiment with making their own filter plug-ins.

2.2 Music Synthesizers

2.2.1 Virtual Instrument Plug-ins

Another kind of plug-in that is often used with recording software is a virtual instrument or synthesizer. These plug-ins allow the software to create music tracks without recording an actual instrument—the software "synthesizes" a sound which resembles an instrument. This can be either a purely synthetic musical sound or a simulation of a real instrument. Another goal for this thesis was to work towards creating an easy way for users to create their own synthesizers. Many plug-ins already exist that allow the user to design their own synthesizer sounds; however, these are typically modelled after analog synthesizers and do not take full advantage of the extra possibilities allowed by the use of the computer.

2.2.2 CSound

One program which is designed specifically for synthesizing music on a computer is CSound, a free program available at [3]. CSound reads in text files following a specified format and generates sound. Although CSound is a very powerful and flexible program, I found that the syntax of the input files was quite difficult to learn and was not very readable. I felt that I could improve on this design.

2.3 Solution: Special-Purpose Programming Language

I decided that for my thesis I would design and implement a simple computer language designed specifically for creating audio filters and synthesizers. I decided that a full-fledged system for creating plug-ins was too large a project for the time available and instead focussed on creating a compiler that could evaluate the design of the language. I decided to create a compiler that

generates programs that can read and write wave-format files—this would allow me to evaluate the language based on how easy it is to create a variety of filters and synthesizers with it. If the language turned out to be a good design then a future project would be expanding it into a full-featured system for creating plug-ins. For no particular reason, I decided to call my new language μ.

3 Design of the μ Language

3.1 Design Goals

μ was designed to meat the following criteria:

1. The common types of filters (equalizers, compressors, expanders, delay effects, modulators) should be easy to implement in μ.
2. μ should not be limited to standard filter designs.
3. People with a solid math background but no programming experience should be able to learn μ easily.
4. μ should be very expressive mathematically. It should be possible to concisely write complex expressions.
5. Things like input, output and memory management should be taken care of automatically by the compiler.

3.2 Characteristics of μ

I decided that the μ language should be a first-order pure functional language based on standard mathematical notation. The following paragraphs explain what exactly is meant by this and how it helps meet the design goals.

3.2.1 Pure Functional

μ is a functional programming language. For a complete discussion what functional languages are and how they are different from other types of languages, I would recommend reading [9]. By calling μ a first-order pure functional language I mean that it supports equational reasoning but not higher-order functions [1, page 298]. Equational reasoning is discussed in section 3.2.3 below.

3.2.2 Modelled After Standard Mathematical Notation

One of the design goals was to make a language that was easy for mathematically-inclined non-programmers to learn. To meet this goal, the μ grammar was modelled after standard mathematical notation. An example of this is shown

in figure 1. On the left is a mathematical function definition and on the right is the equivalent μ function definition.

$$f(x) = \begin{cases} sin(x)/x & \text{for } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

```
f(x) = sin(x)/x for x > 0
     | 0        otherwise
```

Figure 1: A mathematical expression and the equivalent expression in the μ language.

3.2.3 Equational Reasoning

From a semantic standpoint, one important way that μ differs from common imperative languages such as C is that it uses equational reasoning. As with all pure functional languages, functions in μ never have side effects. Variables are also not supported in the traditional imperative sense: once a value is bound to a variable, the value of the variable cannot change. The practical effect of this is that if the statement `y = f(x)` occurs at one point in a program and the statement `z = f(x)` occurs at another point in the program, then it is guaranteed that $z = y$. This is how things normally work in mathematics but it is not how they work in imperative languages. In an imperative language, the value of x could have changed, causing z to be different from y. Also, the function f could have side effects; for example, f could contain a static variable which affects the return value and is changed when f is called. For an example of this, see figure 2.

```
float f( float x ) {
    static float i = 0;
    i += 1;
    return x + i;
}
```

Figure 2: A function with side effects, written in C.

The most important thing that equational reasoning does for μ is it puts a layer of abstraction on top of the von Neumann architecture. Functions and variables behave the way they are supposed to in mathematics, not the way they are supposed to on a von Neumann machine. This is intended to simplify the translation of filters from mathematical or electrical notation into μ. It allows μ to more closely model not only abstract mathematical functions, but also electrical circuits, which are typically described by systems of equations.

3.2.4 "Everything is a Function"

In μ, "everything is a function." Functions can be defined by the programmer using expressions, μ includes some built-in functions, and sound files are treated as functions. After all, a sound is just a function of air pressure versus time.

At the beginning of a μ program, input and output files are bound to function names. The input files can then be read by "evaluating" their functions—the i^{th} sample of the file bound to "`aFile`" is returned by "`aFile(i)`". Similarly, the output file is created by defining it's function.

3.2.5 Flow of Control

A μ program does not contain sequences of statements which are executed in order as with an imperative language. Rather, μ programs consist of a collection of function definitions, one or more of which must be bound to output files. Each output function is evaluated once for each sample in it's output file. For example, if the function $f(x)$ is bound to the output file "output.wav" which is 1,000,000 samples long, then $f(x)$ is evaluated once for each possible integer value of x between 0 and 999,999. Functions in μ support any *rational numbers* as parameters, but the output is sampled for each *integer* value.

The output function(s) may be defined in terms of other functions, in which case the other functions are evaluated as needed. Lazy evaluation is used—functions are only evaluated if they are needed for the output and only over the range needed for the output.

3.3 Design Specifics

This section describes in detail the specifics of the μ syntax and semantics. The complete grammar specification can be found in appendix A.

3.3.1 Program structure

Each μ program consists of two parts. The first part is the pragma section and the second is the definitions section. Both sections must be present in all programs.

The pragma section is intended to contain all declarations concerning i/o and data types. It must contain at least one `#output` pragma and may also contain `#input` or `#include` pragmas. All pragmas start with the "`#`" character. The `#include` pragma causes the contents of the specified file to be added to the definitions section. The `#input` and `#output` pragmas are described below.

3.3.2 Binding Sound Files To Functions

Sound files are bound to functions using `#input` and `#output` pragmas. An example is shown in figure 3. In the example, the file "input.wav" is bound to the function `f_in`; `f_in(i)` now refers to the value of the i^{th} sample from "input.wav". The file "output.wav" will be created using integer values 0–999,999 of `f_out`.

```
#input f_in from "input.wav"
#output f_out to "output.wav" length 1000000 datatype int16
```

Figure 3: `#input` and `#output` pragmas

3.3.3 Defining Functions

Function_name (*Parameter*) = *Expression*

Figure 4: Function definition syntax

Figure 4 shows the syntax for a function definition. *Function_name* is the name of the function and *Parameter* is the name of it's parameter. The parameter name can be any valid identifier that is not used for a function or constant name; more than one function can use the same parameter name. *Expression* is a mathematical expression which evaluates to the value of the function in terms of the parameter.

3.3.4 Basic Expression Syntax and Operators

Expressions involving literal numbers, named constants, the parameter and the `+`, `-`, `*` and `/` operators are written just as they would be in C. In addition, μ contains a summation operator, the `%` operator has a different meaning than in other languages, and μ has it's own form of conditional.

3.3.5 Other Operators

`SUM(` *Iterator* = *LowExpression* `to` *HighExpression* `)(` *Expression* `)`

Figure 5: Summation syntax

The `SUM` operator in μ performs mathematical summation similar to Σ-notation. The syntax is shown in figure 5—*Expression* is evaluated for each integer value of *Iterator* between the values *LowExpression* and *HighExpression*. All the values of *Expression* are summed up to get the result. Figure 6 shows an example μ function definition and equivalent math expression, demonstrating the `SUM` operator.

```
f(x) = SUM( i = 1 to x)(i*x)
```

$$f(x) = \sum_{i-1}^{x} ix$$

Figure 6: μ function and equivalent math expression demonstrating summation.

The "%" operator is more flexible in μ than a simple modulus operator. When used with integer values, it returns the modulus, i.e., `a % b` returns

the remainder of integer division a/b. However, in μ it has been extended to operate on all rational numbers. The definition is:

$$x = a\%b \leftrightarrow (\exists y \in \mathrm{Z} \mid ay + x = b\ ) \ \wedge \ (\neg\exists z \in \mathrm{Z} \mid az + x = b \wedge z > y) \quad (1)$$

Z is the set of integers. This looks like the definition for modulus, but in μ it applies for $\forall x, a, b \in$R. This is useful for creating waves from wavelets because as x varies from 0 to ∞, $x\%n$ loops continually over the range $[0, n]$.

3.3.6 Conditionals

Function_Name `(` *Parameter* `)` `=` *Expression1* `for` *Condition1*
| *Expression2* `for` *Condition2*
| *Expression3* `for` *Condition3*
.
.
.
| *OtherwiseExpression* `otherwise`

Figure 7: Syntax for a conditional function

The only conditional form in μ is the conditional function form. The syntax for a conditional function is shown in figure 7. Each *Condition* specifies a range of values for the parameter. When the function is evaluated, the *Expression* corresponding to the first *Condition* that matches is used to calculate the return value. If no *Condition* matches, then *OtherwiseExpression* is used. Each conditional function must have exactly one *OtherwiseExpression*. An example of a conditional function with equivalent math expression is given in figure 8.

```
f(x) = x  for x < 10
     | 10 otherwise
```

$$f(x) = \begin{cases} x & \text{for } x < 10 \\ 10 & \text{otherwise} \end{cases}$$

Figure 8: A conditional function and mathematical equivalent

3.3.7 Function Composition

Of course it does no good to be able to define all sorts of functions if they cannot be combined in some way and influence the output. Other functions can be referenced inside a function's definition just as they would be in mathematics. A function reference takes the form *Function_name*`(`*Expression*`)`, where *Expression* can be any valid μ expression. Function references are legal anywhere inside an expression where a constant or literal is legal. This includes inside function references, e.g. `h(x) = f(g(x))` is a legal function definition; it means $h(x) = f \circ g(x)$.

3.3.8 Extra Niceties

There are two other features are included in the μ specification to aid the programmer: constants and built-in functions.

A constant is defined much like a function except that there is no parameter for a constant, and the expression in a constant definition cannot contain function references. Constant definitions belong in the definitions section of a μ program.

Built-in functions are just that—functions that can be used in any μ program without being defined by the programmer. They are legal anywhere a function reference is legal. The μ built-in functions are sin, cos, tan, arcsin, arccos, arctan, log and ln. Log and ln are log_{10} and log_e, respectively.

4 Implementation

This section describes the design of muc, the μ compiler that was implemented as part of this thesis.

4.1 Goal of This Implementation

The version of muc that was created for this thesis is an evaluation version intended for testing the effectiveness of the language design. It is not a full-featured compiler system and does not have adequate error checking or optimization of any sort. However, muc was designed and implemented in a manner that should facilitate adding these things later. All language features described in section 3 are implemented in this version of muc except that it only supports one input file and one output file.

4.2 Development Environment

Muc was developed on a Linux PC using the GNU C and C++ compilers, the NEdit text editor, the lex lexical analyser generator, and the yacc compiler generator.

4.3 High-level Program Structure

Muc can be broken down into the following compilation phases: lexical analysis, parsing, semantic actions, intermediate representation and code generation. A typical compiler normally involves more phases than this; some phases that would normally be included in a compiler were deemed unnecessary for muc. Optimization phases were not included, some phases were made unnecessary by the use of C++ instead of assembler as the output language (described in 4.6 below), and some phases are only needed in imperative languages with more complicated control structures. For example, the normal processes of canonicalization and trace generation do not apply because μ does not involve sequences of instructions or jumps.

4.4 Lexing and Parsing

As stated above, lex and yacc were used to aid in the implementation of muc. These tools were selected based primarily on their apparent popularity

and the fact that they are designed to work together. Lex reads a lexical specification from a text file and generates a C source file containing a lexical analyser for the language. For more information on lex, see [4]. Similarly, yacc[5] reads a text file containing a context-free grammar specification with semantic actions and generates a C source file containing an LALR [1, p. 65] parser for the language. The parser generated by yacc can use the lexer generate by lex, which is what was done in this case.

4.5 Intermediate Data Structures

The μ parser generates a number of intermediate data structures which are then used during code generation. These data structures are described below. All global data structures are defined in the file "parse.h" which is listed in appendix B.

4.5.1 Symbol Table

Muc uses a singly-linked list for it's symbol table; the elements of this list are of type `SymTabEnt`. See appendix B for the definition of `SymTabEnt`. `SymTabEnt.name` is the symbol name (identifier) and `SymTabEnt.type` is it's type: `TYPE_CONST`, `TYPE_FUNC` or `TYPE_PARAMETER`. If the symbol refers to a constant then `SymTabEnt.parseTree` points to it's expression tree (see 4.5.4). If the symbol is a function name, then `SymTabEnt.func` points to the function table entry for that function.

4.5.2 Function List

Muc also uses a linked list to store all the function definitions; the elements of this list are `FuncListEnt`s. Each `FuncListEnt` contains a link to the function's name in the symbol table and to the function parameter's entry in the symbol table. `FuncListEnt` also contains a list of functions that are called by the current function. If the function is a conditional function, then the `for` clauses are stored in a list of `ExpressionListEnt`s. `ExpressionListEnt` contains the beginning and end of the range of values that this expression applies to, along with the expression tree itself. `FuncListEnt` contains a separate entry for the `otherwise` clause. This field is also used to store the expression tree for non-conditional functions.

4.5.3 Constant List

Muc keeps a linked list of constants. This list simply contains pointers to the corresponding entries in the symbol tables and allows all constants to be easily found and iterated over.

4.5.4 Expression Trees

The actual mathematical expressions associated with the constants and functions are stored using expression trees. The nodes of these trees belong to the types `EExpression`, `EElement`, `ETerm`, `EFactor`, `EP`, `EAbs`, and `ESum`. Each of these corresponds to a production in the μ grammar; the expression trees reflect the parse trees of μ expressions.

4.6 Code Generation

This section describes the final stage of the compiler—code generation. For an example of some output from the code generator, along with the μ source it was generated from, see appendix D.

4.6.1 Output File

Muc does not generate assembly code. The task of creating a complete compiler system is a large one for an undergraduate thesis; having the compiler generate output in a high level language reduced the scope of the project to manageable proportions. Generating assembler code has the advantage of greatly increased flexibility; the compiler has complete control over all details of the generated executable. This allows for executables which are very well optimized for both space and execution time. The drawback to all this power is the responsibility that comes with it—the compiler must handle register and memory management, the mechanics of function calls, and many other details. Generating output in a high-level language removed the need for muc to take care of these things.

The high-level language which was selected for muc's output is C++. There are four main advantages of C++ over other possibilities. Firstly, C++ is the language with which I am most familiar. Secondly, C++ allows variable declarations to be intermixed with other statements, instead of needing to be declared at the beginning of a file or function. This simplifies the design of the code generator because temporaries can be declared as needed, allowing

the output to be easily generated in a single pass. C++ also allows for the creation of arbitrary nested scopes (using { and }), simplifying the task of variable scope management for muc. Finally, C++ is an international standard which can be compiled on a wide range of platforms.

4.6.2 Conventions Followed in Generated Code

The code generated by muc is spaced and indented in such a way as to make it easily readable. The program is broken into lines as a typically programmer would, despite the fact that C++ does not require statements to be separated by newline characters. The code generator also increases indentation each time scope narrows; function bodies are indented and indentation is increased inside `if` and `for` statements.

Temporaries variables and other names generated by muc always begin with an underscore. Since μ identifiers cannot begin with an underscore, this prevents collisions between names.

4.6.3 Layout of Generated Code

The C++ files generated by muc can be broken down into three sections: declarations, function definitions, and the main function. The main function can further be divided into constant definitions and the main program loop. The file is generated in a single pass from top to bottom.

The first section of the output file contains the constant and function declarations. This section simply contains declarations of all the function and constant names from the μ source. All constants are declared as `float`s and each function is declared as `inline`, returning a `float` and taking a single `float` as it's parameter. Making the functions inline increases the efficiency of the resulting code while still allowing recursion—the C++ compiler ignores the inline specifier on recursive functions. Functions are not defined until the function definition section, and constants are not assigned values until the constant definition section inside `main`.

The second section of the generated files consists of function definitions. Each function definition begins with the variable declaration "`float _retval;`" and ends with the statement "`return _retval;`". For non-conditional functions, the middle part consists simply of evaluating the function expression and storing the result in `_retval`. For conditional functions, an `if...else` statement is used to select the correct expression. For details on expression

evaluation, see section 4.6.4.

The third and final part of the generated code is the main function. The main function begins by assigning values to the constants. This is done by evaluating the expressions from the constant table. (See section 4.6.4 below.) Next, the output audio file is opened using libmu (see section 4.6.5). A for-loop is then used to evaluate the output function for each sample and the results are written out using libmu. Finally, the output audio file is closed.

Once all these sections have been written, the C++ file is closed and is ready for compilation using g++, the GNU C++ Compiler.

4.6.4 Expression Evaluation

Expression evaluation is performed by the muc code generator using the function `printExpr( EExpression *expr, char *temp )`. The source code for this function is shown in figure 9. The `printExpr()` function generates code which evaluates `expr` and stores the result in the C++ variable whose name is stored in `temp`. Each type of node in the expression tree has a similar function associated with it. Each of these functions creates new temporary variables to hold the results of their subexpressions, calls the print routines of the subexpressions, and then combines the results into the variable pointed to by `temp`. The easiest way to understand this process is simply to look at figure 9.

Muc uses the `getTemp()` and `releaseTemp()` functions to manage temporary variables. `getTemp()` returns the name of a new temporary which is valid in the C++ file, `releaseTemp()` "releases" the temporary. The first time a temporary is created, `getTemp()` inserts a variable declaration into the C++ file. If `getTemp()` is called after a temporary has been released, it will return the name of the previously released temporary but will not declare it again. These functions elegantly allow the creation of an indefinite number of temporaries while ensuring that the number of variable declarations in the C++ code is kept to a minimum.

4.6.5 The `libmu` Library

The C++ code generated by muc must be linked with the "libmu" library in order to work. This library contains the file i/o functions used by μ. It consists of six functions. One function opens '.wav' files for reading, another reads a sample from the file, and a third closes the file. The other three

```
void printExpr( EExpression *expr, char* temp )
{
    /*Expression just contains a term; just call printTerm().*/
    if(expr->type==expr_term)
    {
        printTerm(expr->term, temp);
    }
    /*Expression is an input file reference*/
    else if(expr->type==expr_file)
    {
        indent();
        fprintf( outfile, "%s = _readSample((unsigned int)fp);\n", temp );
    }
    /*Expression is addition or subtraction.*/
    else
    {
        /*Create a new temporary variable*/
        char* t1 = getTemp();
        /*Store the first subexpresion in the temp*/
        printExpr( expr->expr, t1 );
        /*Create another temporary*/
        char* t2 = getTemp();
        /*Store the second subexpression in the temporary*/
        printTerm( expr->term, t2);
        indent();
        /*Add or subtract the two temporaries*/
        fprintf( outfile, "%s = %s %c %s;", temp, t1, expr->op, t2 );
        free(t2);
        releaseTemp();
        free(t1);
        releaseTemp();
    }
}
```

Figure 9: The `printExpr()` function from the muc code generator

functions open a file for writing, write a sample, and close the output file. Libmu currently only operates on Microsoft Wave files, as described in [2].

5 Testing

After the implementation of the μ compiler was complete, several programs were written in μ as tests. The purpose of these tests was not just to ensure that the compiler worked properly; they also served to help evaluate the μ language, based primarily on how easy it was to write the programs. For a general discussion of the results, see section 6. The specific tests that were performed are described below.

There were three stages in the testing of muc. First, a version of muc which did not support the `#input` and `#output` pragmas was created. This version simply printed the first 100 integer values of the last function defined. Some simple tests were run with this version to check that expression and function evaluation worked correctly. Next, support for the `#output` pragma was added. This allowed some test synthesizers to be written, as described in section 5.1 below. Finally, `#input` support was added and the test filters described in section 5.2 were created.

5.1 Wav Output Testing With Synthesizers

Doing synthesizer testing before filter testing allows the expression evaluation and wav file output sections to be tested thoroughly before wav file input is implemented. This is a great benefit because problems in the input would ruin the rest of the processing and bugs in output or expression evaluation would make it difficult to diagnose bugs in the input. In short, it allows the muc output to be tested in pieces, rather than all at once.

5.1.1 Additive Synthesis

The first major test program written in μ was a basic Hammond organ simulation using additive synthesis. Additive synthesis is the method of synthesizing a sound by adding together a collection of oscillators. The synthesizer is basically individually generating each harmonic of the note [13].

The Hammond electric organ, in its original form worked by having an electric motor drive a gear. This gear drove other gears which were connected to AC generators; the sound of the organ was the combined output from these generators. The gear ratios were calculated so that the output from the different generators formed a harmonic series. Nine "drawbars" were used by the player to control the relative volume of the different harmonics.

This allowed the player to control the timbre of the organ in much the same way that the timbre of a conventional organ can be controlled by pulling out stops.

The test program created can be found in appendix C.1. The nine drawbar settings, which on an organ can vary between zero and ten, are declared as constants, allowing them to be easily adjusted for different sounds. A "gain" constant is then defined for controlling the final volume. Sine-wave oscillator are used to create each of the harmonics and these are added together to form the output.

This synthesizer was very easy to write in μ and sounds pretty good. Appendix C.6 shows a program which uses this synthesizer to play a tune. The program in C.6 also uses a chorus effect to give the organ a "fuller" sound.

5.1.2 PWM Synthesis

The second synthesizer written in μ is a string pad; a type of synthesizer that is supposed to sound like a string ensemble. Synth string ensembles like this one don't sound at all like real string ensembles, but they are popular instruments in their own right. They typically use PWM synthesis.

PWM stands for Pulse Width Modulation. A "pulse wave" is a wave that is a constant positive value for a units of time, then a constant negative value for b units of time, then a constant positive value for a units of time, etc. The ratio a/b is the *duty cycle* of the pulse wave. A square wave is a pulse wave with a duty cycle of 50%. Figure 10 shows an example of a pulse wave.

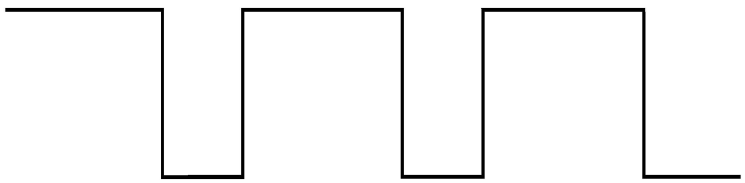

Figure 10: A pulse wave with a duty cycle of 66%.

Pulse width modulation means the duty cycle of the pulse wave is varying over time. PWM synthesis is a common method for creating synthesized

string sounds; for a more complete discussion of PWM and string synthesizers, see [7] and [8].

Appendix C.2 is an implementation of the PWM string pad described in [8]. The design was easy to implement in μ, even though it was created by following instructions written for analog synthesizers. A variety of interesting sounds can be obtained by varying this basic design.

5.2 Wav I/O Testing With Filters

After the synthesizer tests were performed, some filters were written. These tested muc's wav input capability and further tested the capabilities of μ.

The first filter test was an 'allpass' filter. An allpass filter allows all frequencies to pass through unchanged except that it may alter their phase. In this case however, the filter had no effect on the signal—it simply read a file in and created a duplicate of it. This tested the generated code's basic ability to read and write wav files correctly.

5.2.1 Mixers

One basic type of filter is a mixer. A mixer simply reads a number of inputs, adjusts their volumes, and combines the signals together into one output. In μ volume can be adjusted by multiplying the function by a constant and signals can be combined by adding functions together.

The implemented version of muc does not support multiple input files. It can read stereo files—a stereo wav file contains the samples from the two samples intermixed: left, right, left, right, etc. Muc simply reads stereo files just like they were mono. This does not conform to the μ specification but it does allow us to easily create a test program which performs a stereo mixdown; that is, combine the two channels of a stereo file into a single mono file. The code for this can be found in appendix C.3.

5.2.2 Chorus

Another test filter that was written is a chorus effect. Chorus effects make the sound of an instrument sound like multiple instruments using a variable delay. The original signal is combined with a slightly delayed version of itself, and the length of the delay is varied. This creates the effect of multiple instruments because the ear perceives two sounds that are not quite in sync.

Also since the delay is varying, the delayed signal is continuously speeding up and slowing down which causes the pitch of the delayed signal to vary slightly. This increases the perception of multiple instruments.

The chorus program used for testing is shown in appendix C.4. This filter can also be used as a "flange" or comb filter by using a shorter delay and as an echo effect by using a longer delay.

5.2.3 Compression

The final test filter that was written is a compressor. A compressor reduces the dynamic range of a signal by decreasing the volume of the loud parts. The test compressor is shown in appendix C.5. This is a fairly basic compressor but could easily be improved.

The `rms()` function calculates an RMS average signal level for the region around the current point in the file. This level is then used to adjust the output gain—the gain is adjusted to keep the RMS level below the threshold value of 0.9.

Unlike all the other tests, this compressor unfortunately does not work. This seems to be caused by a bug in muc, but the problem could not be found within the time frame of the thesis work. This compressor should work, given a properly working compiler, and was quite straightforward to write. It should also be possible to write a more elaborate compressor by making some simple modifications to this code.

6 Analysis

As stated earlier, the motivation behind the implementation of muc was to test the design of the μ language by implementing some synthesizers and filters in μ.

The stated objective of μ was not to become the tool of choice for programmers creating digital filters, but rather to create an easy way for people to experiment with audio filters and synthesizers. It is my opinion the μ is well suited towards this end, although it requires some additional features.

6.1 Effectiveness For Writing Synthesizers

In comparison to other programming languages, μ makes writing synthesizers easier by automatically handling extraneous details and allowing the user to focus on the actual synthesis of the signal. It is easy to experiment with different synthesizer designs.

The syntax of μ is easier to learn and remember than computer music languages such as CSound while remaining much more flexible than any analog modelling synthesizer. I found that it was straightforward to implement standard synthesizer designs in μ, yet it is not limited to these designs.

To be a truly effective means of creating synthesizers, μ needs better support for playing music once the actual signal generator is created. It is easy to create interesting single note or chord sounds with μ, but it is very cumbersome to get a μ program to play an actual song. The method used in C.6 of controlling pitch and volume with conditional functions is virtually unusable—constructing a very simple melody is a great deal of work.

One possible solution to this is to allow μ programs to have input parameters specifying pitch, volume and aftertouch. A wrapper could then be created which could read MIDI data and invoke the μ program for each note in the piece. Alternatively, the wrapper could allow the program to function in a similar manner as a virtual instrument plug-in for recording software.

6.2 Effectiveness For Writing Filters

I believe that μ is also a useful language for creating audio filters with. Although it may not offer huge improvements over other languages, such as C++, it does simplify the task of creating and experimenting with filters, and the μ paradigm may inspire different filter designs simply by being different

itself. Also, μ makes it easier for mathematically-inclined non-programmers to write digital filters by abstracting away the von Neumann architecture and using more familiar notation.

7 Future Directions

There are many possibilities for the future development of μ. Some of the ideas in this section were part of the original conception of μ; others came about as a result of the implementation and evaluation described in this thesis.

7.1 Optimization

Perhaps the most obvious way in which muc itself could be improved is in the area of optimization. Currently muc makes no attempt at optimizing the code which it outputs. The inefficiency is particularly bad when doing summations: in the compressor test program from C.5, for example, all the samples inside the window are read from the file each time `rms()` is invoked, despite the fact that all but one of them would have been read already.

There are a lot of possibilities for optimizing the code generated by muc. First, the generated code often contains a lot of redundancy—values that are used more than once are generally calculated more than once. Second, the code would benefit from an input and output buffering scheme. Lastly, the absence of side effects and chronological sequences of instructions in μ allows a compiler a lot of options in rearranging the code to get the most efficient execution.

7.2 Input/Output

The input and output buffering mentioned above would not only improve the efficiency of μ programs, it also opens up new i/o possibilities. If input buffering is used to ensure that the input files are read sequentially from start to finish, then the inputs and outputs no longer have to be files, they can be pipes to and from other processes, or any other kind of data stream. μ filters could be applied in real time. This would also allow μ programs to be used as filter plug-ins for recording software.

Another thing muc is missing is the ability to read from and write to multiple files in a single program. This would be simple to implement but couldn't be completed in the time available for this project.

7.3 Extensions to Syntax

There are a number of useful additions which could be made to the μ syntax. One such possibility is a mechanism for specifying program parameters. The benefits of this for synthesizers are mentioned in section 6.1, it also would allow file name and filter parameters to be specified at run-time for filters.

It could also be useful to include numerical integration and differentiation as operators, as well as fast fourier transform. Functions with more than one parameter might also be useful.

8 Conclusion

In the introduction I stated that for my thesis project I designed and implemented a special-purpose programming language for audio digital filter and synthesizer creation, and evaluated this language based on it's ease of use.

The language, called μ, is a first-order pure functional programming language with syntax and semantics modelled after standard mathematical notation. I implemented a compiler, muc, which compiles μ programs into C++ code. Muc is not a full-featured compiler system, but rather it's purpose is to facilitate evaluation of the basic design concepts of the μ language.

After writing some synthesizers and filters in μ, I decided that it was a good design which met it's goals. More work needs to be done to make μ and muc into a complete and useful system. In the final sections of this report I discussed what directions this future work could take.

Overall I am pleased with my work on μ, although I wish more time had been available to implement some different features and perform more tests. I will probably continue to work on μ in the future, and eventually it may become something which is useful to other people.

A μ Language Grammar

A.1 Lexical Specification

Whitespace is ignored in μ files. Anything after "//" until the end of the line is ignored.

The following arithmetic operators are recognized:

`+` addition
`-` subtraction
`-` negation
`*` multiplication
`^` exponentiation
`/` division
`%` modulus

The following single-character tokens are also recognized:

```
' ( ) = < > , # |
```

Integers and floats are defined as:

`[0-9]+`	*INTEGER*
`[0-9]+\.[0-9]+`	*FLOAT*

Identifiers take the form:

`[a-zA-Z][0-9a-zA-Z_]*`	*IDENT*

And qutoted strings are recognized:

`\"[^"]*\"`	*QUOTED_STRING*

The following are reserved words in μ:

Operators	Data Types	Other keywords
INT	**int8**	**parameters**
SUM	**int12**	**input**
	int16	**from**
Built-in Functions	**int20**	**output**
cos	**int24**	**to**
sin	**int32**	**include**
tan	**float32**	**default**
arccos	**float64**	**datatype**
arcsin	**string**	**resolution**
arctan		**for**
		otherwise
		length

All built-in functions are recognized by the lexer as the *BUILTIN_FUNC* token and all data types are recognized as occurrences of the *DATATYPE* token. The other reserved words are each their own token.

A.2 μ Grammar Specification

MuProgram	→	*PragmaList DefList*
PragmaList	→	*GlobalPragma*
	→	*PragmaList GlobalPragma*
GlobalPragma	→	*InputPragma*
	→	*OutputPragma*
	→	*IncludePragma*
InputPragma	→	# **input** *IDENT* **from** *QUOTED_STRING*
OutputPragma	→	# **output** *IDENT* **to** *QUOTED_STRING* **length** *INTEGER* **datatype** *DATATYPE*
IncludePragma	→	# **include** *QUOTED_STRING*
DefList	→	*ConstDef*
	→	*FuncDef*
	→	*DefList ConstDef*
	→	*DefList FuncDef*
ConstDef	→	*IDENT* = *ConstExpr*
FuncDef	→	*IDENT* (*IdentList*) = *Expr*
	→	*IDENT* (*IdentList*) = *Expr* **for** *Condition*
	→	*IDENT* (*IdentList*) = *Expr* **for** *Condition ClauseList*
IdentList	→	*IDENT*
	→	*IdentList* , *IDENT*
ClauseList	→	*ExprClause*
	→	*ClauseList ExprClause*
ExprClause	→	| *Expr* **for** *Condition*
	→	| *Expr* **otherwise**
Number	→	*INTEGER*
	→	*FLOAT*
	→	*IDENT*

$$
\begin{array}{rcl}
\textit{Condition} & \rightarrow & \textit{Number} < \textit{IDENT} < \textit{Number} \\
& \rightarrow & \textit{Number} < = \textit{IDENT} < \textit{Number} \\
& \rightarrow & \textit{Number} < \textit{IDENT} < = \textit{Number} \\
& \rightarrow & \textit{Number} < = \textit{IDENT} < = \textit{Number} \\
& \rightarrow & \textit{Number} > \textit{IDENT} > \textit{Number} \\
& \rightarrow & \textit{Number} > = \textit{IDENT} > \textit{Number} \\
& \rightarrow & \textit{Number} > \textit{IDENT} > = \textit{Number} \\
& \rightarrow & \textit{Number} > = \textit{IDENT} > = \textit{Number} \\
& \rightarrow & \textit{IDENT} < \textit{Number} \\
& \rightarrow & \textit{IDENT} > \textit{Number} \\
& \rightarrow & \textit{Number} < \textit{IDENT} \\
& \rightarrow & \textit{Number} > \textit{IDENT} \\
& \rightarrow & \textit{IDENT} < = \textit{Number} \\
& \rightarrow & \textit{IDENT} > = \textit{Number} \\
& \rightarrow & \textit{Number} < = \textit{IDENT} \\
& \rightarrow & \textit{Number} > = \textit{IDENT} \\
& \rightarrow & \textit{Number} = \textit{IDENT} \\
& \rightarrow & \textit{IDENT} = \textit{Number} \\
\textit{ConstExpr} & \rightarrow & \textit{ConstTerm} \\
\textit{ConstExpr} & \rightarrow & \textit{ConstExpr Op1 ConstTerm} \\
\textit{ConstElement} & \rightarrow & (\ \textit{ConstExpr}\ ) \\
& \rightarrow & \textit{INTEGER} \\
& \rightarrow & \textit{FLOAT} \\
& \rightarrow & \textit{IDENT} \\
& \rightarrow & \textit{ConstAbs} \\
& \rightarrow & \textit{BuiltinFunc}\ (\ \textit{ConstExpr}\ ) \\
\textit{ConstTerm} & \rightarrow & \textit{ConstFactor} \\
& \rightarrow & \textit{ConstTerm Op2 ConstFactor} \\
\textit{ConstFactor} & \rightarrow & \textit{ConstP} \\
& \rightarrow & \textit{ConstP}\ \hat{}\ \textit{ConstFactor} \\
\textit{ConstP} & \rightarrow & -\ \textit{ConstP} \\
& \rightarrow & \textit{ConstElement} \\
\textit{ConstAbs} & \rightarrow & |\ \textit{ConstExpr}\ | \\
\textit{Expr} & \rightarrow & \textit{Term} \\
& \rightarrow & \textit{Expr Op1 Term}
\end{array}
$$

Element	→	(*Expr*)
	→	*INTEGER*
	→	*FLOAT*
	→	*IDENT*
	→	*IDENT* (*Expr*)
	→	*BuiltinFunc* (*Expr*)
	→	*Integral*
	→	*Sum*
	→	*Abs*
Term	→	*Factor*
	→	*Term Op2 Factor*
Factor	→	*P*
	→	*P* ^ *Factor*
P	→	- *P*
	→	*Element*
Abs	→	| *Expr* |
Integral	→	**INT** (*Expr*) *IDENT*
Sum	→	**SUM** (*IDENT* = *Expr* **to** *Expr*) (*Expr*)
Op1	→	+
	→	-
Op2	→	*
	→	/
	→	%

B Parse.h

```
#include <stdio.h>

#ifndef PARSECONSTS_H
#define PARSECONSTS_H

#define ZERO_DBFS 1

#define true            1
#define false           0
#define UNKNOWN_TYPE    0
#define BUILTIN_SIN     1
#define BUILTIN_COS     2
#define BUILTIN_TAN     3
#define BUILTIN_ARCSIN  4
#define BUILTIN_ARCCOS  5
#define BUILTIN_ARCTAN  6
#define TYPE_FUNC       7
#define TYPE_CONST      8
#define TYPE_PARAMETER  9
#define TYPE_INT8       10
#define TYPE_INT12      11
#define TYPE_INT16      12
#define TYPE_INT20      13
#define TYPE_INT24      14
#define TYPE_INT32      15
#define TYPE_FLOAT32    16
#define RANGE_INFINITY  0x7FFFFFFF
#define RANGE_NEG_INFINITY 0xFFFFFFFF

#define GTLT   0
#define GTLTE  1
#define GTELTE 2
#define GTELT  3

extern int   lineNum;
extern char *infileName;
```

```
extern char *outfileName;
extern FILE *outfile;
extern int   semanticErrorDetected;

extern char *audioOutFilename;
extern struct TSymTabEnt *outFunc;
extern int   outFileLen;
extern int   outFileDatatype;
extern char *audioInFilename;

typedef struct TExpression EExpression;
typedef struct TElement EElement;
typedef struct TTerm ETerm;
typedef struct TFactor EFactor;
typedef struct TP EP;
typedef struct TAbs EAbs;
typedef struct TSum ESum;
typedef struct TFuncListEnt FuncListEnt;


/*********************************************************
 * Symbol Table                                          *
 *********************************************************/

typedef struct TSymTabEnt {
    char* name;
    int type;
    int ivalue;
    float fvalue;
    EExpression *parseTree;
    FuncListEnt *func;

    struct TSymTabEnt *next;
} SymTabEnt;

int symTabInitialize();
SymTabEnt *addSymbol( char *name );
SymTabEnt *getSymbol( char *name );
```

```
/*********************************************************
 * Expression Trees                                      *
 *********************************************************/

struct TExpression {
    enum { expr_term, expr_op, expr_file } type;
    ETerm *term;
    char op;
    EExpression* expr;
    char *filename;
};

struct TElement {
    enum {
        element_expr,
        element_integer,
        element_float,
        element_const,
        element_func,
        element_builtin_func,
        element_sum,
        element_abs
    } type;
    EExpression *expr;
    union {
        int integer;
        float floatp;
        SymTabEnt *constant;
        SymTabEnt *func;
        int builtinFunc;
        ESum *sum;
        EAbs *abs;
    } child;
};

struct TTerm {
    enum { term_factor, term_op } type;
```

```
    EFactor *factor;
    char op;
    struct TTerm *term;
};

struct TFactor {
    enum { factor_p, factor_exponent } type;
    EP *p;
    EFactor *factor;
};

struct TP {
    enum {p_p,p_element} type;
    EElement *element;
    EP *p;
};

struct TAbs {
    EExpression *expr;
};

struct TSum {
    SymTabEnt *iter;
    EExpression *start;
    EExpression *end;
    EExpression *expr;
};

/*********************************************************
 * Constant List                                         *
 *********************************************************/

typedef struct TConstListEnt {
    struct TConstListEnt *next;
    SymTabEnt *symbol;
} ConstListEnt;

void constListInitialize();
```

```
ConstListEnt *getConstListHead();
void addConst( SymTabEnt *symbol );

/*********************************************************
 * Function List                                         *
 *********************************************************/

typedef struct TExpressionListEnt {
    EExpression *expr;
    float start;
    float end;
    int rangetype;
    struct TExpressionListEnt *next;
} ExpressionListEnt;

typedef struct TFuncCallListEnt {
    SymTabEnt *func;
    EExpression *param;
    struct TFuncCallListEnt *next;
} FuncCallListEnt;

struct TFuncListEnt {
    SymTabEnt *symbol;
    EExpression *otherwiseExpr;
    ExpressionListEnt *exprList;
    SymTabEnt *parameters[1];
    FuncCallListEnt *nestedCalls;
    struct TFuncListEnt *next;
};

void funcListInitialize();
FuncListEnt *getFuncListHead();
FuncListEnt *addFunc( SymTabEnt *symbol, SymTabEnt *parameter, EExpression *otherw

/*********************************************************
 * Code Generators                                       *
 *********************************************************/
```

```
void printFunc( FuncListEnt*func );
void printConstExpressions( FILE *outfile );
void printFuncExpressions( FILE *outfile );
void printExpr( EExpression *expr, char *temp );
void printElement( EElement *element, char *temp );
void printTerm( ETerm *term, char *temp );
void printFactor( EFactor *factor, char *temp );
void printP( EP *p, char *temp );
void printAbs( EAbs *abs, char *temp );
void printSum( ESum *sum, char *temp );

#endif
```

C Sample μ Programs

C.1 Additive Synthesis

Following is an example of an additive synthesizer implemented in μ. It is a simplified simulation of a Hammond organ. See section 5.1.1 for a complete discussion of this program and section C.6 for an example of this synthesizer in action.

```
#output o to "organ.wav" length 800000 datatype int16

Pi = 3.141592654
SampleRate = 44100  //CD sampling rate
//Constant to Hz to samples/sec
Hz = 2*Pi/SampleRate

//Define low A (Hz)
A = 220

//Signal gain (amplification) just before output
gain = 1

//The organ drawbar settings
db1 = 10    //16'
db2 = 2     //5 1/3 '
db3 = 8     //8'
db4 = 7     //4'
db5 = 2     //2 2/3 '
db6 = 6     //2'
db7 = 2     //1 3/5 '
db8 = 4     //1 1/3 '
db9 = 8     //1'

//Here's where the real work is is done.
//Generate the tones corresponding to the nine tone wheels.
wheel1(t) = sin(t*2*Pi/SampleRate)
wheel2(t) = sin(2*t*2*Pi/SampleRate)
wheel3(t) = sin(3*t*2*Pi/SampleRate)
wheel4(t) = sin(4*t*2*Pi/SampleRate)
```

```
wheel5(t) = sin(5*t*2*Pi/SampleRate)
wheel6(t) = sin(6*t*2*Pi/SampleRate)
wheel7(t) = sin(7*t*2*Pi/SampleRate)
wheel8(t) = sin(8*t*2*Pi/SampleRate)
wheel9(t) = sin(9*t*2*Pi/SampleRate)
tone(t) = db1*wheel1(t) + db2 *wheel2(t) + db3*wheel3(t)
             +db4*wheel4(t) + db5*wheel5(t) + db6*wheel6(t)
             +db7*wheel7(t) + db8*wheel8(t) + db9*wheel9(t)

//Here is the actual output - play low A
o(s) = gain*tone(A*t)/90
```

C.2 PWM Synthesis

Here is an example of a string pad created using PWM synthesis which demonstrates the use of wavelets in μ. It is discussed in section 5.1.2.

```
#output o to "synthstrings.wav" length 400000 datatype int16

//Pitch of middle-A in Hz.
pitch = 440

//Define a triangle wavelet over [0,1]
triangle_wavelet(x) =  4*x       for 0    <= x < 0.25
                    | -4*(x-0.5) for 0.25 <= x < 0.75
                    |  4*(x-1)   for 0.75 <= x < 1
                    |  0         otherwise

//Now use the triangle wavelet to build a triangle wave
triangle_wave(x) = triangle_wavelet(x%1)

//Define a pulse wavelet over [-1,1]
pulse_wavelet(x) =  1 for  0 <= x <= 1
                 | -1 for -1 <= x <  0
                 |  0 otherwise

//Some low frequency oscillators to modulate the pulse waves with
lfo1(t) = 0.4*triangle_wave(8*t)
```

```
lfo2(t) = 0.4*triangle_wave(7.5*t)
lfo3(t) = 0.4*triangle_wave(6.15*t)

//Create three modulated pulse waves with the lfos defined above
PWMwave1(t) = pulse_wavelet(((t*pitch)%1)-0.75+0.25*lfo1(t))
PWMwave2(t) = pulse_wavelet(((t*pitch)t%1)-0.75+0.25*lfo2(t))
PWMwave3(t) = pulse_wavelet(((t*pitch/2)%1)-0.75+0.25*lfo3(t)))

//Finally, combine the PWM waves together to create the output
o(s) = 0.5*PWMwave1(s/SampleRate) + 0.5*PWMwave2(s/SampleRate)
                 + 0.5*PWMwave3(s/SampleRate)
```

C.3 Stereo Mixdown

This program mixes the two channels of a stereo file and produces a mono file. The current version of the μ specification does not say what to do with stereo files; the current version of muc returns the left channel as the even elements of the input function and the right channel as the odd elements. This program does not, therefore, conform strictly to the μ specification.

```
#input i from "The Microsoft Sound.wav"
#output o to "MSSound-mono.wav" length 150000 datatype int16

left(s) = i(x*2)
right(s) = i(x*2+1)

o(s) = 0.5*left(s) + 0.5*right(s)
```

C.4 Chorus

This μ program applies a chorus effect to a file, as described in section 5.2.2.

```
#input f_in from "input.wav"
#output f_out to "output.wav" length 100000 datatype int16

ChorusRate = 6     //in Hz
ChorusDepth = 0.5  //in milliseconds
```

```
lfo(x) = ChorusDepth*44.1*sin(ChorusRate*2*Pi*x/SampleRate)
chorused(t) = 0.5*melody(beatn(t)) + 0.5*melody(beatn(t-ChorusDepth+lfo(t)))
```

C.5 Compression

Here is a basic compressor implemented in μ. See section 5.2.3 for description.

```
#output f_out to "comped.wav" length 800000 datatype int16
#input f_in from "input.wav"

//lookahead and window width in samples
lookahead = 40
window = 80

gain = 1.1

//calculate rms average volume over the window.
rms(x) = (SUM( i = x-window to x )( f_in(i+lookahead)^2 )/win)^0.5

//The compression function
//maps input volume to output gain
compfunc(a) = a   for 0 <= a <= 0.9
            | 0.9 for a > 0.9
            | 0   otherwise

compressed(x) = f_in(compfunc(rms(x))/rms(x))*f_in(x)

f_out(x) = gain*compressed(x)
```

C.6 Organ Melody

```
#output o to "organ.wav" length 800000 datatype int16

Pi = 3.141592654
SampleRate = 44100  //CD sampling rate
//Constant to Hz to samples/sec
Hz = 2*Pi/SampleRate
```

```
bpm = 120    //Tempo in beats per minute

//Pitches (in Hz) for some notes
C = 261.63
D = 293.66
E = 329.63
F = 349.23
G = 392.00

//Signal gain (amplification) just before output
gain = 1

//The organ drawbar settings
db1 = 10    //16'
db2 = 2     //5 1/3 '
db3 = 8     //8'
db4 = 7     //4'
db5 = 2     //2 2/3 '
db6 = 6     //2'
db7 = 2     //1 3/5 '
db8 = 4     //1 1/3 '
db9 = 8     //1'

//Convert samples to beats
beatn(s) = s*(bpm/60)/SampleRate
//Convert beats to samples
sampleb(beat) = beat*SampleRate/(bpm/60)

//Here's where the real work is is done.
//Generate the tones corresponding to the nine tone wheels.
wheel1(t) = sin(t)
wheel2(t) = sin(2*t)
wheel3(t) = sin(3*t)
wheel4(t) = sin(4*t)
wheel5(t) = sin(5*t)
wheel6(t) = sin(6*t)
wheel7(t) = sin(7*t)
wheel8(t) = sin(8*t)
```

```
wheel9(t) = sin(9*t)
tone(t) = db1*wheel1(t) + db2 *wheel2(t) + db3*wheel3(t)
            +db4*wheel4(t) + db5*wheel5(t) + db6*wheel6(t)
            +db7*wheel7(t) + db8*wheel8(t) + db9*wheel9(t)

//Here we define the "envelopes"-how volume
//changes over time for each note.
envelope_quaternote(x) = 1          for 0   <= x < 0.9
                       | -10*x + 10 for 0.9 <= x < 1
                       | 0          otherwise
envelope_halfnote(x) = 1          for 0   <= x < 1.9
                     | -10*x + 20 for 1.9 <= x < 2
                     | 0          otherwise
envelope_fade(x) = 1         for 0   <= x < 1.9
                 | -x/6 + 20 for 1.9 <= x < 7.9
                 | 0         otherwise

//Now play a melody by varying pitch over time
//t is in beats.
melody(t) = tone(E*sampleb(t)*Hz)     for  0 <= t <  2
          | tone(F*sampleb(t-2)*Hz)  * envelope_quaternote(t-2)  for  2 <= t <  3
          | tone(G*sampleb(t-3)*Hz)  * envelope_halfnote(t-3)    for  3 <= t <  5
          | tone(F*sampleb(t-5)*Hz)  * envelope_quaternote(t-5)  for  5 <= t <  6
          | tone(E*sampleb(t-6)*Hz)  * envelope_quaternote(t-6)  for  6 <= t <  7
          | tone(D*sampleb(t-7)*Hz)  * envelope_quaternote(t-7)  for  7 <= t <  8
          | tone(C*sampleb(t-8)*Hz)  * envelope_halfnote(t-8)    for  8 <= t < 10
          | tone(D*sampleb(t-10)*Hz) * envelope_quaternote(t-10) for 10 <= t < 11
          | tone(E*sampleb(t-11)*Hz) * envelope_quaternote(t-11) for 11 <= t < 12
          | tone(E*sampleb(t-12)*Hz) * envelope_halfnote(t-12)   for 12 <= t < 14
          | tone(D*sampleb(t-14)*Hz) * envelope_halfnote(t-14)   for 14 <= t < 16
          | tone(C*sampleb(t-16)*Hz) * envelope_halfnote(t-16)   otherwise

//A chorus effect is used to give the organ a "fuller" sound.
ChorusRate = 6     //in Hz
ChorusDepth = 0.5  //in milliseconds
lfo(x) = ChorusDepth*44.1*sin(ChorusRate*2*Pi*x/SampleRate)
chorused(t) = 0.5*melody(beatn(t)) + 0.5*melody(beatn(t-ChorusDepth+lfo(t)))
```

```
//Here is the actual output
o(s) = gain*chorused(s)/90
```

D Code Generation Example

Below is excerpts of the C++ code generated by muc for the organ melody program in C.6. The triple dots indicate places where parts of the code have been omitted. This listing displays the overall layout of the generated code along with some examples of generated functions.

```
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include "libmu.h"

float Pi;
float SampleRate;
float bpm;
float C;
.
.
.
float Hz;
float ChorusRate;
float ChorusDepth;
inline float o( float t );
.
.
.
inline float envelope_quater( float x );
inline float sampleb( float beat );
inline float beatn( float s );

inline float o( float t ) {
    float _retval;
    float _t0;
    float _t1;
    _t1 = gain;
    float _t2;
    float _t3;
    _t3 = t;
```

```
    _t2 = chorused(_t3);
    _t0 = _t1 * _t2;
    _t1 = (float)90;
    _retval = _t0 / _t1;
    return _retval;
}

.
.
.

inline float envelope_quater( float x ) {
    float _retval;
    if( x >= 0 && x < 0 ) {
        float _t0;
        _t0 = (float)20;
        float _t1;
        _t1 = x;
        _retval = _t0 * _t1;
    } else if( x > 0.45 && x <= 4.01116e-34 ) {
        float _t0;
        float _t1;
        _t1 = (float)2;
        _t1 =-_t1;
        float _t2;
        float _t3;
        float _t4;
        _t4 = x;
        float _t5;
        _t5 = 0.450000;
        _t3 = _t4 - _t5;
        _t2 = _t3;
        _t0 = _t1 * _t2;
        _t1 = (float)1;
        _retval = _t0 + _t1;
    } else if( x >= 0 && x <= 0.45 ) {
        _retval = (float)1;
    } else {
```

```
        _retval = (float)0;
    }
    return _retval;
}

inline float sampleb( float beat ) {
    float _retval;
    float _t0;
    float _t1;
    _t1 = beat;
    float _t2;
    _t2 = SampleRate;
    _t0 = _t1 * _t2;
    float _t3;
    _t3 = bpm;
    float _t4;
    _t4 = (float)60;
    _t2 = _t3 / _t4;
    _t1 = _t2;
    _retval = _t0 / _t1;
    return _retval;
}

inline float beatn( float s ) {
    float _retval;
    float _t0;
    float _t1;
    _t1 = s;
    float _t2;
    float _t3;
    float _t4;
    _t4 = bpm;
    float _t5;
    _t5 = (float)60;
    _t3 = _t4 / _t5;
    _t2 = _t3;
    _t0 = _t1 * _t2;
    _t1 = SampleRate;
```

```
    _retval = _t0 / _t1;
    return _retval;
}


int main()
{
    int _outputLength = 800000;
    if(0==_openOutWav("organ.wav", 12))
    {
        fprintf(stderr,"Error:Could not open \"organ.wav\" for writing.\n");
        return 1;
    }
    {
        Pi = 3.141593;
    }
    {
        SampleRate = (float)44100;
    }
    {
        bpm = (float)120;
    }
    {
        C = 261.630005;
    }
    .
    .
    .
    {
        float _t0;
        float _t1;
        _t1 = (float)2;
        float _t2;
        _t2 = Pi;
        _t0 = _t1 * _t2;
        _t1 = SampleRate;
        Hz = _t0 / _t1;
    }
```

```
    {
        ChorusRate = (float)6;
    }
    {
        ChorusDepth = 0.500000;
    }

    for( int _i = 0; _i < _outputLength; _i++ )
    {
        _writeSample(o((float)_i));
    }
    _closeOutWav();
    return 0;
}
```

E Glossary

Dynamic Range - The difference in volume between the quiet parts and loud parts of a piece of music.

LFO - Low Frequency Oscillator; an oscillator with a subsonic frequency. LFOs have a frequency to low to be heard, but are used to modulate parameters such as the delay in a chorus or the duty cycle in a pulse wave.

MIDI - Musical Instrument Digital Interface—Protocol for sending music data over serial lines. MIDI signals do not define the sound of the music; they contain the notes, volume and other information that allow an electronic keyboard or synthesizer to play a piece of music. MIDI was originally created as a standard interface between keyboards and synthesizers, but MIDI data can also be saved in files.

PCM - Pulse Code Modulation - the method of storing audio digitally by regularly "sampling" the wave height and storing the value as a digital number.

PWM - Pulse Width Modulation—see section 5.1.2.

Timbre - A violin playing A-440 and a flute playing A-440 sound different because they have a different *timbre*. Each is creating a 440Hz tone and a collection of harmonics above it, but which harmonics are present and to what extent, along with how they vary over time, is different. This is what gives different instruments different sounds or *timbres*.

F CS4997 Summary Sheet

References

[1] Appel, Andrew. 2002. *Modern Compiler Implementation in Java.* Cambridge University Press, Cambridge.

[2] Bagwell, Chris. *Audio File Formats FAQ*, [On-Line]. Available: http://home.attbi.com/ chris.bagwell/AudioFormats.html [Accessed on 2002-20-11]

[3] Booth, David, et. al. *The CSound Manual*, [On-Line]. Available: http://www.lakewoodsound.com/csound/ [Accessed on 2002-20-11]

[4] Lesk, M. E. and Schmidt, E. *Lex - A Lexical Analyzer Generator*, [On-Line]. Available: http://dinosaur.compilertools.net/lex/index.html [Accessed on 2003-9-4]

[5] Johnson, Stephen. *Yacc: Yet Another Compiler-Compiler*, [On-Line]. Available: http://dinosaur.compilertools.net/yacc/index.html [Accessed on 2003-9-4]

[6] Papoulis, Athanasios. 1977. *Signal Analysis.* McGraw-Hill, New York.

[7] Reid, Gordon. 2000. Synth Secrets Part 10: Modulation. In *Sound On Sound Magazine, February 2000.* SOS Publishing, Cambridge.

[8] Reid, Gordon. 2003. Synth Secrets: Synthesizing Strings—PWM & String Sounds. In *Sound On Sound Magazine, March 2003.* SOS Publishing, Cambridge.

[9] Sebesta, Robert. 2002. *Concepts of Programming Languages.* Addison-Wesley Publishing Company, Boston.

[10] Sloan, Randy. 2002. *The Audiophile's Project Sourcebook.* McGraw-Hill, New York.

[11] Smith, Julius. *Introduction to Digital Filters.* [On-Line]. Available :http://www.ccrma.stanford.edu/ jos/filters/ [Accessed on 2002-20-11]

[12] Smith, Steven. 1999. *The Scientist and Engineer's Guide to Digital Signal Processing.* California Technical Publishing. San Diego.

[13] Steiglitz, Ken. 1996 *A Digital Signal Processing Primer.* Addison-Wesley Publishing Company, Menlo Park.

[14] Taylor, Fred. 1983. *Digital Filter Design Handbook.* Marcel Dekker, New York.