Atomic Sound Manipulation Language (ASML)

Final Report

Team SoundHammer

Frank Smith – fas2114

Tim Favorite – tuf1

I. Introduction

Atomic Sound Manipulation Language is Team SoundHammer's solution to the problem of how to make a high level computer language that describes effects that can be applied to sound waves. The idea is to give the programmer access to the most elemental levels of a sound's composition in an easy, intuitive way. This elemental access to sounds is the idea from which our language derives its name, as we intended the language to operate upon sound at the "atomic" level.

ASML will feature typical computer language constructs such as loops, if statements, comparators, and arithmetic operators. It will use these features to manipulate new primitive types based upon the most fundamental components of sound: frequency, amplitude, and time. Programmers in ASML will be able to use these statements and operators to produce general purpose and specialized functions to act upon sound waves, their individual frequencies, and other available types. Programmers will then be able to aggregate these functions into even more robust effects that can be intelligently triggered.

Team SoundHammer would like ASML to bring power and ease to the manipulation of sound waves in the same way that a language like AWK makes it easy to write powerful text file manipulation programs. We would like to have the user be able to use simple commands to open wav files, isolate frequencies and time ranges from those files, and manipulate the data therein on either a micro or a macro scale.

Functional Aspects

Functionality

Any real world sound is, in essence, a sum of the amplitudes of the range of frequencies audible to the human ear (20 Hz - 20 kHz). Each frequency is represented by a simple sine wave with a unique wavelength. When a dog barks, or a violin plays, many frequencies are emitted, each at different amplitudes, and the combination of these (called a complex wave) is what gives the sound its unique character.

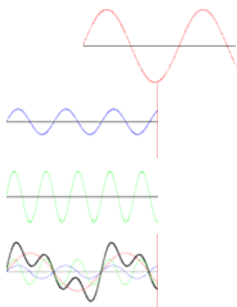


Figure 1: Three simple waves followed by a complex wave made up of the sum of the three simple waves (in black) (obtained from http://www.umanitoba.ca/faculties/arts/linguistics/russell/138/sec4/specgraf.htm)

So to work with a sound at the most basic level and perform functions such as pitch shifting or high-pass filtering, we must be able to pull out individual frequencies or ranges of frequencies from the complex wave of an input sound, and manipulate them. We'll represent input sound as an indexed list of frequencies, and users will be able to access an individual frequency as in the following example, where we are accessing the 440 Hz frequency:

```
wave f440 = input at 440Hz;
```

Users will also be able to access a given sample time from the input sound, as in the following example, where we access a 1 second range of samples starting at 5 seconds into the input:

```
wave sec5 = input at 5000ms to 6000ms;
```

Major Characteristics

ASML is implemented as an imperative language - with the notion of states intact. Programs will accept .wav files as input and move the input through a series of functions in order to accomplish the desired effects upon the sounds.

ASML is a strongly typed language. While we will not be allowing users to create user defined types, we would also like to ensure against users doing actions that do not make sense within the context of sound manipulation. There is no reason for a user to write an equation for an amplitude as a function of a frequency, so a user is prevented from doing this.

- For example:
  - freq f = 10000Hz;
  - ampl a = f/16;

Such an assignment does not make sense from the perspective of sound manipulation because frequency and amplitude are measurements of two collaborating but unrelated characteristics of sound. What is more significant with respect to why this is a strongly typed language is that such a construction is not really useful. The numbers applying to these two different characteristics are on different scales and magnitudes and simply do not play well together for such a function, so we intend to make the program more strict in this respect.

Although this paper has thus far focused on proposed new primitive types, the language will include such standbys as ints, floats, and strings as well. There is clear value in having arithmetic types in such a language, and we will allow new primitives to be coerced into these old types where that is logical.

Furthermore, these types can be useful for purposes of debugging in the language, and this is a reason that we include a console printing facility.

The grammar designed for the ASML language has a framework outlined where other ASML programs could be included within the current program, enabling library support, however, due to time constraints, we did not implement this.

Finally, it is a goal of ours to try to implement a whole language syntax style that is clearer and more readable than most existing high level languages. The small pieces of code so far are intended to represent our notion of how this could look and it is a standard that we will try to accomplish in our working version of the compiler.

Control Flow

Program invocations will always have at least two arguments: a program file, specified by a -p, and an input file, specified by a -i in the command line. In the code, this is represented by the symbol input. Additionally, an optional output file argument is allowed, specified by a -o. Further arguments will be represented within the code by formal arguments declared in the declaration of main. When the program is executed, it searches the code for a main function, and then executes all of the code inside of it. External function calls to functions defined in the same file are permitted. The main must end with a return statement that returns a data structure that can be written to a wav file, which will be written to the output file (if there is a -o argument) or back to the input file (if there is no -o).

Possible Applications

ASML allows for the creation of new sound effects by sound engineers in the realms of music, television and film, and telecommunications. Future support for libraries will make a standard set of effects readily available for sound engineers who are satisfied with the broad range of effects available in applications such as Pro Tools, but their open nature will allow for the potential of tweaking the effects as well, should that be desirable.

Challenges

Major challenges involved finding the tools in the Java Sound API to achieve the granularity desired in our language - it seems to have been designed with pre-set mixer functions already built in. Some of the limitations of this API were overcome by using the open-source Tritonus library (http://tritonus.org), however this project seems to have been discontinued several years ago, and there seemed to be no other libraries that reached even the slight improvement on Java Sound API that Tritonus did. By the time we reached the realization that Java might not be the best language to implement this in, we had made too much progress to reverse course and implement in C, which most DSP libraries seem to be implemented in (due to its speed advantages over Java). However, as the focus of the class is not sound, we will implemented the features that required only the most casual knowledge of music theory and calculus.

II. Language Tutorial

1. Command Line

Before running an ASML program, a user must ensure that Java 1.5 or higher is installed, and that the Tritonus jar file is in the classpath. Additionally, to generate code from the grammar files, ANTLR 3 is required. When running an ASML program from the command line, two arguments are required - a program file and an input file, which are specified by a -p and a -i flag. Another argument of note is the output file, specified by the -o flag. This argument is optional, but if no output file is specified then the input file will be overwritten. Additional arguments are required when the main method in the program calls for them. An example of a call to the command line:

```
java asml.ASML -p HelloWorld.asml -i "test.wav"
```

This will call the program HelloWorld.asml and give it an input file of test.wav. The program will overwrite test.wav when it terminates and returns an output wave. To direct the output elsewhere, simply declare a -o argument as previously mentioned:

```
java asml.ASML -p HelloWorld.asml -i "test.wav" -o "output.wav"
```

And to pass additional arguments, say an integer:

```
java asml.ASML -p HelloWorld.asml -i "test.wav" -o "output.wav" 2
```

2. Structure of a Program

Every ASML program that can be executed must have a main function which returns a wave. When a program is executed, it will execute the code in main, traveling outside of main only if directed to do so by main's code. Additional declared functions are possible but not required. To declare a function, use the following syntax:

```
fun wave main()
 /* ... main function block ... */
end fun
```

Begin with the keyword fun to indicate it is a function. Then, follow up with the return type of the function (wave), followed by the name of the function (in this case, main), followed by the parameter list surrounded by parentheses. The body of the function is next, and will terminate once reaching the keywords "end fun".

3. Turning up the Volume

Say you'd like to double the volume on a wave file. It's very easy to do in ASML - simply multiply the wave by 2.

```
fun wave main()
   wave doublewave = 2 * input;
   return doublewave;
end fun
```

Some things to note about this program: input is a wave that is automatically added whenever a program is invoked with the -i flag (which will be all the time, since programs will not run without an input). ASML looks for the file specified by the string and then reads from it to create the wave. When a wave is multiplied by a "scalar" (an int or a float), every sample in the wave will be multiplied by that scalar. Multiplying the input wave by 2 effectively doubles its volume. Then, the wave that is assigned to it is returned by main, meaning the results get either written to the specified output file or overwritten over the input file if no -o flag is specified in the command line. Note that while doublewave is an acceptable value to return since its type matches the return type of main, returning a value such as 2 (an integer) would not be acceptable.

4. Mixing Two Files Together

So you have a second file that you'd like to mix in with the input file. This is also a very straightforward operation in ASML.

```
fun wave main(wave secondwave)
   wave mixed = input + secondwave;
   return mixed;
end fun
```

This looks a little different from what we've dealt with before due to the parameter in main, but it's really not too different. The user will have to add an additional argument in the command line to specify the file that should be assigned to the secondwave parameter. From there, we have two waves that we're dealing with, input and secondwave. To mix these, simply add them together - this will return another wave that represents the mixing of the two. Then main returns this result.

5. Low-Pass Filter

Moving into the more advanced realms of digital signal processing (DSP), making something like a low-pass filter - a filter that will only let through low frequencies of a wave - is also very easy to make in ASML.

```
fun wave main()
   wave low = input at 20Hz to 500Hz;
```

```
   return low;
end fun
```

The at operator in this context is taking the range of frequencies from 20 to 500 Hz in input and returning only those frequencies to the new wave low. Main then returns low to the output file.

III. Language Reference Manual

1. Introduction

The purpose of Atomic Sound Manipulation Language is to manipulate sound waves (represented as .wav files) in order to make sound effects. To accomplish this, ASML uses a system of primitives that represent the fundamental components of sound (frequency, amplitude, time, and an overarching primitive type called wave), as well as standard programming control statements (conditionals and loops) to allow users to build algorithms. ASML also uses the concepts of functions and external references to allow the creation of effects libraries that can be reused or aggregated to form more complex and powerful effects. The end result is a C-like language that is specially geared to facilitate the creation and use of digital sound effect libraries.

This manual is intended to explain the structure of the language and to describe the basic usage of its operators, identifiers, expressions, and statements.

2. Lexical Conventions

A. Tokens

There are six classes of tokens in ASML: identifiers, keywords, constants, string literals, operators and separators (not including whitespace). Whitespace, including spaces, tabs, newlines, and form feeds, is ignored and used only to separate tokens. Comments are also ignored.

B. Comments

The character sequence /* introduces a comment, and the comment ends at the first occurrence of the character sequence */. Comments cannot exist within string literals.

C. Identifiers

An identifier is a sequence of letters, digits, and underscores. The first character in an identifier must be a letter or underscore, beyond that, the identifier can be any combination of the above characters. Identifiers are case sensitive, meaning the identifiers *tangerine* and *Tangerine* would represent two separate identifiers.

(Letter|'_')+ (Letter|'_'|Number)*;

D. Keywords

The following identifiers are reserved for use as keywords:

`ampl`	`for`	`print`
`at`	`freq`	`return`
`amplof`	`fun`	`time`
`else`	`if`	`to`
`end`	`include`	`wave`
`float`	`int`	`while`

E. Constants

There are five kinds of constants, described in detail below.

1) Integer constants: Whole numbers represented as a series of digits, not beginning with the digit 0 unless representing the number 0.

2) Float constants: These consist of an optional integer part followed by a decimal point and a series of digits of a minimum length of 1. There is no support for exponents in ASML.

3) Frequency constants: Represented by a float followed by the suffix Hz.
4) Amplitude constants: Represented by a float followed by the suffix a. *NOTE THAT THIS TYPE MUST CONTAIN A DECIMAL WHEN IT IS SPECIFIED.*
5) Time constants: Represented by a float followed by the suffix ms.

Type interaction in ASML is conducted scientifically- that is, operations between types are legal where the product scientifically yields another type in the language. For example, adding two frequencies are legal because scientifically, this results in a frequency. However, multiplying two frequencies is illegal because it results in a new unit: Hz^2, which does not make sense for this language. See below:

```
freq a = 5Hz;
freq b = 10Hz;
int i = 10;

freq c = a + b;
/*legal, c == 15Hz*/
freq c = a * b;
/*illegal! c == 50Hz2 - this doesn't make sense for our language and is a
Semantic Error.*/
freq c = a * i;
/*legal, c == 50Hz because of the propagation of units.*/
freq c = a + i;
/*illegal! 15Hz + 10 cannot be algebraically resolved any further, so this is
a Semantic Error.*/
```

Legal type interactions are described below in the section describing expressions.

F. String Literals
A string literal is a sequence of characters surrounded by double quotes ("). Special characters are denoted by an escape character \, and include \n for a newline, \t for a tab, and \" for a double quote. The terminating double quote must not be immediately preceded by a \ to avoid ambiguity with the escape character.

G. Operators
The following characters and character sequences are the set of operators in ASML, which have the same functionality as the C Programming Language.
+, -, *, /, %, = , <, >, <=, >=, !=, ==, ||, &&, at, to, amplof, !, - (negative)

H. Separators
The '(' and ')' characters surround expressions and argument lists in function calls. The ',' character separates arguments in a function definition and call. The ';' character is required to terminate all statements.

3. Program Flow
All ASML programs will consist of any number of include statements, which will signify external library files to be included in the program (note: any main functions found in external library files will not be included in the program), and a series of function definitions. If the program is to be executed, one of the defined functions must be called main.

Include statements are used to make functions defined in external files accessible to functions within the current program file. Include statements are the only structures that can appear outside of a function definition in an ASML program. The string argument of the include statement represents the local or absolute path name to the intended ASML file. After it is called, the entire include statement is replaced with the text of the referenced file with the exception of the referenced file's "main" function.

Users will have to be careful in naming their functions, so as not to create conflicts in referenced libraries.

Example: `include "reverb.asml";`

Implementation Note: while it is grammatically possible to have include statements in an ASML file, they currently do not do anything. This part of the language was dropped because of time constraints and its being less critical than the core functionality of the language. However, include statements were kept in the grammar for possible future implementation.

Functions are defined by the header "fun", a return type, zero or more comma-separated parameters bordered by parentheses, then a series of statements, terminating in the footer "end fun". All functions must return a value according to their return type. Once the function returns, control returns to the function that called it. If the main function returns, the program terminates.

Example:
```
fun wave main()
   freq f = 500Hz;
   /* stuff happens here */
   wave output = input at f;
   return output;
end fun
```

When an ASML program is run, the runtime engine will search for a function called main. Upon finding the main function, the engine will execute the statements in main in order. The main function has access to the wave value "input," which is supplied by the required -i value in the command line. The value entered for -i is the path to the wave file that is to be passed into the program. Further command line arguments are possible, and will be referred to by the formal arguments declared in the main function's header. The number of additional arguments in the command line must match the number of formal arguments in main. Variables must be declared before they are referenced, but functions can be declared anywhere in the same file and still reference each other. ASML does not support prototypes.

Main must always return a value of type `wave`, and must always have return type of `wave`. It will write the specified `wave` to the file specified by the -o option in the command line if one is given, otherwise it will overwrite the input file. Only .WAV files are supported- no conversion to other formats (i.e. MP3, OGG, AU) is supported by ASML at this time.

4. Meaning of Identifiers

Identifiers can refer to either functions or simple variables. The principal types of variables are ints, floats, waves, freqs, ampls, and times. Identifiers store type information as well as values and are used to track the scope of a value. ASML's rules for declaring variables, functions, and parameters, the meaning and interactions of types, scope rules, and rules concerning other language structures are discussed in further detail elsewhere in this document.

5. Expressions

A large component of ASML functionality is encompassed by expressions. There are many different types of expressions, and in order of precedence (from least to highest), they are:
Assignment expressions
Logical expressions ("&&": and, "||": or)
Relational expressions (<, >. <=, >=, ==, !=)
Addition/subtraction (+, -)
Multiplication/division/modulus (*, /, %)
Unary expressions (! signifying "not", - signifying negative, and the `amplof` operator)
At expressions (used to retrieve information from sound waves)
Function calls
Top expressions (parentheses, variables, and constants)

All expressions are left associative except for assignment expressions, which are right associative.

A. Assignment Expressions

Assignment expressions consist of a left hand side variable or an At Expression, followed by

the assignment operator, followed by a right-hand side expression. The left hand side of the expression will be set to the value to which the right hand side evaluates. The left and right hand sides of the assignment operator must evaluate to the same type.
Examples:
```
wave w1;
wave w2;
freq f;
time t;
ampl a;
f = 550Hz;
f = f + 100Hz;
t = 20000ms;
w1 at f = input at (f-100Hz); /*note, input is a reserved variable that always
holds a wave*/
a = amplof w1 at f at t;
w1 = w2 = input at f to 1000Hz; /*right associativity allows chaining of
assigns.*/
```

B. Logical Expressions
A logical expression refers to any expression where there is a relational or some higher precedence expression on either side of a logical operator (&&, ||). A logical expression resolves to a boolean integer value (1 or 0 - true or false, respectively). Both the left and right hand side expressions must resolve to an integer (non-zero is true, 0 is false) - if either does not, the logical expression will return a semantic error.
Examples:
```
int i = a == b && c >=d;
int j = i || 1;
```

C. Relational Expressions
A relational expression refers to an expression where there is an addition/subtraction or some higher precedence expression on either side of a relational operator (==, !=, <, >, <=, >=). A relational expression resolves to a boolean integer value (1 or 0 - true or false, respectively). Relational expressions are only legal between like types, except in the following cases: integers and floats can be compared, and waves cannot be compared to anything.
Examples:
```
int i = a < b;
int j = b != c;
```

D. Addition/Subtraction
Addition and subtraction operators are denoted by the + and - operators. For integer and float types, these work according to normal arithmetic rules. Values of type frequency, time, or amplitude may be added or subtracted by other values of like types only because addition and subtraction do not resolve algebraically between these different types.

The functionality when waves are added to waves is a little different. When a wave is added to another wave, the result is a third wave which is a "mix" of the original two waves, as if the two sound waves had been sent through a mixer on two different channels. For example:

```
wave band = drum + bass + guitar; /* wave band is now a mix of drum, bass, and
guitar */
```

Waves may also be added to integers or floats, which will essentially raise the amplitude by that scalar value. Waves can also subtract an integer or float which will have the opposite effect, however, waves may only be on the left hand side of a subtraction operation.

Finally, for use with the print statement only, string literals may be added to other string literals, integers, floats, times, frequencies, and amplitudes. Addition on a string, no matter what the other operands are, will return a string.

E. Multiplication/Division/Modulus
Multiplication and division operators are denoted by the * and / operators. For integer and float types, these work according to normal arithmetic rules. Values of type frequency, time, or amplitude may be multiplied or divided by floats and integers only (when frequencies, times, or amplitudes are multiplied or divided by floats and integers, the result will be a frequency, time, or amplitude, respectively). This is because multiplying and dividing these types by anything other than integers and floats scientifically results in new, unsupported types (see types section above for reference).

The modulus operator is used for integers only, and functions according to how it works in languages such as C and Java.

F. Unary Expressions
The unary '!' operator is used to reverse the logical value of an expression that evaluates to an integer. If an expression evaluates to a non-zero, the '!' operator changes the value to 0. Conversely, if an expression evaluates to 0, the operator changes the value to 1. The unary '-' operator is used to apply a negative to an integer, a float, or a wave (times, frequencies, and amplitude are illegal). For the arithmetic types it is the functional equivalent of multiplying the legal values by -1. When applied to a wave, it creates the inverse of the wave thus that:

```
wave w = input;
return w + -w;
```

returns a blank, soundless wave to the designated output file.

The amplof operator is used to determine the amplitude of an expression evaluating to a wave. For waves at a single point in time (e.g., `input at 10000ms`), this will evaluate to the amplitude at that point in time for that wave. For waves representing a range of time, this will evaluate to the maximum amplitude over that range. Examples:

```
ampl a = amplof input at 10000ms;
ampl b = amplof input at 650Hz at 20000ms to 25000ms;
```

G. At Expressions
The at operator is used to extract specific information from a wave given frequency or time information. Given a frequency, an at expression with a wave will return a wave with only that frequency. Given a time, an at expression with a wave will return a sample of that wave at that time. Examples:

```
/*Note that the type for input is always 'wave'*/
wave a = input at 600Hz; /* contains only frequency 600Hz */
wave b = input at 10000ms; /* contains sample of wave 10000 ms into the wave
*/
```

If a user wants to retrieve a range of frequencies or times, the to operator may be used in conjunction with the at operator:

```
wave a = input at 600Hz to 700Hz; /* contains frequencies 600-700Hz */
wave b = input at 10000ms to 20000ms; /* contains 10 seconds of data from 10-
20 seconds into the file */
```

At operators can be accreted as follows:

```
wave c = input at 600Hz at 10000ms;
    /*retrieves a sample from only frequency 600Hz at 10000ms into the wave*/
```

Another powerful feature of the at expression is the ability to only assign part of a previously initialized wave to another wave by putting an at expression on the left side of an assign:

```
wave a = input;
a at 250ms to 20250ms = input at 0ms to 20000ms;
return a + input;   /*adds a 1/4 second echo to input wave for the first 20
seconds */
```

Implementation note: this is currently only implemented when the at operation has time operands.

Implementation note: At is not currently functioning properly when used in conjunction with wave addition. Due to limitations in the underlying Java implementation which partially used the third-party library Tritonus, we were forced to do all wave manipulations via the read method in custom AudioInputStreams, which resulted in a lazy evaluation of the at operation, when we really needed an eager evaluation.

H. Function Calls

Function calls work as they would in a language like C or Java. Since functions are not required to return a value, but referring to their output when they do not return a value causes a semantic exception. Function call expressions evaluate to a literal with a type corresponding to the function's return type, and with a value corresponding to the evaluation of the expression in the function's return statement.
Example:

```
wave w = echo(input,2);
```

I. Top Expressions

The highest level of precedence is reserved for numbers, identifiers, and expressions surrounded by parentheses.

6. Declarations

ASML supports the following kinds of declarations:

1. Function Declarations: fun-decl
2. Variable Declarations: decl

The two basic kinds of declarations available in ASML are function declarations and variable declarations. Variable Declarations are a form of statement, but Function Declarations are a separate entity. Both kinds of declarations specify the intended type for an identifier (return type in the case of functions), but where function declarations require a definition as well, for variables definition is optional.

A. Function Declarations

fun-decl:
`fun` TYPE () block `fun`
`fun` TYPE (params) block `fun`

Function declarations are the only structure besides include statements that are allowed in the general space of the program. It is within their blocks that all other statements appear (including variable declarations). The TYPE in the function declaration represents the return type for the function, and the parentheses contain a list of parameters for the function (described below). Functions cannot be declared within other functions- the declarations can only occur in program space. After the block, the function is bounded by the `fun` keyword.

While ASML does not require a function to return a value- the lack of a default return value will result in a semantic exception if the caller tries to refer to its return value. If a return statement is specified, the type of the returned value must match the return type of the function in which it resides.

Functions in exterior files can only be accessed via "include" calls to other ASML program

files. The user must be careful to avoid name conflicts during such a transaction. Functions must be declared (and therefore also defined) before they can be referenced.
Implementation note: exterior libraries are not supported at this time.

A.1 Parameters

```
params:
  params , param
  param
param:
   TYPE ID ;
```

Params are the part of the function declaration that specifies the legal arguments that can be passed to the function when it is called. A calling function much pass actual arguments that match the amount and the types specified by the params.

B. Variable Declarations

```
decl:
   TYPE ID;
   TYPE ID = expr;
```

Variable declarations associate a storage type (e.g. int, float, etc.) with an identifier. Identifiers may be defined at declaration time, but they do not have to be. Identifiers must be declared via this mechanism before they can be referenced by any part of the program. In the case of the definition variety of decl, the expression must evaluate to the same type as the TYPE on the left hand side of the assignment operator.

7. Statements

ASML supports the following kinds of statements:

1. Variable Declarations: decl
2. Expressions: expr
3. Conditional (if): if-stmt
4. Iterative: while-stmt, for-stmt
5. Return statement: return-stmt
6. Print statement: print-stmt

Statements can only appear within a 'block.' Blocks themselves are simply a list of statements followed by an `end` keyword. Blocks are a part of function declarations as well as conditional and iterative statements. It is through the block mechanism that statements can be nested and how statements are otherwise bound to functions. One of the implications of statements being bound to blocks is that there are essentially no global variables - they are only defined within their own block or in nested blocks below them. This differs from both C and Java because while the latter language does not allow global variables, variables can be declared outside of functions (as member variables). This is not the case in ASML.

All statements besides conditionals and iteratives are terminated by semi colons. Conditional and iterative statements are terminated in ways similar to functions: with their primary keyword, e.g.: an if statement is terminated by the keyword `if`, and a while terminated by `while`. Coupled with the mandatory 'end' keyword that comes with blocks, it helps disambiguate the language in terms of grammar (handles the dangling else problem) and in terms of annotation. It is easier to to recognize the beginning of a statement that stops at an 'end if' combination than a generic 'end.'

Declarations and Expressions are discussed in detail in previous sections. The following is greater detail into the other 4 kinds of statements.

A. Declaration

As a statement, a declaration is a decl terminated by a semicolon.

B. Expression

As a statement, an expression is an expr terminated by a semicolon.

C. Conditional

if-stmt:

`if` (expr) block `else` block `if`

`if` (expr) block `if`

The expression must evaluate to an integer. In the first case, if the parenthetical expression evaluates to a non-zero, the first block is entered. However, if it evaluates to zero, the second block is entered. In the second case, the block is entered if and only if the expression evaluates to a non-zero. The expressions must always evaluate to an integer value. The standard way of using this statement would be to use the relational and logical operators described in section 5 above in the expression- these operators guarantee int results. Although this is the standard, arithmetic expressions that evaluate to any integer are allowed (as in C).

The overall if statement is bounded by a second if keyword, which acts to define the end of the `if` statement. For instance:

```
if(a == b)
  b = b + 1;
end
else
  b = b - 1;
end if
```

defines a single if-stmt where the final '`if`' terminates the statement. On the other hand, the following is illegal:

```
if(a == b)
  b = b + 1;
  end
if else
  b = b - 1;
  end
```

This does not work because the second `if` terminates the statement, and the `else` has no if-stmt with which to be associated. The presence of the terminating `if` clears up grammatical ambiguity as to which if-stmt an '`else`' belongs.

D. Iterative

while-stmt:

`while` (expr) block `while`

for-stmt:

`for` (expr; expr; expr) block `for`

In a while-stmt, ASML will enter the block if and only if the expression expr evaluates to an int that is non-zero. After the statements in the block are completely executed, control will jump back and check the expression again- continuing to do so until the expression evaluates to the int 0. It is expected of users to use the relational and logical operators in the expression, but any expression that evaluates to an integer will be accepted.

The for-stmt uses three expressions to create a loop: $expression_1$, $expression_2$, and $expression_3$ relative to the grammar above. $Expression_1$ is evaluated first and is only done once. $Expression_2$ is evaluated before reentering the block with each loop. $Expression_3$ is evaluated after all of the statements in the block are executed with every loop. The loop continues as long as the second expression evaluates to a number that is non-zero. The typical usage for these expressions would be to have the first one establish the loop counter, the second one evaluate whether the loop should continue, and the third to update the counter. However, apart from the restriction that the second expression must evaluate to an

integer, any expression may be entered in the other slots.

Both expressions are bounded by repeating the relevant key word while and for, respectively.

```
int evens = 0;
  int a;
for(a = 0; a < b; a = a + 1)
  if((a % 2) == 0)
     evens = evens + 1;
  end if
end for
```

E. Return

return-stmt:

```
return expr;
```

When the return statement is executed, control is returned to the place where the function in which it resides was called. The value of expr replaces the function call in the place where the call was made. When return is called from the 'main' function, the value contained in the expression is written to the output file. This is either specified by the -o command line argument or by the -i command line argument(default). The expression expr must evaluate to the type of the function where the return statement is called.

F. Print

print-stmt:

`print` expr;

The print statement is intended to be used for debugging purposes, and prints out string literals or the result of a value added to a string literal.

8. Scope

The scope of identifiers is based around the block structure; a block itself (as described elsewhere) is a group of statements bounded by the `end` keyword. An identifier has scope within the block in which it is declared, as well as in any nested sub-blocks, unless a sub-block contains a new declaration with the same identifier. Identifiers always represent the most local value.

9. Sample program

The following program implements the delay effect. The user inputs a delay time, say 500 ms, and length, say 30000ms, and main calls the delay function which copies the original wave into a new wave starting at position 500ms, and ending 30 seconds later. The original wave is then set equal to a mix of itself and the new wave, which starts 500 ms after the original starts, causing the newly mixed wave to have a delay effect.

```
fun wave delay(wave in, time del, time length)
   wave in2 at del to (del + length) = in at 0ms to length;
   in = in + in2;
   return in;
end fun

fun wave main(time del, time length)
   wave output = delay(input, del, length);
   return output;
end fun
```

(*Implementation note: This program currently would not work due to the aforementioned problem with at expressions not working in conjunction with the add operator. If this bug were resolved,*

this would be an excellent example of the potential of this language.)

IV. Project Plan

1. Engineering Processes

Our engineering process for the project was loosely based upon the Extreme Programming model. The primary tenets that we chose to follow were the use of paired programming, test driven development, automated unit testing, and a risk/reward based approach to requirements.

As stated we loosely followed these tenets. With respect to paired programming, we always made sure to code the core functionality of the system in pair: almost all of the modules from the asml.walker package were produced through paired programming, as well as all of the files written in ANTLR. The team would meet several times a week (and as the semester wore on, nightly) to discuss progress made in respective various stages of the project. The primary benefits of this approach were that we would not only be able to fit individual modules together more easily, but we would both have a better understanding of the project as a whole rather than our own limited pieces. There were, however, pieces that were done mostly in isolation- these were the DSP-heavy code (items such as the windowed sync method), done by Tim, and the JUnit tests and their test data, done by Frank.

A policy of automated unit testing was followed rigorously throughout the development process, but it was not always done in the traditional test driven manner. We would often conceptually define our test parameters before coding, but write the actual tests themselves after coding was complete. True test driven development requires the tests to be fully coded when the tested code is still at at most a skeletal stage, but we found that this approach often wasted time- especially if there was a problem at the design stage. This approach was used for the testing of SymbolTable and FunctionRecord, however and performed admirably. Otherwise tests were made after the fact but were still highly effective at uncovering issues at early stages.

Our risk/reward approach to requirements mostly came into play in the form of deciding which parts of our language would need to be cut for time. The approach was informal in that we did not set up any kind of points system and did not develop use cases or user stories to use as guidelines (in retrospect, these may have helped). This was due to the fact that we had to learn a lot about the capabilities of the Digital Signal Processing Java libraries (the open-source Tritonus library in conjunction with the standard Java Sound packages) during development and could not make informed decisions about these requirements at early stages. We did however, use the XP principle of leaving off high risk/low reward pieces of the project in our engineering process. Examples of code that were cut are: external library support, and wave operators such as multiplication (convolving). In the first case we decided that the core functionality needed to get done and was behind schedule so we decided to cut library functionality. In the second case we found that the primary functionality of convolve could be reproduced by making a function composed of other operators in our library and so we didn't need to create an operator for the purpose.

2. Programming Style

Our project was heavily influenced by extreme programming philosophies and to that end, our requirements were flexible, we engaged in pair programming frequently, and wrote many JUnit tests. In terms of the code itself, we stuck to a few conventions:

- Ensured that method/variable/class names made sense in terms of functionality and context - eschew declaring variables that have no contextual meaning (e.g. instead of declaring x to be the left hand side of an operation, we would declare lhs). This will make the code far more readable, and essentially allow the code to document itself in a manner of speaking.

- Organized code into packages based on functional area. We had four packages, all covering different areas of the language. The asml package handled the front-end materials - the loader, the grammar, both walkers, and the parser. The asml.walker package handled the back-end materials that aided Walker.g (which executes the program as an interpreter) - symbol tables, type abstractions, and related materials. The asml.walker.streams package handled pure Java DSP code used in asml.walker. Last, but certainly not least, was the asml.test package, which

contained our automated test suite materials.

- Declared variables with a one letter prefix implying where it was created. For example, a variable with the name mValue means it was declared as a member variable, a variable with the name aValue means it was declared as an argument, and a variable with the name tValue means it was declared as a temporary value (ie, inside a method).

- Where something in the code was not readily apparent as to what it did/why it was there, documented brief explanation before it using comments.

- Discussed major changes to common code with each other before uploading to our Subversion repository, and avoided uploading code to our SVN repository unless it compiled and it passed a basic functional testing.

3. Project Timeline

June 11th - White Paper Due
June 20th - Complete Lexer
June 27th - Language Reference Manual Due, Complete Parser
July 23rd - Walker Helper classes designed and implemented
July 30th - Wave and Wave helper classes designed and implemented
August 2nd - Function Walker completed
August 4th - Walker completed
August 5th - Development Complete, Begin Testing
August 10th - Project Due, Paper Due

a. Developer's Log
We kept a rigorous log of every single change made to every single file throughout our development time, however these are all stuck inside of our SVN repository. We though we could export (and cleanup by hand) these records from SVN, but we cannot. However, here is a link to our SVN if you would like to view our code and logs:
https://atomicsoundmanipulationlanguage.googlecode.com/svn/trunk

We can say authoritatively that we were able to stick by our rough timeline throughout development. During the month-long gap between June 27th and July 23rd we implemented and tested the AST, and began our design phase for the walker and walker support code. As you can see in the appendix, a lot of tests and support code were written during this time.

4. Roles and Responsibilities

Frank was the team lead, language architect, and tester - he was responsible for becoming familiar with ANTLR 3's many differences between previous versions of ANTLR, designed much of the control flow structure, and wrote the entire JUnit test suite.

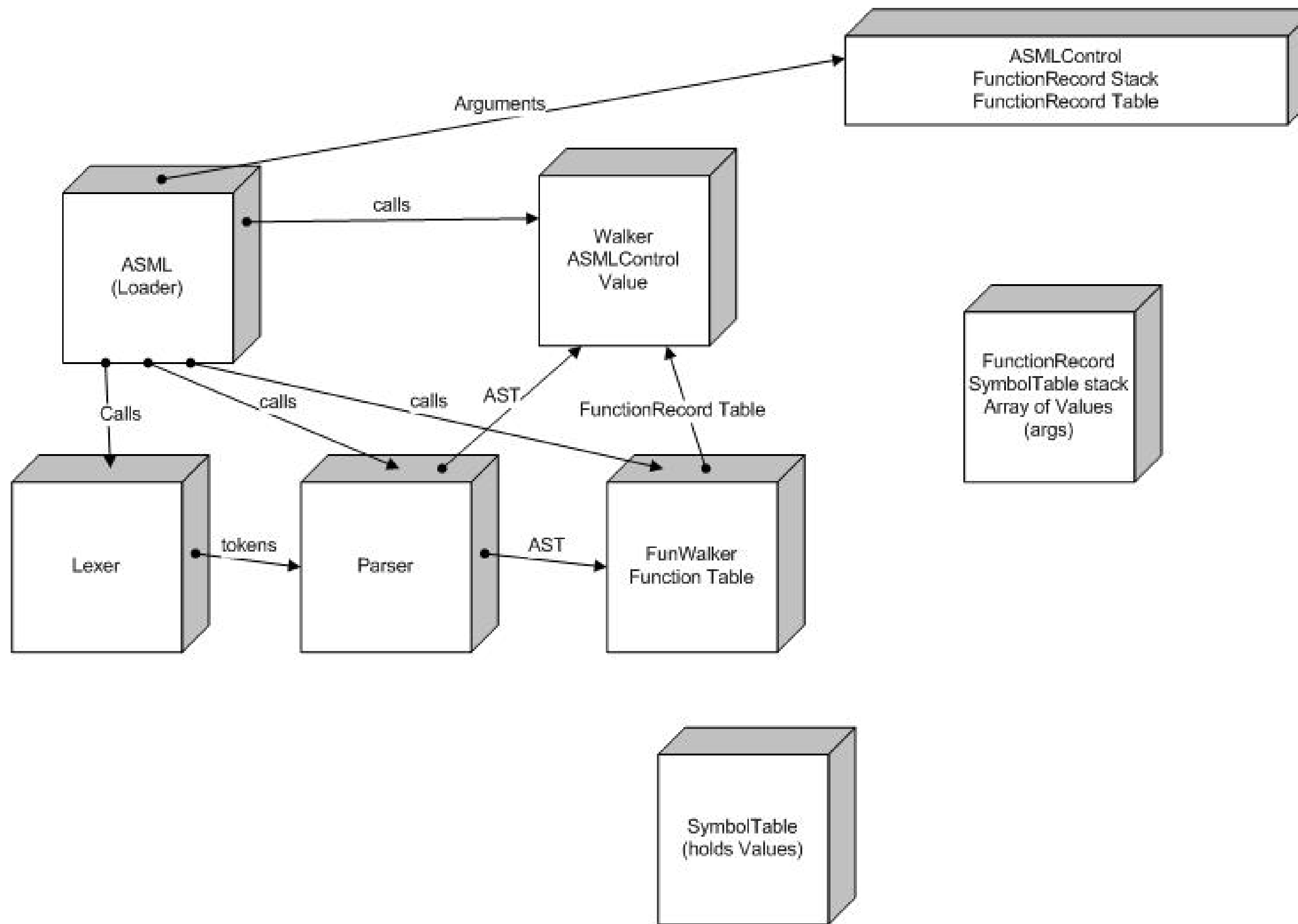
Tim was the DSP "guru" and documenter - he was responsible for researching the many intricacies of wave mechanics, researching implementations in Java, and documented much of the code.

Most of the development in the language with a few exceptions in control flow and DSP code was done using paired programming.

5. Project Environment and Tools

ASML was coded using Java version 1.5.0, using the Eclipse SDK (version 3.2). To generate lexer, parser, and walker code, we used ANTLR v. 3 and ANTLRWorks (varying versions between 1.0 and 1.1). For the DSP code, in addition to the standard Java Sound packages, we used the open-source Tritonus DSP packages (found here: http://sourceforge.net/projects/tritonus/). For the test suite, we used JUnit 3.8.1. Finally, for version control we used the Subversion repository provided by Google Code.

V. Architectural Design



1. Module Explanation and Interface guide.

a. Core Functionality

All of these modules were implemented during pair programming and were designed in tandem.

ASML- Essentially runs the program. Accepts command-line arguments pointing to the source code, input .wav, command-line arguments specified by the source code's main function, and the optional output argument. It feeds the program to the lexer, parser, FunWalker, and then passes the program on to the Walker for execution. Interfaces directly with all of the other Core Pieces of code and instantiates the ASMLControl support code.

ASMLLexer- lexer. Interfaces with the parser in the standard manner according to ANTLR3.

ASMLParser- parser. Interfaces with the lexer and the walkers in the standard manner according to ANTLR3.

ASMLFunWalker- This walker fills out the function record table and therefore interfaces heavily with the FunctionRecord class. See the FunctionRecord description for details on its purpose. FunWalker provides the function table that is used by ASMLControl to check the validity and aid in the execution of function calls.

ASMLWalker- This walker is the main engine of the interpreter as it performs all of the operations by walking through the AST produced by ASMLParser. It interfaces heavily with both Value and ASMLControl. ASMLControl aids the walker's execution of control flow statements as the program is executed. Value encapsulates the basic expression operations of the walker.

It does not flow straight through the source code, rather, it looks for the main function and

when it finds that, makes a call to ASMLControl.doCallMain(). ASMLWalker then executes statements as ASMLControl loads them onto the input stream.

b. Support Code
Implemented in pair, and designed by Frank.

Value- This is an abstract class that is made to represent all of the types of ASML. The modules that implement it are: ASMLAmplitude, ASMLError, ASMLFloat, ASMLFrequency, ASMLInteger, ASMLString, ASMLTime, and ASMLWave. These extensions abstract each respective type for interaction with the SymbolTable and each other. Value encapsulates all of the expression operations with its methods. example from expression rule of the walker:

```
...
| ^(MULT_OP lhs = expr rhs = expr){
     $v = lhs.multiply(rhs);}
...
```

The expression rule returns a Value (so that is the type of both lhs and rhs). The methods of Value consider their owner to be the left hand side of the operation they implement, and the value passed to them as the right hand side. Each different extension of Value overrides the operations where they are a legal left hand side. The default behavior in Value is to throw an exception. This way each implementation of Value enforces the proper semantic rules for its type- in the above example if the lhs is a frequency and the rhs is also a frequency, the ASMLFrequency module would issue an ASMLSemanticExpression on the call to multiply.
Value is the basis for the polymorphic enforcement of Semantic rules as well as the execution of expressions in the clean manner depicted above.

Value can only return a real value (e.g. the double that is encapsulated by the ASMLFloat module) when cast into its implementing type. For this purpose Value maintains a type field as well as other information that is useful to the symbol table (such as a name when Value represents an identifier).

Type- Type acts as an enumerated type and aids the Value module in casting to Specific types. There is a type field in Value, and this class is used to aid in its usage. When an ASMLInteger is instantiated, the constructor sets the underlying Value.mType to Type.INT;

SymbolTable- stores all of the identifiers available at the current scope level. To accomplish this it keeps a reference to its parent, which would be the symbol table of a higher scope. Therefore when an identifier is referenced it can check itself and then, if it doesn't find the identifier, its parent. This is implemented as a HashMap with a String key (the identifier's name) and a value of type Value. This Value is aware of its name as well as the value associated with that name- if the identifier is defined its isInstantiated member is 'true', otherwise if it's only declared, it is false.

The Symbol table acts as part of the function record.

FunctionRecord- The function record serves two purposes. The first purpose is to act as an abstraction of each function declaration in the program. Towards this end, it has fields for storing the name, return type, formal parameters and a pointer to the code block of a specific function. Each function record is created and added to a FunctionRecordTable by the FunWalker.

The second purpose is to keep track of the symbol table and allow the passage of arguments when a function is called. The caller passes an array of defined values that are compared to the list of declared values that the FunctionRecord got during the FunWalker's execution. This second phase is handled within ASMLControl, which maintains a stack of function calls. When a function call is made ASMLControl checks the FunctionRecordTable provided by the FunWalker and if the called function is found, creates a new instance of it, passes it its arguments and puts it on the stack.

In its second role the function record handles all calls to the symbol table. It does this by

maintaining a stack of SymbolTable objects with the symbol table of the current scope on top. All calls to the symbol table are directed to the top of the stack. When the walker enters a block a new symbol table is pushed onto the stack and when the block is exited that symbol table is popped off.

ASMLControl- The purpose of ASMLControl is to aid the ASMLWalker in its execution of code flow. To accomplish this task, it maintains a stack of active functions. The current active function is kept on the top of the stack. This is useful because ASMLControl can direct any calls to the current symbol table to the currently active function, which will consult its own symbol table stack.

Functions are pushed onto the stack when a function call is made, and are popped back off when the call is completed. When ASMLWalker encounters the main function, it calls ASML.doCallMain() which pushes the main function onto the stack and starts the flow of execution.

ASMLControl controls the execution through its maintenance of this function record stack, as well as its possession of the Function Record Table produced by the ASMLFunWalker module. As each function record contains a reference to its code block, ASMLControl can put this code into the stream of execution as calls are made. This, in effect, is the engine on which the ASML interpreter runs.

As this control over code flow is a major part of ASMLControl's purpose, we decided to also use it to hold code for executing the control flow code of various statements as well. The purpose of this was to neaten the ASMLWalker code which was already burdened by the need for try/catch blocks due to a well-known bug in ASML3's code with the "throws" statement. That is why ASMLControl has methods like .doFor(), .doIf(), and doReturn() among others.

VI. Test Plan

1. Overview

Because ASML was implemented as an interpreter, we cannot provide target language samples for test samples that we provide. Furthermore, the test suites that we created involve a large number of different programs that were included in large text files. These will be explained below with references to the files which we will include in our appendix (as they are hundreds of lines long apiece). However, we can use this space to detail the overall testing process that we followed throughout the development of ASML.

2. Unit Testing

Unit Testing was carried out using the automated harness provided by JUnit 3.8.1. Almost every source file in the program has an associated JUnit test file. These break down into two kinds: tests that read programs from token separated text files, and tests that rely on static data.

a. File Based

The file based test modules included in the appendix are: ASMLParserTest.java, ASMLASTTest.java, and ASMLFunWalkerTest.java. All three of these test modules read from formatted text files comprised of dozens of individual programs designed to test the specific parts of the source code. They refer to TestPrograms.txt, ASTPrograms.txt, and FunWalkerTest.txt, respectively. These tests were designed to allow testers to easily add new test programs as they thought of them to the respective files. All they had to do was write the test in and make sure that it was labeled appropriately or contained the right control data (depending on which harness was being used).

The parser tester's file is comprised of groups of good and bad programs for different areas of the grammar (e.g. programs featuring legal and illegal if statements). Running the

automated test harness puts the good programs through the parser and ensures that they are accepted by the parser. Then it runs the the bad programs through and assures that the parser throws exceptions against them. The groups are individually labeled in the source file in order to make the testing more granular.

The AST tester's file is comprised of almost all of the basic structures considered legal by our language (e.g. if statements with elses, nested if statements, if statements with and without expressions, etc.) followed by a string representation of how the tree should look (as a control statement). The test harness runs each individual program and compares the string representation of the parser's AST output to the control tree featured in the file. Any discrepancy results in a failure. The groups are individually labeled in the source file in order to make the testing more granular.

FunWalker's test is the simplest of all of these and simply allows a user to write tests to the file and label them. Funwalker consists of only one permanent test at the moment and it verifies that function records are properly created by the walker. There are two commented out cases that are intended to result in Semantic Errors- they must remain commented out unless tested individually because semantic errors cause the program to exit- and this results in JUnit being stopped as well- an unfortunate side effect.

Implementation Note: we would have had the walkers simply throw errors instead of handling them themselves, but ANTLR3 has a known bug that does not allow the 'throws' command to be used in a .g file.

b. Static

The static tests are more straightforward and are comprised of all of the functions in the test package not mentioned above. They essentially work by trying to find representative cases for a set of actions and providing positive and negative tests against them.

For example, ASMLIntegerTest.java is designed to test how ASMLInteger handles expression operators against every other kind of type. Therefore each operator is tested against each type with an ASMLInteger as the LHS. In cases where this is a legal operation, the result is checked for accuracy (e.g. integer + float, wave + wave). In cases where this is not a legal operation (e.g. integer + frequency) the harness checks that an ASMLSemanticExpression is thrown.

Another example would be the workings of FunctionRecordTest.java where we test whether a function record responds properly to having parameters passed into it- either updating its symbol table or throwing an ASMLSemanticException.

3. System Testing

Although integration testing is considered a critical part of the system testing process, it was not 'formally' conducted during our project. This is because integration testing was often carried out ad-hoc as part of unit testing. For instance, in order to test FunctionRecord.java at all, we had to test its integration with SymbolTable.java simply because they referred to each other. The individual type tests (e.g. ASMLIntegerTest.java and ASMLFloatTest.java) referred to each other as well- bringing a certain level of integration testing. Furthermore, the parser's automated unit tests relied on output from the lexer and the ast unit tests relied on output from the parser and so on down the line. So there are unit tests but they are quite informal.

System-wide end to end testing took place during the final two weeks of development as the most high level pieces of code were put down. These tests were run through the program loader ASML.java through a series of test files. Frank ran mostly structural tests- seeing if operators produced the correct output and exceptions when put through the interpreter itself. These tests can be seen in the appendix in the file TestDump.txt. Each individual program was put into an ASML source file that was fed to ASML.java along with a dummy .wav file and whatever arguments were necessary.

Tim's end to end tests involved making sure that the system actually manipulated files as expected. To

do so, he created a number of small programs that demonstrated each wave operation and fed them to the loader, and analyzed the output by ear. Further, more accurate automated testing was not conducted due to time constraints.

4. Testcase Criteria

Tests at the unit stage were generally chosen to verify positive and negative behavior of the different language elements in isolation. That is to say, if statements would be tested in every legal form and every illegal form that we could conceive in relative isolation. For completeness these tests would include one or two instances of mixing different statements (putting an if into a while for example). Where possible, 'borderline' or representative cases were found to test each aspect of the language.

End to end testing was very similar albeit less formalized due to time constraints. Generally, programs were written with an eye towards the sort of programming problems used in the classroom. One example that was conceived in order to test scoping rules is the following:

```
/* 4
 * Test for testing scope, declare, and assign*/
fun wave main()
   int i;

   i = 1;
   print "should be 1: " + i;
   if(1)
      int i=10;
      print "should be 10: " + i;
      i = i + 5;
      print "should be 15: " + i;
   end if

   i = i + 5;
   print "should be 6: " + i;
end fun
```

Other examples include a recursive fibonacci sequence program that was used to determine how well function calls were working.

5. Who did what

Frank: designed and implemented all automated test harnesses (e.g. all files in the asml.test package). Wrote all reference files to be used by automated test harnesses. Wrote the end to end tests featured in the TestDump.txt file (included in the appendix).

Tim: Reviewed automated test reference files. Wrote end-to-end tests for wave operations included in wavetests.txt (included in the appendix).

VII. Lessons Learned

Tim: Learned the value of prioritizing features. No software project is ever going to be 100% ideal, and compromises will inevitably have to be made for the sake of time and other factors. Also learned not to put off more difficult problems until later - might have had the at operator fully functioning had I tackled it sooner rather than later. I do wish I had taken this course in a different term (not the summer, with its compressed schedule) - had we had more time we could have implemented more of the features we wanted, not to mention better absorb the lessons designing this language gave.

Frank: The lesson that I hope I learned this time is that projects always seem easier to

accomplish at the very beginning of the semester. I am sad that we had to leave features out and have a few known bugs in the final version we are submitting, but we could not have possibly worked harder during the semester to accomplish this feat. I also think that we worked intelligently by having rigorous tests and catching a lot of tough bugs early. Unfortunately i think we were a little too ambitious with our original plans for something to be done during the summer semester.
Related to this, i think we needed to have our code freeze earlier than we did. Our code freeze was pushed back several days after it was already only supposed to be about 5 days or so. We could have used more time to both fix bugs and prepare this report.

VIII. Appendix

1a. ASML.java (Loader)

```
/**
 *
 */
package asml;

import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;

import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;

import asml.walker.ASMLControl;
import asml.walker.FunctionRecord;


/**
 * This class acts as the loader for the ASML language.  You call this class and
feed it the
 * program and the input data and it will spit out the right results.
 * @author Frank A. Smith and Tim Favorite
 *
 */
public class ASML {

   /**
    * @param args
    * -p: this specifies the *.asml program that is being run.  Required.
    * -i: this specifies the *.wav input file.  This file is accessible through the
"input"
    *     keyword in the program's main() function.  Required.
    * -o: this specifies the *.wav output file.  If this is not specified, it
    *     defaults to the specified input file.
    * unmarked arguments: will be taken in order and must conform to the argument
list for
    *     the program's main() function.
    */
   public static void main(String[] args) {
      if(args.length < 4)
         error("Insufficient arguments to run an ASML program", true);

      ArrayList<String> tUnspecdArgs = new ArrayList<String>();
      String tProgramFile = "";
      String tInputFile = "";
      String tOutputFile = "";
```

```
    String previousArg = "";

    for(int i=0; i<args.length; i++){
      if(args[i].equals("-p") || args[i].equals("-i") || args[i].equals("-o"))
        previousArg = args[i];
      else {
        if (previousArg.equals("-p")){
          tProgramFile = args[i];
          previousArg = "";
        } else if (previousArg.equals("-i")){
          tInputFile = args[i];
          previousArg = "";
        } else if (previousArg.equals("-o")){
          tOutputFile = args[i];
          previousArg = "";
        } else tUnspecdArgs.add(args[i]);
      }
    }

    if (tProgramFile.equals("")) error("No program file was specified.", true);
    if (tInputFile.equals("")) error("No input file was specified.", true);
    if (tOutputFile.equals("")) tOutputFile = tInputFile;

    checkFileName(".asml", tProgramFile);
    checkFileName(".wav", tInputFile);
    checkFileName(".wav", tOutputFile);

    try {
      callASML(tProgramFile, tInputFile, tOutputFile, tUnspecdArgs);
    } catch (RecognitionException e) {
      error(e.getMessage(), false);
    }
  }

  private static void callASML(String programFile, String inputFile, String
outputFile,
      ArrayList<String> unspecdArgs) throws RecognitionException {
    ANTLRFileStream input = null;

    //exit the program if it's a bad input file
    try {
      input = new ANTLRFileStream(programFile);
    } catch (IOException e) {
      error(e.getMessage(), true);
    }

    //Lexer
    ASMLLexer lexer = new ASMLLexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexer);

    //Parser and AST construction
    ASMLParser parser = new ASMLParser(tokens);
    ASMLParser.program_return result = parser.program();
    CommonTree t = (CommonTree)result.getTree();

    CommonTreeNodeStream nodes=new CommonTreeNodeStream(t);
```

```
    //Function Identification
    ASMLFunWalker f_walker = new ASMLFunWalker(nodes);
    f_walker.program();

    HashMap<String, FunctionRecord> tFunMap = f_walker.getFunctionTable();

    //Interperter
    nodes=new CommonTreeNodeStream(t);
    ASMLWalker e_walker = new ASMLWalker(nodes);
    ASMLControl control = new ASMLControl(tFunMap,
        unspecdArgs, inputFile, outputFile, e_walker);
    e_walker.setControl(control);
    e_walker.program();
  }

  private static void error(String msg, boolean aPrintMsg){
    if(aPrintMsg){
      System.err.println(msg);
      System.err.println("Usage: \n"+
          "-p: this specifies the *.asml program that is being run.  Required.\n"
+
          "-i: this specifies the *.wav input file.  This file is accessible
through the 'input'\n"+
          "      keyword in the program's main() function.  Required.\n" +
          "-o: this specifies the *.wav output file.  If this is not specified,
it\n" +
          "      defaults to the specified input file.\n" +
          "unmarked arguments: will be taken in order and must conform to the
argument list for\n" +
          "      the program's main() function. ");
    }
    System.exit(-1);
  }

  private static void checkFileName(String extension, String fileName){
    if (!fileName.endsWith(extension))
      error("File " + fileName + " is not of type ." + extension, true);
  }

}
```

1b. ASML.g (Lexer/Parser)

```
grammar ASML;

options{output = AST;
 ASTLabelType=CommonTree;}

tokens{ //tokens to act as artificial roots for structures wo/keywords or operators
    DECLRT;
    PARAMRT;
    CALLRT;
    BLOCKRT;
}
@header {package asml;}
@members{
    public boolean hasError = false;

    public void setHasError(boolean aHasError){
```

```
        hasError = aHasError;
    }

@Override
    public void reportError(RecognitionException e){
        super.reportError(e);
        hasError = true;
    }

    @Override
    protected void mismatch(IntStream input, int ttype, BitSet follow)
        throws RecognitionException{
        throw new MismatchedTokenException(ttype, input);
    }

    @Override
    public void recoverFromMismatchedSet(IntStream input,
        RecognitionException e, BitSet follow) throws RecognitionException{
        throw e;
    }
}

@rulecatch {
    catch (RecognitionException e) {
        reportError(e);
        throw e;
    }
}

@lexer::header {package asml;}
@lexer::members{
    public String stripEscapeChars(String in){
        String newStr = in.replaceAll("\\\\\"", "\"");

            char[] formSlashes = new char[newStr.length()];
            int charsAdded = 0;
            for(int i=0; i<newStr.length(); i++){
                if((newStr.charAt(i) == '\\') &&
                        (newStr.charAt(i+1) == '\\')){
                    formSlashes[charsAdded++] = newStr.charAt(++i);

                }//end if
                else{
                    formSlashes[charsAdded++] = newStr.charAt(i);

                }//end else
            }//end for
            newStr = new String(formSlashes);
            newStr = newStr.substring(0, charsAdded);

            return newStr;
    }
}

program    :    (include_stmt)*(fun_decl)+EOF!;

include_stmt
    :     INCLUDE^ STRING SEMI!;
fun_decl:    FUN TYPE ID LPARENS params? RPARENS block FUN -> ^(FUN TYPE ID params?
block);

block    :    stmt* END     -> ^(BLOCKRT stmt*);

//stmts     :    (stmt stmts)?;
```

```
stmt    :    decl SEMI!
     |    expr SEMI!
     |    if_stmt
     |    while_stmt
     |    for_stmt
     |    print_stmt
     |    return_stmt;

if_stmt    :    IF^ LPARENS! expr RPARENS! block (ELSE! block)? IF!;

for_stmt:    FOR^ LPARENS! expr SEMI! expr SEMI! expr RPARENS! block FOR!;
while_stmt
    :    WHILE^ LPARENS! expr RPARENS! block WHILE!;
return_stmt
    :    RETURN^ expr SEMI!;
print_stmt:    PRINT^ expr SEMI!;


params    :    param (COMMA! params)?;
param    :    TYPE ID        -> ^(PARAMRT TYPE ID);
decl    :
    TYPE ID (ASSIGN expr)?    -> ^(DECLRT TYPE ID expr?);

expr_list
    :    expr (COMMA! expr_list)?;
expr    :    log_expr (ASSIGN^ expr)?;
log_expr:    rel_expr (LOG_OP^ rel_expr)*;
rel_expr:    add_expr (REL_OP^ add_expr)*;
add_expr:    mult_expr ((ADD_OP^ | SUB_OP^) mult_expr)*;
mult_expr
    :    unary_expr ((MULT_OP^ | DIV_OP^ | MOD_OP^) unary_expr )*;
unary_expr
    :    ('!'^ | '-'^ | AMPLOF^)? at_expr;
at_expr    :    fun_call (AT^ fun_call (TO! fun_call)?)*;

fun_call:    ID LPARENS expr_list? RPARENS     -> ^(CALLRT ID expr_list?)
        | top_expr            -> top_expr;

top_expr:    LPARENS! expr RPARENS! | NUMBER | STRING | ID;



COMMENT    :    '/*' (options{greedy = false;}: .)* '*/'{skip();};

ADD_OP    :    '+';
SUB_OP    :    '-';
MULT_OP    :    '*';
DIV_OP    :    '/';
MOD_OP    :    '%';
ASSIGN    :    '=';

REL_OP    :    '<' | '>' | '<=' | '>=' | '==' | '!=';
LOG_OP    :    '||' | '&&';



LPARENS    :    '(';
RPARENS    :    ')';
COMMA    :    ',';
SEMI    :    ';';

fragment LETTER
    :    ('a'..'z')|('A'..'Z');
fragment DIGIT
```

```
    :    ('0'..'9');
fragment INTEGER
    :    (DIGIT)+;
fragment FRAC
    :    '.'(INTEGER);

/*CONSTANT:    NUMBER;*/

NUMBER     :    INTEGER /*set type to int*/
        ('Hz'/*set type to freq*/
        |'ms'/*set type to time*/)?
    | INTEGER? FRAC /*set type to float*/
        ('a' /*set type to ampl*/
        |'Hz'/*set type to freq*/
        |'ms'/*set type to time*/)?;


TYPE    :    'ampl'|'float'|'freq'|'int'|'time'|'wave';

/*fragment STR_QUOTE
    :    '\\' '"' { setText("\"");};
fragment STR_BACKSLASH
    :    '\\' '\\' { setText("\\");};*/
fragment STR_CONTENT
    :    (('\\''"') | ('\\''\\') | ~('"'|'\\'))*;
STRING    :    '"' t = STR_CONTENT '"'
{setText(stripEscapeChars($STR_CONTENT.text));};

WS    :    (' ' | '\t' | '\n' | '\r')+ {skip();};


AMPLOF    :    'amplof';
AT    :    'at';
ELSE    :     'else';
END    :    'end';
FOR    :    'for';
FUN    :    'fun';
IF    :    'if';
INCLUDE    :    'include';
PRINT    :    'print';
RETURN    :    'return';
TO    :    'to';
WHILE    :    'while';

ID    :     (LETTER|'_')(LETTER|'_'|DIGIT)*;
```

1c. ASMLFunWalker.g (Function Walker)

```
tree grammar ASMLFunWalker;

options {
tokenVocab=ASML; // reuse token types
ASTLabelType=CommonTree; // $label will have type CommonTree
}
@header {
package asml;
import asml.walker.*;
import java.util.HashMap;
}

@members {
HashMap<String, FunctionRecord> FunctionTable =
    new HashMap<String, FunctionRecord>();
```

```
public HashMap<String, FunctionRecord> getFunctionTable(){
    return FunctionTable;
}
}

program    :
    (include_stmt)*(fun_decl)+;

include_stmt:
    ^(INCLUDE STRING);

fun_decl
@init{
ArrayList<Value> formalParams = new ArrayList<Value>();
int blockIndex = 2;
}
@after{
if(FunctionTable.containsKey($name.text)){
    System.err.println("Function '" + $name.text +
        "' cannot be declared more than once in a program.");
    System.exit(-1);
}

if(($name.text.equals("main")) && !($type.text.equals("wave"))){
    System.err.println("Function 'main' must return a wave value.");
    System.exit(-1);
}

CommonTree blockRt = (CommonTree)$fun_decl.start.getChild(blockIndex);

int funType = Type.WAVE;
if($type.text.equals("int"))
    funType = Type.INT;
else if($type.text.equals("ampl"))
    funType = Type.AMPL;
else if($type.text.equals("float"))
    funType = Type.FLOAT;
else if($type.text.equals("freq"))
    funType = Type.FREQ;
else if($type.text.equals("time"))
    funType = Type.TIME;

FunctionRecord newFunRec = null;
try{
newFunRec = new FunctionRecord(
    funType, $name.text, formalParams, blockRt);
}catch(ASMLSemanticException e){
    System.err.println(e.getMessage());
    System.exit(-1);
}

FunctionTable.put($name.text, newFunRec);
}
:^(FUN type=TYPE name=ID (par=param
        {formalParams.add(par);
         blockIndex++;})* .);

param returns [Value v]:
    ^(PARAMRT type=TYPE name=ID){
    if($type.text.equals("int"))
        return new ASMLInteger($name.text);
    else if($type.text.equals("ampl"))
        return new ASMLAmplitude($name.text);
    else if($type.text.equals("float"))
```

```
        return new ASMLFloat($name.text);
    else if($type.text.equals("freq"))
        return new ASMLFrequency($name.text);
    else if($type.text.equals("time"))
        return new ASMLTime($name.text);
    else if($type.text.equals("wave"))
        return new ASMLWave($name.text);};
```

1d. ASMLWalker.g (Walker)

```
tree grammar ASMLWalker;

options {
tokenVocab=ASML; // reuse token types
ASTLabelType=CommonTree; // $label will have type CommonTree
}
@header {
package asml;
import asml.walker.*;
}

@members{
CommonTreeNodeStream stream = (CommonTreeNodeStream)input;
ASMLControl control = null;
public void setControl(ASMLControl aControl){
    control = aControl;
}
}

program
@init{
control.setStream(stream);
}
:(include_stmt)*(fun_decl)+;

include_stmt:
    ^(INCLUDE STRING);

fun_decl
@after{
try{
    if($name.text.equals("main"))
        control.doCallMain();
}catch(ASMLSemanticException e){
    System.err.println(e.getMessage());
    System.exit(-1);}
}
:^(FUN TYPE name=ID param* .);

block
@init{
try{control.enterScope();}
catch(ASMLSemanticException e){
    System.err.println(e.getMessage());
    System.exit(-1);
}}
@after{
try{control.exitScope();}
catch(ASMLSemanticException e){
    System.err.println(e.getMessage());
    System.exit(-1);
}}
:^(BLOCKRT stmt*);
```

```
stmt
@init{
    try{
    if(control.isCurrentFunctionLocked()){
        matchAny(input);
        return;
    }
    }catch(java.util.EmptyStackException ignore){}
}
:      decl
    | expr
    | if_stmt
    | while_stmt
    | for_stmt
    | print_stmt
    | return_stmt;

if_stmt
@after{
if(eval.getType() != Type.INT){
    System.err.println("Semantic exception: Expressions for conditional statements
must evaluate to an int.");
    System.exit(-1);
    }
int evalTo = 0;
try{evalTo = ((ASMLInteger)eval).getValue();}
catch(Exception e){
    System.err.println(e.getMessage());
    System.exit(-1);
    }

if(evalTo != 0){
    CommonTree block1=(CommonTree)$if_stmt.start.getChild(1);
    stream.push(stream.getNodeIndex(block1));
    block();
    stream.pop();
    }
else if($if_stmt.start.getChildCount() == 3){
    CommonTree block2=(CommonTree)$if_stmt.start.getChild(2);
    stream.push(stream.getNodeIndex(block2));
    block();
    stream.pop();
    }
}
:^(IF eval=expr . .?);

while_stmt
@after{
if(eval.getType() != Type.INT){
    System.err.println("Semantic exception: Expressions for conditional statements
must evaluate to an int.");
    System.exit(-1);
    }

int evalTo = 0;
try{evalTo = ((ASMLInteger)eval).getValue();}
catch(Exception e){
    System.err.println(e.getMessage());
    System.exit(-1);
    }
if(evalTo != 0){
    CommonTree tExpr=(CommonTree)$while_stmt.start.getChild(0);
    CommonTree tBlock=(CommonTree)$while_stmt.start.getChild(1);
```

```
    try{control.doWhile(tExpr, tBlock);}
    catch(Exception e){
        System.err.println(e.getMessage());
        System.exit(-1);
        }
    }
}
:^(WHILE eval=expr .);

for_stmt
@after{
if(eval.getType() != Type.INT){
    System.err.println("Semantic exception: Expressions for conditional statements
must evaluate to an int.");
    System.exit(-1);
    }

int evalTo = 0;
try{evalTo = ((ASMLInteger)eval).getValue();}
catch(Exception e){
    System.err.println(e.getMessage());
    System.exit(-1);
    }

if(evalTo != 0){
    CommonTree tEval=(CommonTree)$for_stmt.start.getChild(1);
    CommonTree tExec=(CommonTree)$for_stmt.start.getChild(2);
    CommonTree tBlock=(CommonTree)$for_stmt.start.getChild(3);
    try{control.doFor(tEval, tExec, tBlock);}
    catch(Exception e){
        System.err.println(e.getMessage());
        System.exit(-1);
        }
    }
}
:^(FOR expr eval=expr . .);

print_stmt
    :
    ^(PRINT val = expr){
        try {
            ASMLString str = new ASMLString(val);
            System.out.println(str.getValue());
        } catch (ASMLSemanticException e){
            System.err.println("Print: expression must evaluate to a string.");
            System.exit(-1);
        }
    };
return_stmt
@after{
try{control.doReturn(retval);}
catch(Exception e){
    System.err.println(e.getMessage());
    System.exit(-1);
    }
}
:^(RETURN retval=expr);

decl
@after{
try{
    if($decl.start.getChildCount() == 3)
        control.doDeclare($name.text, $type.text, rhs);
    else
```

```
            control.doDeclare($name.text, $type.text);
        }
    catch(ASMLSemanticException e){
        System.err.println(e.getMessage());
        System.exit(-1);
        }
    }
    :^(DECLRT type=TYPE name=ID (rhs = expr)?);
    /*params    :
        param ( params)?;*/
    param    :
        ^(PARAMRT TYPE ID);

    expr returns [Value v]
    @init{
        ArrayList<Value> aParams = new ArrayList<Value>();
        boolean hasSecondExpr = false;
    }:
          ^(ASSIGN lhs = expr rhs = expr){
              try{$v = control.doAssign(lhs, rhs);}
            catch(ASMLSemanticException e){
                System.err.println(e.getMessage());
                System.exit(-1);
            }
          }
        | ^(LOG_OP lhs = expr rhs = expr){
            try{$v = lhs.logic(rhs, $LOG_OP.text);}
            catch(ASMLSemanticException e){
                System.err.println(e.getMessage());
                System.exit(-1);
            }
        }
        | ^(REL_OP lhs = expr rhs = expr){
            try{$v = lhs.relate(rhs, $REL_OP.text);}
            catch(ASMLSemanticException e){
                System.err.println(e.getMessage());
                System.exit(-1);
            }
        }
        | ^(ADD_OP lhs = expr rhs = expr){
            try{$v = lhs.add(rhs);}
            catch(ASMLSemanticException e){
                System.err.println(e.getMessage());
                System.exit(-1);
            }
        }
        | ^(SUB_OP lhs = expr (rhs=expr{hasSecondExpr = true;})?){
            try{
                if(hasSecondExpr)
                    $v = lhs.subtract(rhs);
                else
                    $v = lhs.negate();}
            catch(ASMLSemanticException e){
                System.err.println(e.getMessage());
                System.exit(-1);
            }
        }
        | ^(MULT_OP lhs = expr rhs = expr){
            try{$v = lhs.multiply(rhs);}
            catch(ASMLSemanticException e){
                System.err.println(e.getMessage());
                System.exit(-1);
            }
        }
```

```
| ^(DIV_OP lhs = expr rhs = expr){
    try{$v = lhs.divide(rhs);}
    catch(ASMLSemanticException e){
        System.err.println(e.getMessage());
        System.exit(-1);
    }
}
| ^(MOD_OP lhs = expr rhs = expr){
    try{$v = lhs.mod(rhs);}
    catch(ASMLSemanticException e){
        System.err.println(e.getMessage());
        System.exit(-1);
    }
}
| ^(AMPLOF lhs = expr){
    try{$v = lhs.amplof();}
    catch(ASMLSemanticException e){
        System.err.println(e.getMessage());
        System.exit(-1);
    }
}
| ^('!' lhs=expr){
    try{$v = lhs.not();}
    catch(ASMLSemanticException e){
        System.err.println(e.getMessage());
        System.exit(-1);
    }
}
| ^(AT wv=expr ex1=expr (ex2=expr{hasSecondExpr = true;})?){
    try{
        if(hasSecondExpr)
            $v = control.doAt(wv, ex1, ex2);
        else
            $v = control.doAt(wv, ex1);
    }
    catch(ASMLSemanticException e){
        System.err.println(e.getMessage());
        System.exit(-1);
    }
}
| ^(CALLRT name=ID (par=expr{aParams.add(par);})*){
    try{
        $v = control.doCallFunction($name.text, aParams);
    } catch(ASMLSemanticException e){
        System.err.println(e.getMessage());
        System.exit(-1);
    }
}
| ID{
    try {
        $v = control.getSymbol($ID.text);
    } catch(ASMLSemanticException e){
        System.err.println(e.getMessage());
        System.exit(-1);
    }
}
| NUMBER{$v = Value.valueOf($NUMBER.text);}
| STRING{$v = new ASMLString($STRING.text);};
```

2a. ASMLAllTests.java

package asml.test;

import junit.framework.TestCase;

```
import junit.framework.TestSuite;
import junit.framework.Test;
/**
 *
 * @author Frank Smith
 *
 */
public class ASMLAllTests extends TestCase {

   public static Test suite () {
      TestSuite suite = new TestSuite("Automated tests for ASML");
      //$JUnit-BEGIN$
      suite.addTestSuite(ASMLLexerTest.class);
      suite.addTestSuite(ASMLParserTest.class);
      suite.addTestSuite(ASMLASTTest.class);
      suite.addTestSuite(ASMLFloatTest.class);
      suite.addTestSuite(ASMLIntegerTest.class);
      suite.addTestSuite(ASMLFrequencyTest.class);
      suite.addTestSuite(ASMLAmplitudeTest.class);
      suite.addTestSuite(ASMLTimeTest.class);
      suite.addTestSuite(ASMLStringTest.class);
      suite.addTestSuite(SymbolTableTest.class);
      suite.addTestSuite(FunctionRecordTest.class);
      suite.addTestSuite(ASMLFunWalkerTest.class);

      return suite;
   }

}
```

2b. ASMLAllWalkerTests.java

```
package asml.test;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
/**
 *
 * @author Frank Smith
 *
 */
public class ASMLAllWalkerTests extends TestCase {

   public static Test suite () {
      TestSuite suite = new TestSuite("Automated tests for ASML");
      //$JUnit-BEGIN$
      suite.addTestSuite(ASMLFloatTest.class);
      suite.addTestSuite(ASMLIntegerTest.class);
      suite.addTestSuite(ASMLFrequencyTest.class);
      suite.addTestSuite(ASMLAmplitudeTest.class);
      suite.addTestSuite(ASMLTimeTest.class);
      suite.addTestSuite(ASMLStringTest.class);
      suite.addTestSuite(SymbolTableTest.class);
      suite.addTestSuite(FunctionRecordTest.class);

      return suite;
   }
}
```

2c. ASMLAmplitudeTest.java

```java
package asml.test;

import asml.walker.ASMLAmplitude;
import asml.walker.ASMLFloat;
import asml.walker.ASMLFrequency;
import asml.walker.ASMLInteger;
import asml.walker.ASMLSemanticException;
import asml.walker.ASMLString;
import asml.walker.ASMLTime;
import asml.walker.Type;
import asml.walker.Value;
import junit.framework.TestCase;

/**
 *
 * @author Frank Smith
 *
 */
public class ASMLAmplitudeTest extends TestCase {
   ASMLAmplitude mLHS;

   public ASMLAmplitudeTest(String name) {
      super(name);
   }

   protected void setUp() throws Exception {
      super.setUp();
      mLHS = new ASMLAmplitude(5.0);
   }

   public void testConstructors() throws ASMLSemanticException{
      ASMLAmplitude tAmp;

      //un-named value
      tAmp = new ASMLAmplitude(5.0);
      assertEquals(5.0, tAmp.getValue());
      assertEquals(Type.AMPL, tAmp.getType());
      assertTrue(tAmp.isInitialized());
      assertFalse(tAmp.isStorable());

      //declared, undefined
      tAmp = new ASMLAmplitude("test");
      assertEquals("test", tAmp.getName());
      assertEquals(Type.AMPL, tAmp.getType());
      assertFalse(tAmp.isInitialized());
      assertTrue(tAmp.isStorable());

      //declared, defined
      tAmp = new ASMLAmplitude(5, "test");
      assertEquals(5.0, tAmp.getValue());
      assertEquals("test", tAmp.getName());
      assertEquals(Type.AMPL, tAmp.getType());
      assertTrue(tAmp.isInitialized());
      assertTrue(tAmp.isStorable());
   }

   public void testAdd() {
      Value tRHS, tResult;
      String tRHSType = "";
```

```
        try {
            //Amplitudes
            tRHSType = "Amplitude rhs";
            tRHS = new ASMLAmplitude(3.5);
            tResult = mLHS.add(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.AMPL, tResult.getType());
            assertEquals(8.5, ((ASMLAmplitude)tResult).getValue());

            //Strings
            tRHSType = "string rhs";
            tRHS = new ASMLString(" times a lady");
            tResult = mLHS.add(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.STRING, tResult.getType());
            assertEquals("5.0a times a lady", ((ASMLString)tResult).getValue());
        } catch (ASMLSemanticException e) {
            fail("Legal Ops throw Semantic Exception: " +
                    tRHSType + " " + e.getMessage());
        }

        //Illegal operations
        Value tMismatches[] = {
            new ASMLTime(1), new ASMLFloat(1),
            new ASMLInteger(1), new ASMLFrequency(1)};
        int numFails = 0;
        for(int i=0; i<tMismatches.length; i++){
            try {
                tResult = mLHS.add(tMismatches[i]);
                fail("exception not thrown for mismatch: " + i);
            } catch (ASMLSemanticException e) {
                numFails++;
            }
        }
        assertEquals(tMismatches.length, numFails);
    }

    public void testSubtract() {
        Value tRHS, tResult;
        String tRHSType = "";

        try {
            //Amplitudes
            tRHSType = "amplitude rhs";
            tRHS = new ASMLAmplitude(3.5);
            tResult = mLHS.subtract(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.AMPL, tResult.getType());
            assertEquals(1.5, ((ASMLAmplitude)tResult).getValue());
        } catch (ASMLSemanticException e) {
            fail("Legal Ops throw Semantic Exception: " +
                    tRHSType + " " + e.getMessage());
        }

        //Illegal operations
        Value tMismatches[] = {
            new ASMLTime(1), new ASMLFloat(1), new ASMLString("1"),
            new ASMLInteger(1), new ASMLFrequency(1)};
        int numFails = 0;
        for(int i=0; i<tMismatches.length; i++){
```

```
        try {
            tResult = mLHS.subtract(tMismatches[i]);
            fail("exception not thrown for mismatch: " + i);
        } catch (ASMLSemanticException e) {
            numFails++;
        }
    }
    assertEquals(tMismatches.length, numFails);
}

public void testMultiply() {
    Value tRHS, tResult;
    String tRHSType = "";

    try {
        //Integers
        tRHSType = "integer rhs";
        tRHS = new ASMLInteger(3);
        tResult = mLHS.multiply(tRHS);
        assertNotNull(tResult);
        assertEquals(Type.AMPL, tResult.getType());
        assertEquals(15.0, ((ASMLAmplitude)tResult).getValue());

        //Floats
        tRHSType = "float rhs";
        tRHS = new ASMLFloat(3.0);
        tResult = mLHS.multiply(tRHS);
        assertNotNull(tResult);
        assertEquals(Type.AMPL, tResult.getType());
        assertEquals(15.0, ((ASMLAmplitude)tResult).getValue());

        //Amplitudes
        tRHSType = "amplitude rhs";
        tRHS = new ASMLAmplitude(3.0);
        tResult = mLHS.multiply(tRHS);
        assertNotNull(tResult);
        assertEquals(Type.AMPL, tResult.getType());
        assertEquals(15.0, ((ASMLAmplitude)tResult).getValue());
    } catch (ASMLSemanticException e) {
        fail("Legal Ops throw Semantic Exception: " +
                tRHSType + " " + e.getMessage());
    }

    //Illegal operations
    Value tMismatches[] = {
        new ASMLString("1"), new ASMLTime(1), new ASMLFrequency(1)};
    int numFails = 0;
    for(int i=0; i<tMismatches.length; i++){
        try {
            tResult = mLHS.multiply(tMismatches[i]);
            fail("exception not thrown for mismatch: " + i);
        } catch (ASMLSemanticException e) {
            numFails++;
        }
    }
    assertEquals(tMismatches.length, numFails);
}

public void testDivide() {
    Value tRHS, tResult;
```

```
    String tRHSType = "";

    try {
       //Integers
       tRHSType = "integer rhs";
       tRHS = new ASMLInteger(2);
       tResult = mLHS.divide(tRHS);
       assertNotNull(tResult);
       assertEquals(Type.AMPL, tResult.getType());
       assertEquals(2.5, ((ASMLAmplitude)tResult).getValue());

       //Floats
       tRHSType = "float rhs";
       tRHS = new ASMLFloat(2.0);
       tResult = mLHS.divide(tRHS);
       assertNotNull(tResult);
       assertEquals(Type.AMPL, tResult.getType());
       assertEquals(2.5, ((ASMLAmplitude)tResult).getValue());

       //Amplitudes
       tRHSType = "Amplitude rhs";
       tRHS = new ASMLAmplitude(2.0);
       tResult = mLHS.divide(tRHS);
       assertNotNull(tResult);
       assertEquals(Type.AMPL, tResult.getType());
       assertEquals(2.5, ((ASMLAmplitude)tResult).getValue());
    } catch (ASMLSemanticException e) {
       fail("Legal Ops throw Semantic Exception: " +
             tRHSType + " " + e.getMessage());
    }

    //Illegal operations
    Value tMismatches[] = {
       new ASMLTime(1), new ASMLString("1"),
       new ASMLFrequency(1)};
    int numFails = 0;
    for(int i=0; i<tMismatches.length; i++){
       try {
          tResult = mLHS.divide(tMismatches[i]);
          fail("exception not thrown for mismatch: " + i);
       } catch (ASMLSemanticException e) {
          numFails++;
       }
    }
    assertEquals(tMismatches.length, numFails);
}

public void testRelate() {
    Value tRHS, tResult;

    String tOps[] = {"<", "<=", ">", ">=", "==", "!="};
    double compareF[] = {4.9, 5.0, 5.1};
    int tResult4[] = {0, 0, 1, 1, 0, 1};
    int tResult5[] = {0, 1, 0, 1, 1, 0};
    int tResult6[] = {1, 1, 0, 0, 0, 1};
    int tResults[][] = {tResult4, tResult5, tResult6};

    int i=-1, j=-1;

    //Legal operations
```

```
try {
   //Amplitudes
   for(i=0; i<compareF.length; i++){
      tRHS = new ASMLAmplitude(compareF[i]);
      for(j=0; j<tOps.length; j++){
         tResult = mLHS.relate(tRHS, tOps[j]);
         assertNotNull(tResult);
         assertEquals(Type.INT, tResult.getType());
         assertEquals(tResults[i][j], ((ASMLInteger)tResult).getValue());
      }
   }
} catch (ASMLSemanticException e) {
   fail("Legal Ops throw Semantic Exception:" + " i: " + i + " j: " + j +
         e.getMessage());
}

//Illegal operations
Value tMismatches[] = {new ASMLTime(1), new ASMLFloat(1),
      new ASMLInteger(1), new ASMLFrequency(1),
      new ASMLString(" times a lady")};
int numFails = 0;
for(i=0; i<tMismatches.length; i++){
   try {
      tResult = mLHS.relate(tMismatches[i], "<");
      fail("exception not thrown for mismatch: " + i);
   } catch (ASMLSemanticException e) {
      numFails++;
   }
}
assertEquals(tMismatches.length, numFails);

try {
   tResult = mLHS.relate(new ASMLAmplitude(1), "q");
   fail("exception not thrown for bad op");
} catch (ASMLSemanticException e) {
}
}

public void testMod() {
   //Illegal operations
   Value tMismatches[] = {new ASMLFloat(1), new ASMLAmplitude(1),
         new ASMLFrequency(1), new ASMLTime(1),
         new ASMLString(" times a lady"), new ASMLInteger(1),};
   int numFails = 0;
   for(int i=0; i<tMismatches.length; i++){
      try {
         mLHS.mod(tMismatches[i]);
         fail("exception not thrown for mismatch: " + i);
      } catch (ASMLSemanticException e) {
         numFails++;
      }
   }
   assertEquals(tMismatches.length, numFails);
}

public void testLogic() {
   //Illegal operations
   Value tMismatches[] = {new ASMLFloat(1), new ASMLAmplitude(1),
         new ASMLFrequency(1), new ASMLTime(1),
         new ASMLString(" times a lady"), new ASMLInteger(1),};
```

```
        int numFails = 0;
        for(int i=0; i<tMismatches.length; i++){
            try {
                mLHS.logic(tMismatches[i], "||");
                fail("exception not thrown for mismatch: " + i);
            } catch (ASMLSemanticException e) {
                numFails++;
            }
        }
        assertEquals(tMismatches.length, numFails);
    }

    public void testNot() {
        //Illegal operations
        try {
            mLHS.not();
            fail("exception not thrown");
        } catch (ASMLSemanticException e) {    }
    }

    public void testNegate() {
        //Illegal operations
        try {
            mLHS.negate();
            fail("exception not thrown");
        } catch (ASMLSemanticException e) {    }
    }

    public void testAmplof() {
        //Illegal operations
        try {
            mLHS.amplof();
            fail("exception not thrown");
        } catch (ASMLSemanticException e) {    }
    }

    public void testAtValue() {
        //Illegal operations
        Value tMismatches[] = {new ASMLFloat(1), new ASMLAmplitude(1),
                new ASMLFrequency(1), new ASMLTime(1),
                new ASMLString(" times a lady"), new ASMLInteger(1),};
        int numFails = 0;
        for(int i=0; i<tMismatches.length; i++){
            try {
                mLHS.at(tMismatches[i]);
                fail("exception not thrown for mismatch: " + i);
            } catch (ASMLSemanticException e) {
                numFails++;
            }
        }
        assertEquals(tMismatches.length, numFails);
    }

    public void testAtValueValue() {
        //Illegal operations
        Value tMismatches[] = {new ASMLFloat(1), new ASMLAmplitude(1),
                new ASMLFrequency(1), new ASMLTime(1),
                new ASMLString(" times a lady"), new ASMLInteger(1),};
        int numFails = 0;
        for(int i=0; i<tMismatches.length; i++){
```

```
            try {
                mLHS.at(tMismatches[i], tMismatches[i]);
                fail("exception not thrown for mismatch: " + i);
            } catch (ASMLSemanticException e) {
                numFails++;
            }
        }
        assertEquals(tMismatches.length, numFails);
    }
}
```

2d. ASMLASTTest.java

```
package asml.test;

import junit.framework.TestCase;
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;
import asml.*;

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.Iterator;

/**
 *
 * @author Frank Smith
 *
 */
public class ASMLASTTest extends TestCase {
    private ASMLParser        mParser;
    private ASMLLexer         mLexer;
    private ArrayList<String> mTestProgs;
    private ArrayList<String> mControlTrees;

    public ASMLASTTest(String name) {
        super(name);
        mTestProgs    = new ArrayList<String>();
        mControlTrees = new ArrayList<String>();
    }

    protected void setUp() throws Exception {
        super.setUp();

        mLexer = new ASMLLexer();
        mParser = new ASMLParser(new CommonTokenStream(mLexer));
    }

    protected void tearDown() throws Exception {
        super.tearDown();

        mLexer = null;
        mParser = null;
    }

/*   public void testSimple()throws RecognitionException{
        String prog = "fun wave main() end fun fun int b() end fun";

        ASMLLexer lexer = new ASMLLexer(new ANTLRStringStream(prog));
        CommonTokenStream tokens = new CommonTokenStream(lexer);
```

```
        ASMLParser parser = new ASMLParser(tokens);
        ASMLParser.program_return r = parser.program();
        CommonTree t = (CommonTree)r.getTree();
        System.out.println(t.toStringTree());
    }*/

    public void testFunctions(){
        try {
            loadTestsAndResults("functions");
            Iterator<String> progsIter   = mTestProgs.iterator();
            Iterator<String> controlIter = mControlTrees.iterator();
            runTests(progsIter, controlIter);
        } catch (Exception e) {
            fail("problem loading test progs: "+e.getMessage());
        }
    }

    public void testExpressions(){
        try {
            loadTestsAndResults("expressions");
            Iterator<String> progsIter   = mTestProgs.iterator();
            Iterator<String> controlIter = mControlTrees.iterator();
            runTests(progsIter, controlIter);
        } catch (Exception e) {
            fail("problem loading test progs: "+e.getMessage());
        }
    }

    public void testDeclarations(){
        try {
            loadTestsAndResults("declarations");
            Iterator<String> progsIter   = mTestProgs.iterator();
            Iterator<String> controlIter = mControlTrees.iterator();
            runTests(progsIter, controlIter);
        } catch (Exception e) {
            fail("problem loading test progs: "+e.getMessage());
        }
    }

    public void testStatements(){
        try {
            loadTestsAndResults("statements");
            Iterator<String> progsIter   = mTestProgs.iterator();
            Iterator<String> controlIter = mControlTrees.iterator();
            runTests(progsIter, controlIter);
        } catch (Exception e) {
            fail("problem loading test progs: "+e.getMessage());
        }
    }

    protected void runTests(Iterator<String> aPrograms,
            Iterator<String> aControl){

        int i = 1;
        CommonTree tCurTree;
        String tCurTreeVal;

        try {
            while (aPrograms.hasNext() && aControl.hasNext()) {
                setLexer(mLexer, aPrograms.next());
```

```
            setParser(mParser, mLexer);
            tCurTree = (CommonTree)mParser.program().getTree();
            tCurTreeVal = tCurTree.toStringTree();

            assertNotNull(tCurTreeVal);
            assertEquals("failed on program: "+ i,
                  aControl.next(), tCurTreeVal);
            i++;
         }
      } catch (Exception e) {
         fail("failed on program: "+ i + ": " + e.getMessage());
      }
   }

   private void setLexer(Lexer lex, String input){
      lex.reset();
      lex.setCharStream(new ANTLRStringStream(input));
   }

   private void setParser(Parser par, Lexer lex){
      par.reset();
      par.setTokenStream(new CommonTokenStream(lex));
   }

   private void loadTestsAndResults(String tag) throws Exception{
      BufferedReader in = new BufferedReader(
            new FileReader("asml\\test\\ASTPrograms.txt"));

      mTestProgs.clear();
      mControlTrees.clear();

      String line, progTemp = "";
      while((line = in.readLine())!= null){
         if((line.compareTo("") != 0) &&
               (line.charAt(0)=='@') &&
               (line.substring(1).compareTo(tag) == 0)){
            progTemp = "";
            //contTemp = "";
            while((line = in.readLine())!= null){
               if((line.compareTo("") == 0) ||
                     ((line.substring(0, 1).compareTo("/") == 0) &&
                           line.substring(1,2).compareTo("/") == 0))
                  continue;
               if(line.charAt(0)=='#'){
                  if(progTemp.compareTo("")!=0){
                     mTestProgs.add(progTemp);
                     progTemp = "";
                     mControlTrees.add(line.substring(1, line.length()));
                  }//end if it's a program separator
               }//end if == #
               else if(line.charAt(0)=='@'){ //it's the next tag, we're done
                  in.close();
                  return;
               }//end if == @
               else //it is the line of a program
                  progTemp += (" "+line);
            }//end while
         }//end if we are at our tag
      }//end while
```

```
        if(progTemp.compareTo("")!=0)
            mTestProgs.add(progTemp);
        in.close();

        if(mTestProgs.isEmpty() || mControlTrees.isEmpty())
            throw new Exception("no programs matched requested tag");

        if(mTestProgs.size() != mControlTrees.size())
            throw new Exception("unequal number of test programs and control strings");
    }
}
```

2e. ASMLFloatTest.java

```
package asml.test;

import junit.framework.TestCase;
import asml.walker.*;

/**
 * @author Frank Smith
 */
public class ASMLFloatTest extends TestCase {
    ASMLFloat mLHS;

    public ASMLFloatTest(String name) {
        super(name);
    }

    protected void setUp() throws Exception {
        super.setUp();
        mLHS = new ASMLFloat(5.0);
    }

    public void testConstructors() throws ASMLSemanticException{
        ASMLFloat tInt;

        //un-named value
        tInt = new ASMLFloat(5.0);
        assertEquals(5.0, tInt.getValue());
        assertEquals(Type.FLOAT, tInt.getType());
        assertTrue(tInt.isInitialized());
        assertFalse(tInt.isStorable());

        //declared, undefined
        tInt = new ASMLFloat("test");
        assertEquals("test", tInt.getName());
        assertEquals(Type.FLOAT, tInt.getType());
        assertFalse(tInt.isInitialized());
        assertTrue(tInt.isStorable());

        //declared, defined
        tInt = new ASMLFloat(5, "test");
        assertEquals(5.0, tInt.getValue());
        assertEquals("test", tInt.getName());
        assertEquals(Type.FLOAT, tInt.getType());
        assertTrue(tInt.isInitialized());
        assertTrue(tInt.isStorable());
    }

    public void testAdd() {
```

```
    Value tRHS, tResult;
    String tRHSType = "";

    //Legal operations
    try {
       //Integers
       tRHSType = "integer rhs";
       tRHS = new ASMLInteger(3);
       tResult = mLHS.add(tRHS);
       assertNotNull(tResult);
       assertEquals(Type.FLOAT, tResult.getType());
       assertEquals(8.0, ((ASMLFloat)tResult).getValue());

       //Floats
       tRHSType = "float rhs";
       tRHS = new ASMLFloat(3.5);
       tResult = mLHS.add(tRHS);
       assertNotNull(tResult);
       assertEquals(Type.FLOAT, tResult.getType());
       assertEquals(8.5, ((ASMLFloat)tResult).getValue());

       //Strings
       tRHSType = "string rhs";
       tRHS = new ASMLString(" times a lady");
       tResult = mLHS.add(tRHS);
       assertNotNull(tResult);
       assertEquals(Type.STRING, tResult.getType());
       assertEquals("5.0 times a lady", ((ASMLString)tResult).getValue());
    } catch (ASMLSemanticException e) {
       fail("Legal Ops throw Semantic Exception: " +
             tRHSType + " " + e.getMessage());
    }

    //Illegal operations
    Value tMismatches[] = {new ASMLAmplitude(1), new ASMLFrequency(1),
          new ASMLTime(1)};
    int numFails = 0;
    for(int i=0; i<tMismatches.length; i++){
       try {
          tResult = mLHS.add(tMismatches[i]);
          fail("exception not thrown for mismatch: " + i);
       } catch (ASMLSemanticException e) {
          numFails++;
       }
    }
    assertEquals(tMismatches.length, numFails);

    //TODO: test for waves
 }

 public void testSubtract() {
    Value tRHS, tResult;
    String tRHSType = "";

    //Legal operations
    try {
       //Integers
       tRHSType = "integer rhs";
       tRHS = new ASMLInteger(3);
       tResult = mLHS.subtract(tRHS);
```

```
            assertNotNull(tResult);
            assertEquals(Type.FLOAT, tResult.getType());
            assertEquals(2.0, ((ASMLFloat)tResult).getValue());

            //Floats
            tRHSType = "float rhs";
            tRHS = new ASMLFloat(3.5);
            tResult = mLHS.subtract(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.FLOAT, tResult.getType());
            assertEquals(1.5, ((ASMLFloat)tResult).getValue());
        } catch (ASMLSemanticException e) {
            fail("Legal Ops throw Semantic Exception: " +
                    tRHSType + " " + e.getMessage());
        }

        //Illegal operations
        Value tMismatches[] = {new ASMLAmplitude(1), new ASMLFrequency(1),
                new ASMLTime(1), new ASMLString(" times a lady")};
        int numFails = 0;
        for(int i=0; i<tMismatches.length; i++){
            try {
                tResult = mLHS.subtract(tMismatches[i]);
                fail("exception not thrown for mismatch: " + i);
            } catch (ASMLSemanticException e) {
                numFails++;
            }
        }
        assertEquals(tMismatches.length, numFails);

        //TODO: test for waves
    }

    public void testMultiply() {
        Value tRHS, tResult;
        String tRHSType = "";

        //Legal operations
        try {
            //Integers
            tRHSType = "integer rhs";
            tRHS = new ASMLInteger(3);
            tResult = mLHS.multiply(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.FLOAT, tResult.getType());
            assertEquals(15.0, ((ASMLFloat)tResult).getValue());

            //Floats
            tRHSType = "float rhs";
            tRHS = new ASMLFloat(1.5);
            tResult = mLHS.multiply(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.FLOAT, tResult.getType());
            assertEquals(7.5, ((ASMLFloat)tResult).getValue());

            //Ampls
            tRHSType = "ampl rhs";
            tRHS = new ASMLAmplitude(1.5);
            tResult = mLHS.multiply(tRHS);
            assertNotNull(tResult);
```

```
      assertEquals(Type.AMPL, tResult.getType());
      assertEquals(7.5, ((ASMLAmplitude)tResult).getValue());

      //Freqs
      tRHSType = "freq rhs";
      tRHS = new ASMLFrequency(1.5);
      tResult = mLHS.multiply(tRHS);
      assertNotNull(tResult);
      assertEquals(Type.FREQ, tResult.getType());
      assertEquals(7.5, ((ASMLFrequency)tResult).getValue());

      //Times
      tRHSType = "time rhs";
      tRHS = new ASMLTime(1.5);
      tResult = mLHS.multiply(tRHS);
      assertNotNull(tResult);
      assertEquals(Type.TIME, tResult.getType());
      assertEquals(7.5, ((ASMLTime)tResult).getValue());

      //Waves
      //TODO test for waves
    } catch (ASMLSemanticException e) {
      fail("Legal Ops throw Semantic Exception: " +
          tRHSType + " " + e.getMessage());
    }

    //Illegal operations
    Value tMismatches[] = {new ASMLString(" times a lady")};
    int numFails = 0;
    for(int i=0; i<tMismatches.length; i++){
      try {
        tResult = mLHS.multiply(tMismatches[i]);
        fail("exception not thrown for mismatch: " + i);
      } catch (ASMLSemanticException e) {
        numFails++;
      }
    }
    assertEquals(tMismatches.length, numFails);
  }

  public void testDivide() {
    Value tRHS, tResult;
    String tRHSType = "";

    //Legal operations
    try {
      //Integers
      tRHSType = "integer rhs";
      tRHS = new ASMLInteger(2);
      tResult = mLHS.divide(tRHS);
      assertNotNull(tResult);
      assertEquals(Type.FLOAT, tResult.getType());
      assertEquals(2.5, ((ASMLFloat)tResult).getValue());

      //Floats
      tRHSType = "float rhs";
      tRHS = new ASMLFloat(.5);
      tResult = mLHS.divide(tRHS);
      assertNotNull(tResult);
      assertEquals(Type.FLOAT, tResult.getType());
```

```
            assertEquals(10.0, ((ASMLFloat)tResult).getValue());
        } catch (ASMLSemanticException e) {
            fail("Legal Ops throw Semantic Exception: " +
                    tRHSType + " " + e.getMessage());
        }

        //Illegal operations
        Value tMismatches[] = {new ASMLAmplitude(1), new ASMLFrequency(1),
                new ASMLTime(1), new ASMLString(" times a lady")};
        int numFails = 0;
        for(int i=0; i<tMismatches.length; i++){
            try {
                tResult = mLHS.divide(tMismatches[i]);
                fail("exception not thrown for mismatch: " + i);
            } catch (ASMLSemanticException e) {
                numFails++;
            }
        }
        assertEquals(tMismatches.length, numFails);
    }

    public void testRelate() {
        Value tRHS, tResult;
        String tRHSType = "";

        String tOps[] = {"<", "<=", ">", ">=", "==", "!="};
        double compareF[] = {4.9, 5.0, 5.1};
        int tResult4[] = {0, 0, 1, 1, 0, 1};
        int tResult5[] = {0, 1, 0, 1, 1, 0};
        int tResult6[] = {1, 1, 0, 0, 0, 1};
        int tResults[][] = {tResult4, tResult5, tResult6};

        int i=-1, j=-1;

        //Legal operations
        try {
            //Integers
            tRHSType = "Integers";
            for(i=4; i<=6; i++){
                tRHS = new ASMLInteger(i);
                for(j=0; j<tOps.length; j++){
                    tResult = mLHS.relate(tRHS, tOps[j]);
                    assertNotNull(tResult);
                    assertEquals(Type.INT, tResult.getType());
                    assertEquals(tResults[i-4][j], ((ASMLInteger)tResult).getValue());
                }
            }

            //Floats
            tRHSType = "Floats";
            for(i=0; i<compareF.length; i++){
                tRHS = new ASMLFloat(compareF[i]);
                for(j=0; j<tOps.length; j++){
                    tResult = mLHS.relate(tRHS, tOps[j]);
                    assertNotNull(tResult);
                    assertEquals(Type.INT, tResult.getType());
                    assertEquals(tResults[i][j], ((ASMLInteger)tResult).getValue());
                }
            }
        } catch (ASMLSemanticException e) {
```

```
            fail("Legal Ops throw Semantic Exception: " +
                    tRHSType + " i: " + i + " j: " + j +
                    e.getMessage());
        }

        //Illegal operations
        Value tMismatches[] = {new ASMLAmplitude(1), new ASMLFrequency(1),
                new ASMLTime(1), new ASMLString(" times a lady")};
        int numFails = 0;
        for(i=0; i<tMismatches.length; i++){
            try {
                tResult = mLHS.relate(tMismatches[i], "<");
                fail("exception not thrown for mismatch: " + i);
            } catch (ASMLSemanticException e) {
                numFails++;
            }
        }
        assertEquals(tMismatches.length, numFails);

        try {
            tResult = mLHS.relate(new ASMLInteger(1), "q");
            fail("exception not thrown for bad op");
        } catch (ASMLSemanticException e) {
        }
    }

    public void testNegate() {
        Value tResult;
        String tRHSType = "";

        try {
            //Integers
            tRHSType = "integer rhs";
            tResult = mLHS.negate();
            assertNotNull(tResult);
            assertEquals(Type.FLOAT, tResult.getType());
            assertEquals(-5.0, ((ASMLFloat)tResult).getValue());
        } catch (ASMLSemanticException e) {
            fail("Legal Ops throw Semantic Exception: " +
                    tRHSType + " " + e.getMessage());
        }
    }

    public void testMod() {

        //Illegal operations
        Value tMismatches[] = {new ASMLFloat(1), new ASMLAmplitude(1),
                new ASMLFrequency(1), new ASMLTime(1),
                new ASMLString(" times a lady"), new ASMLInteger(1),};
        int numFails = 0;
        for(int i=0; i<tMismatches.length; i++){
            try {
                mLHS.mod(tMismatches[i]);
                fail("exception not thrown for mismatch: " + i);
            } catch (ASMLSemanticException e) {
                numFails++;
            }
        }
        assertEquals(tMismatches.length, numFails);
    }
```

```
public void testLogic() {

    //Illegal operations
    Value tMismatches[] = {new ASMLFloat(1), new ASMLAmplitude(1),
          new ASMLFrequency(1), new ASMLTime(1),
          new ASMLString(" times a lady"), new ASMLInteger(1),};
    int numFails = 0;
    for(int i=0; i<tMismatches.length; i++){
        try {
            mLHS.logic(tMismatches[i], "<");
            fail("exception not thrown for mismatch: " + i);
        } catch (ASMLSemanticException e) {
            numFails++;
        }
    }
    assertEquals(tMismatches.length, numFails);
}

public void testNot() {

    //Illegal operations
    try {
        mLHS.not();
        fail("exception not thrown");
    } catch (ASMLSemanticException e) {    }
}

public void testAmplof() {

    //Illegal operations
    try {
        mLHS.amplof();
        fail("exception not thrown");
    } catch (ASMLSemanticException e) {    }
}

public void testAtValue() {

    //Illegal operations
    Value tMismatches[] = {new ASMLFloat(1), new ASMLAmplitude(1),
          new ASMLFrequency(1), new ASMLTime(1),
          new ASMLString(" times a lady"), new ASMLInteger(1),};
    int numFails = 0;
    for(int i=0; i<tMismatches.length; i++){
        try {
            mLHS.at(tMismatches[i]);
            fail("exception not thrown for mismatch: " + i);
        } catch (ASMLSemanticException e) {
            numFails++;
        }
    }
    assertEquals(tMismatches.length, numFails);
}

public void testAtValueValue() {

    //Illegal operations
    Value tMismatches[] = {new ASMLFloat(1), new ASMLAmplitude(1),
          new ASMLFrequency(1), new ASMLTime(1),
```

```
            new ASMLString(" times a lady"), new ASMLInteger(1),};
        int numFails = 0;
        for(int i=0; i<tMismatches.length; i++){
            try {
                mLHS.at(tMismatches[i], tMismatches[i]);
                fail("exception not thrown for mismatch: " + i);
            } catch (ASMLSemanticException e) {
                numFails++;
            }
        }
        assertEquals(tMismatches.length, numFails);
    }

}
```

2f. ASMLFrequencyTest.java

```
package asml.test;

import junit.framework.TestCase;
import asml.walker.*;

/**
 *
 * @author Frank Smith
 *
 */
public class ASMLFrequencyTest extends TestCase {
    ASMLFrequency mLHS;

    public ASMLFrequencyTest(String name) {
        super(name);
    }

    protected void setUp() throws Exception {
        super.setUp();
        mLHS = new ASMLFrequency(5.0);
    }

    public void testConstructors() throws ASMLSemanticException{
        ASMLFrequency tFreq;

        //un-named value
        tFreq = new ASMLFrequency(5.0);
        assertEquals(5.0, tFreq.getValue());
        assertEquals(Type.FREQ, tFreq.getType());
        assertTrue(tFreq.isInitialized());
        assertFalse(tFreq.isStorable());

        //declared, undefined
        tFreq = new ASMLFrequency("test");
        assertEquals("test", tFreq.getName());
        assertEquals(Type.FREQ, tFreq.getType());
        assertFalse(tFreq.isInitialized());
        assertTrue(tFreq.isStorable());

        //declared, defined
        tFreq = new ASMLFrequency(5, "test");
        assertEquals(5.0, tFreq.getValue());
        assertEquals("test", tFreq.getName());
        assertEquals(Type.FREQ, tFreq.getType());
```

```
        assertTrue(tFreq.isInitialized());
        assertTrue(tFreq.isStorable());
    }

    public void testAdd() {
        Value tRHS, tResult;
        String tRHSType = "";

        try {
            //Frequencies
            tRHSType = "frequency rhs";
            tRHS = new ASMLFrequency(3.5);
            tResult = mLHS.add(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.FREQ, tResult.getType());
            assertEquals(8.5, ((ASMLFrequency)tResult).getValue());

            //Strings
            tRHSType = "string rhs";
            tRHS = new ASMLString(" times a lady");
            tResult = mLHS.add(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.STRING, tResult.getType());
            assertEquals("5.0Hz times a lady", ((ASMLString)tResult).getValue());
        } catch (ASMLSemanticException e) {
            fail("Legal Ops throw Semantic Exception: " +
                    tRHSType + " " + e.getMessage());
        }

        //Illegal operations
        Value tMismatches[] = {
            new ASMLAmplitude(1), new ASMLFloat(1),
            new ASMLInteger(1), new ASMLTime(1)};
        int numFails = 0;
        for(int i=0; i<tMismatches.length; i++){
            try {
                tResult = mLHS.add(tMismatches[i]);
                fail("exception not thrown for mismatch: " + i);
            } catch (ASMLSemanticException e) {
                numFails++;
            }
        }
        assertEquals(tMismatches.length, numFails);
    }

    public void testSubtract() {
        Value tRHS, tResult;
        String tRHSType = "";

        try {
            //Frequencies
            tRHSType = "frequency rhs";
            tRHS = new ASMLFrequency(3.5);
            tResult = mLHS.subtract(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.FREQ, tResult.getType());
            assertEquals(1.5, ((ASMLFrequency)tResult).getValue());
        } catch (ASMLSemanticException e) {
            fail("Legal Ops throw Semantic Exception: " +
                    tRHSType + " " + e.getMessage());
```

```
    }

    //Illegal operations
    Value tMismatches[] = {
        new ASMLAmplitude(1), new ASMLFloat(1), new ASMLString("1"),
        new ASMLInteger(1), new ASMLTime(1)};
    int numFails = 0;
    for(int i=0; i<tMismatches.length; i++){
        try {
            tResult = mLHS.subtract(tMismatches[i]);
            fail("exception not thrown for mismatch: " + i);
        } catch (ASMLSemanticException e) {
            numFails++;
        }
    }
    assertEquals(tMismatches.length, numFails);
}

public void testMultiply() {
    Value tRHS, tResult;
    String tRHSType = "";

    try {
        //Integers
        tRHSType = "integer rhs";
        tRHS = new ASMLInteger(3);
        tResult = mLHS.multiply(tRHS);
        assertNotNull(tResult);
        assertEquals(Type.FREQ, tResult.getType());
        assertEquals(15.0, ((ASMLFrequency)tResult).getValue());

        //Floats
        tRHSType = "float rhs";
        tRHS = new ASMLFloat(3.0);
        tResult = mLHS.multiply(tRHS);
        assertNotNull(tResult);
        assertEquals(Type.FREQ, tResult.getType());
        assertEquals(15.0, ((ASMLFrequency)tResult).getValue());
    } catch (ASMLSemanticException e) {
        fail("Legal Ops throw Semantic Exception: " +
                tRHSType + " " + e.getMessage());
    }

    //Illegal operations
    Value tMismatches[] = {
        new ASMLAmplitude(1), new ASMLString("1"),
        new ASMLTime(1), new ASMLFrequency(1)};
    int numFails = 0;
    for(int i=0; i<tMismatches.length; i++){
        try {
            tResult = mLHS.multiply(tMismatches[i]);
            fail("exception not thrown for mismatch: " + i);
        } catch (ASMLSemanticException e) {
            numFails++;
        }
    }
    assertEquals(tMismatches.length, numFails);
}

public void testDivide() {
```

```
    Value tRHS, tResult;
    String tRHSType = "";

    try {
        //Integers
        tRHSType = "integer rhs";
        tRHS = new ASMLInteger(2);
        tResult = mLHS.divide(tRHS);
        assertNotNull(tResult);
        assertEquals(Type.FREQ, tResult.getType());
        assertEquals(2.5, ((ASMLFrequency)tResult).getValue());

        //Floats
        tRHSType = "float rhs";
        tRHS = new ASMLFloat(2.0);
        tResult = mLHS.divide(tRHS);
        assertNotNull(tResult);
        assertEquals(Type.FREQ, tResult.getType());
        assertEquals(2.5, ((ASMLFrequency)tResult).getValue());

        //Frequencies
        tRHSType = "frequency rhs";
        tRHS = new ASMLFrequency(2.0);
        tResult = mLHS.divide(tRHS);
        assertNotNull(tResult);
        assertEquals(Type.FLOAT, tResult.getType());
        assertEquals(2.5, ((ASMLFloat)tResult).getValue());
    } catch (ASMLSemanticException e) {
        fail("Legal Ops throw Semantic Exception: " +
                tRHSType + " " + e.getMessage());
    }

    //Illegal operations
    Value tMismatches[] = {
        new ASMLAmplitude(1), new ASMLString("1"),
        new ASMLTime(1)};
    int numFails = 0;
    for(int i=0; i<tMismatches.length; i++){
        try {
            tResult = mLHS.divide(tMismatches[i]);
            fail("exception not thrown for mismatch: " + i);
        } catch (ASMLSemanticException e) {
            numFails++;
        }
    }
    assertEquals(tMismatches.length, numFails);
}

public void testRelate() {
    Value tRHS, tResult;

    String tOps[] = {"<", "<=", ">", ">=", "==", "!="};
    double compareF[] = {4.9, 5.0, 5.1};
    int tResult4[] = {0, 0, 1, 1, 0, 1};
    int tResult5[] = {0, 1, 0, 1, 1, 0};
    int tResult6[] = {1, 1, 0, 0, 0, 1};
    int tResults[][] = {tResult4, tResult5, tResult6};

    int i=-1, j=-1;
```

```
    //Legal operations
    try {
        //Frequencies
        for(i=0; i<compareF.length; i++){
            tRHS = new ASMLFrequency(compareF[i]);
            for(j=0; j<tOps.length; j++){
                tResult = mLHS.relate(tRHS, tOps[j]);
                assertNotNull(tResult);
                assertEquals(Type.INT, tResult.getType());
                assertEquals(tResults[i][j], ((ASMLInteger)tResult).getValue());
            }
        }
    } catch (ASMLSemanticException e) {
        fail("Legal Ops throw Semantic Exception:" + " i: " + i + " j: " + j +
                e.getMessage());
    }

    //Illegal operations
    Value tMismatches[] = {new ASMLAmplitude(1), new ASMLFloat(1),
            new ASMLInteger(1), new ASMLTime(1),
            new ASMLString(" times a lady")};
    int numFails = 0;
    for(i=0; i<tMismatches.length; i++){
        try {
            tResult = mLHS.relate(tMismatches[i], "<");
            fail("exception not thrown for mismatch: " + i);
        } catch (ASMLSemanticException e) {
            numFails++;
        }
    }
    assertEquals(tMismatches.length, numFails);

    try {
        tResult = mLHS.relate(new ASMLFrequency(1), "q");
        fail("exception not thrown for bad op");
    } catch (ASMLSemanticException e) {
    }
}

public void testMod() {

    //Illegal operations
    Value tMismatches[] = {new ASMLFloat(1), new ASMLAmplitude(1),
            new ASMLFrequency(1), new ASMLTime(1),
            new ASMLString(" times a lady"), new ASMLInteger(1),};
    int numFails = 0;
    for(int i=0; i<tMismatches.length; i++){
        try {
            mLHS.mod(tMismatches[i]);
            fail("exception not thrown for mismatch: " + i);
        } catch (ASMLSemanticException e) {
            numFails++;
        }
    }
    assertEquals(tMismatches.length, numFails);
}

public void testLogic() {
    //Illegal operations
    Value tMismatches[] = {new ASMLFloat(1), new ASMLAmplitude(1),
```

```
            new ASMLFrequency(1), new ASMLTime(1),
            new ASMLString(" times a lady"), new ASMLInteger(1),};
        int numFails = 0;
        for(int i=0; i<tMismatches.length; i++){
            try {
                mLHS.logic(tMismatches[i], "||");
                fail("exception not thrown for mismatch: " + i);
            } catch (ASMLSemanticException e) {
                numFails++;
            }
        }
        assertEquals(tMismatches.length, numFails);
    }

    public void testNot() {

        //Illegal operations
        try {
            mLHS.not();
            fail("exception not thrown");
        } catch (ASMLSemanticException e) {    }
    }

    public void testNegate() {
        //Illegal operations
        try {
            mLHS.negate();
            fail("exception not thrown");
        } catch (ASMLSemanticException e) {    }
    }

    public void testAmplof() {
        //Illegal operations
        try {
            mLHS.amplof();
            fail("exception not thrown");
        } catch (ASMLSemanticException e) {    }
    }

    public void testAtValue() {
        //Illegal operations
        Value tMismatches[] = {new ASMLFloat(1), new ASMLAmplitude(1),
            new ASMLFrequency(1), new ASMLTime(1),
            new ASMLString(" times a lady"), new ASMLInteger(1),};
        int numFails = 0;
        for(int i=0; i<tMismatches.length; i++){
            try {
                mLHS.at(tMismatches[i]);
                fail("exception not thrown for mismatch: " + i);
            } catch (ASMLSemanticException e) {
                numFails++;
            }
        }
        assertEquals(tMismatches.length, numFails);
    }

    public void testAtValueValue() {
        //Illegal operations
        Value tMismatches[] = {new ASMLFloat(1), new ASMLAmplitude(1),
            new ASMLFrequency(1), new ASMLTime(1),
```

```
            new ASMLString(" times a lady"), new ASMLInteger(1),};
        int numFails = 0;
        for(int i=0; i<tMismatches.length; i++){
            try {
                mLHS.at(tMismatches[i], tMismatches[i]);
                fail("exception not thrown for mismatch: " + i);
            } catch (ASMLSemanticException e) {
                numFails++;
            }
        }
        assertEquals(tMismatches.length, numFails);
    }

}
```

2g. ASMLFunWalkerTest.java

```
package asml.test;

import java.io.*;
import java.util.ArrayList;
import java.util.HashMap;

import asml.*;
import asml.walker.*;
import junit.framework.TestCase;
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;

/**
 *
 * @author Frank Smith
 *
 */
public class ASMLFunWalkerTest extends TestCase {

    public ASMLFunWalkerTest(String name) {
        super(name);
    }

    protected void setUp() throws Exception {
        super.setUp();
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

    public void testProgram1() throws Exception{
        ASMLFunWalker tFWalk = createWalker("program 1");

        tFWalk.program();

        HashMap<String, FunctionRecord> tFunTable = tFWalk.getFunctionTable();

        assertEquals(3, tFunTable.size());

        //fun 1
        FunctionRecord tCurrRecord = tFunTable.get("fun1Args2");
        assertEquals(2, tCurrRecord.getNumArgs());
        assertNotNull(tCurrRecord.getBlockRt());
```

```
        ArrayList<Value> tActArgs = new ArrayList<Value>();
        try {
            tCurrRecord.passParamValue(tActArgs);
            fail("Should not have been able to pass empty arg list");
        } catch (ASMLSemanticException e) {    }

        tActArgs.add(new ASMLInteger(1));
        tActArgs.add(new ASMLFloat(2));
        tCurrRecord.passParamValue(tActArgs);
        try {
            tCurrRecord.setRetVal(new ASMLInteger(2));
            fail("Should not have been able to set return value to wrong type.");
        } catch (ASMLSemanticException e) {    }
        tCurrRecord.setRetVal(new ASMLAmplitude(2));

        //fun 2
        tCurrRecord = tFunTable.get("fun2Args3");
        assertEquals(3, tCurrRecord.getNumArgs());
        assertNotNull(tCurrRecord.getBlockRt());

        tActArgs.clear();
        try {
            tCurrRecord.passParamValue(tActArgs);
            fail("Should not have been able to pass empty arg list");
        } catch (ASMLSemanticException e) {    }

        tActArgs.add(new ASMLFrequency(1));
        tActArgs.add(new ASMLAmplitude(2));
        tActArgs.add(new ASMLTime(2));
        tCurrRecord.passParamValue(tActArgs);
        try {
            tCurrRecord.setRetVal(new ASMLInteger(2));
            fail("Should not have been able to set return value to wrong type.");
        } catch (ASMLSemanticException e) {    }
        tCurrRecord.setRetVal(new ASMLTime(2));

        //fun 3
        tCurrRecord = tFunTable.get("fun3Args0");
        assertEquals(0, tCurrRecord.getNumArgs());
        assertNotNull(tCurrRecord.getBlockRt());

        try {
            tCurrRecord.passParamValue(tActArgs);
            fail("Should not have been able to pass populated arg list");
        } catch (ASMLSemanticException e) {    }

        tActArgs.clear();
        tCurrRecord.passParamValue(tActArgs);
        try {
            tCurrRecord.setRetVal(new ASMLTime(2));
            fail("Should not have been able to set return value to wrong type.");
        } catch (ASMLSemanticException e) {    }
        tCurrRecord.setRetVal(new ASMLInteger(2));
    }

/*  public void testProgram2() throws Exception{
        ASMLFunWalker tFWalk = createWalker("program 2");

        tFWalk.program();
```

```
    }*/

/*   public void testProgram3() throws Exception{
        ASMLFunWalker tFWalk = createWalker("program 3");

        tFWalk.program();
    }*/

    private ASMLFunWalker createWalker(String aProgTag) throws Exception{
        String tProgram = getProgramString(aProgTag);

        ANTLRStringStream input = new ANTLRStringStream(tProgram);
        ASMLLexer lexer = new ASMLLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ASMLParser parser = new ASMLParser(tokens);
        ASMLParser.program_return r = null;
        try {
            r = parser.program();
        } catch (RecognitionException e) {
            fail("Parse error: " + e.getMessage());
        }

        CommonTree t = (CommonTree)r.getTree();
        CommonTreeNodeStream nodes = new CommonTreeNodeStream(t);
        nodes.setTokenStream(tokens);
        ASMLFunWalker walker = new ASMLFunWalker(nodes);

        return walker;
    }

    private String getProgramString(String aProgTag) throws Exception{
        String tProgram = "";
        BufferedReader in = new BufferedReader(
                new FileReader("asml\\test\\FunWalkerTest.txt"));

        String line = "";
        while((line = in.readLine())!= null){
            if((line.compareTo("") != 0) &&
                    (line.charAt(0)=='@') &&
                    (line.substring(1).compareTo(aProgTag) == 0)){
                while((line = in.readLine())!= null){
                    if((line.compareTo("") == 0) ||
                            ((line.substring(0, 1).compareTo("/") == 0) &&
                                    line.substring(1,2).compareTo("/") == 0))
                        continue;
                    if(line.charAt(0)=='@'){ //it's the next aProgTag, we're done
                        return tProgram;
                    }//end if == @
                    else //it is the line of a program
                        tProgram += (" "+line);
                }//end while
            }//end if we are at our aProgTag
        }//end while


        in.close();

        if(tProgram.equals(""))
            throw new Exception("no programs matched requested aProgTag");
```

```
        return tProgram;
    }
}
```

2h. ASMLIntegerTest.java

```
package asml.test;

import junit.framework.TestCase;
import asml.walker.*;

/**
 *
 * @author Frank Smith
 *
 */
public class ASMLIntegerTest extends TestCase {
    ASMLInteger mLHS;

    public ASMLIntegerTest(String name) {
        super(name);
    }

    protected void setUp() throws Exception {
        super.setUp();
        mLHS = new ASMLInteger(5);
    }

    public void testConstructors() throws ASMLSemanticException{
        ASMLInteger tInt;

        //un-named value
        tInt = new ASMLInteger(5);
        assertEquals(5, tInt.getValue());
        assertEquals(Type.INT, tInt.getType());
        assertTrue(tInt.isInitialized());
        assertFalse(tInt.isStorable());

        //declared, undefined
        tInt = new ASMLInteger("test");
        assertEquals("test", tInt.getName());
        assertEquals(Type.INT, tInt.getType());
        assertFalse(tInt.isInitialized());
        assertTrue(tInt.isStorable());

        //declared, defined
        tInt = new ASMLInteger(5, "test");
        assertEquals(5, tInt.getValue());
        assertEquals("test", tInt.getName());
        assertEquals(Type.INT, tInt.getType());
        assertTrue(tInt.isInitialized());
        assertTrue(tInt.isStorable());
    }

    public void testASMLIntegerValue(){

        ASMLInteger tInt;
        try {
            tInt = new ASMLInteger(new ASMLInteger(5, "test"));
            assertEquals(5, tInt.getValue());
            assertEquals("test", tInt.getName());
```

```
            assertEquals(Type.INT, tInt.getType());
            assertTrue(tInt.isInitialized());
            assertTrue(tInt.isStorable());
        } catch (ASMLSemanticException e1) {
            fail("Could not assign ASMLInteger");
        }


        //Illegal ops
        Value tNonInts[] = {
            new ASMLAmplitude(1), new ASMLFloat(1), new ASMLFrequency(1),
            new ASMLString("1"), new ASMLTime(1)};
        for(int i=0; i<tNonInts.length; i++)
            try {
                tInt = new ASMLInteger(tNonInts[i]);
                fail("No exception thrown for Non int Value #" + i);
            } catch (ASMLSemanticException e) {}
    }

    public void testAdd() {
        Value tRHS, tResult;
        String tRHSType = "";

        //Legal operations
        try {
            //Integers
            tRHSType = "integer rhs";
            tRHS = new ASMLInteger(3);
            tResult = mLHS.add(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.INT, tResult.getType());
            assertEquals(8, ((ASMLInteger)tResult).getValue());

            //Floats
            tRHSType = "float rhs";
            tRHS = new ASMLFloat(3.5);
            tResult = mLHS.add(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.FLOAT, tResult.getType());
            assertEquals(8.5, ((ASMLFloat)tResult).getValue());

            //Strings
            tRHSType = "string rhs";
            tRHS = new ASMLString(" times a lady");
            tResult = mLHS.add(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.STRING, tResult.getType());
            assertEquals("5 times a lady", ((ASMLString)tResult).getValue());
        } catch (ASMLSemanticException e) {
            fail("Legal Ops throw Semantic Exception: " +
                    tRHSType + " " + e.getMessage());
        }

        //Illegal operations
        Value tMismatches[] = {new ASMLAmplitude(1), new ASMLFrequency(1),
                new ASMLTime(1)};
        int numFails = 0;
        for(int i=0; i<tMismatches.length; i++){
            try {
                tResult = mLHS.add(tMismatches[i]);
```

```
            fail("exception not thrown for mismatch: " + i);
        } catch (ASMLSemanticException e) {
            numFails++;
        }
    }
    assertEquals(tMismatches.length, numFails);

    //TODO: test for waves
}

public void testSubtract() {
    Value tRHS, tResult;
    String tRHSType = "";

    //Legal operations
    try {
        //Integers
        tRHSType = "integer rhs";
        tRHS = new ASMLInteger(3);
        tResult = mLHS.subtract(tRHS);
        assertNotNull(tResult);
        assertEquals(Type.INT, tResult.getType());
        assertEquals(2, ((ASMLInteger)tResult).getValue());

        //Floats
        tRHSType = "float rhs";
        tRHS = new ASMLFloat(3.5);
        tResult = mLHS.subtract(tRHS);
        assertNotNull(tResult);
        assertEquals(Type.FLOAT, tResult.getType());
        assertEquals(1.5, ((ASMLFloat)tResult).getValue());
    } catch (ASMLSemanticException e) {
        fail("Legal Ops throw Semantic Exception: " +
                tRHSType + " " + e.getMessage());
    }

    //Illegal operations
    Value tMismatches[] = {new ASMLAmplitude(1), new ASMLFrequency(1),
            new ASMLTime(1), new ASMLString(" times a lady")};
    int numFails = 0;
    for(int i=0; i<tMismatches.length; i++){
        try {
            tResult = mLHS.subtract(tMismatches[i]);
            fail("exception not thrown for mismatch: " + i);
        } catch (ASMLSemanticException e) {
            numFails++;
        }
    }
    assertEquals(tMismatches.length, numFails);

    //TODO: test for waves
}

public void testMultiply() {
    Value tRHS, tResult;
    String tRHSType = "";

    //Legal operations
    try {
        //Integers
```

```
            tRHSType = "integer rhs";
            tRHS = new ASMLInteger(3);
            tResult = mLHS.multiply(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.INT, tResult.getType());
            assertEquals(15, ((ASMLInteger)tResult).getValue());

            //Floats
            tRHSType = "float rhs";
            tRHS = new ASMLFloat(1.5);
            tResult = mLHS.multiply(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.FLOAT, tResult.getType());
            assertEquals(7.5, ((ASMLFloat)tResult).getValue());

            //Ampls
            tRHSType = "ampl rhs";
            tRHS = new ASMLAmplitude(1.5);
            tResult = mLHS.multiply(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.AMPL, tResult.getType());
            assertEquals(7.5, ((ASMLAmplitude)tResult).getValue());

            //Freqs
            tRHSType = "freq rhs";
            tRHS = new ASMLFrequency(1.5);
            tResult = mLHS.multiply(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.FREQ, tResult.getType());
            assertEquals(7.5, ((ASMLFrequency)tResult).getValue());

            //Times
            tRHSType = "time rhs";
            tRHS = new ASMLTime(1.5);
            tResult = mLHS.multiply(tRHS);
            assertNotNull(tResult);
            assertEquals(Type.TIME, tResult.getType());
            assertEquals(7.5, ((ASMLTime)tResult).getValue());

            //Waves
            //TODO test for waves
        } catch (ASMLSemanticException e) {
            fail("Legal Ops throw Semantic Exception: " +
                    tRHSType + " " + e.getMessage());
        }

        //Illegal operations
        Value tMismatches[] = {new ASMLString(" times a lady")};
        int numFails = 0;
        for(int i=0; i<tMismatches.length; i++){
            try {
                tResult = mLHS.multiply(tMismatches[i]);
                fail("exception not thrown for mismatch: " + i);
            } catch (ASMLSemanticException e) {
                numFails++;
            }
        }
        assertEquals(tMismatches.length, numFails);
    }
```

```
public void testDivide() {
   Value tRHS, tResult;
   String tRHSType = "";

   //Legal operations
   try {
      //Integers
      tRHSType = "integer rhs";
      tRHS = new ASMLInteger(3);
      tResult = mLHS.divide(tRHS);
      assertNotNull(tResult);
      assertEquals(Type.INT, tResult.getType());
      assertEquals(1, ((ASMLInteger)tResult).getValue());

      //Floats
      tRHSType = "float rhs";
      tRHS = new ASMLFloat(.5);
      tResult = mLHS.divide(tRHS);
      assertNotNull(tResult);
      assertEquals(Type.FLOAT, tResult.getType());
      assertEquals(10.0, ((ASMLFloat)tResult).getValue());
   } catch (ASMLSemanticException e) {
      fail("Legal Ops throw Semantic Exception: " +
            tRHSType + " " + e.getMessage());
   }

   //Illegal operations
   Value tMismatches[] = {new ASMLAmplitude(1), new ASMLFrequency(1),
         new ASMLTime(1), new ASMLString(" times a lady")};
   int numFails = 0;
   for(int i=0; i<tMismatches.length; i++){
      try {
         tResult = mLHS.divide(tMismatches[i]);
         fail("exception not thrown for mismatch: " + i);
      } catch (ASMLSemanticException e) {
         numFails++;
      }
   }
   assertEquals(tMismatches.length, numFails);
}

public void testMod() {
   Value tRHS, tResult;
   String tRHSType = "";

   //Legal operations
   try {
      //Integers
      tRHSType = "integer rhs";
      tRHS = new ASMLInteger(3);
      tResult = mLHS.mod(tRHS);
      assertNotNull(tResult);
      assertEquals(Type.INT, tResult.getType());
      assertEquals(2, ((ASMLInteger)tResult).getValue());
   } catch (ASMLSemanticException e) {
      fail("Legal Ops throw Semantic Exception: " +
            tRHSType + " " + e.getMessage());
   }

   //Illegal operations
```

```
        Value tMismatches[] = {new ASMLFloat(1), new ASMLAmplitude(1),
                new ASMLFrequency(1), new ASMLTime(1),
                new ASMLString(" times a lady")};
        int numFails = 0;
        for(int i=0; i<tMismatches.length; i++){
            try {
                tResult = mLHS.mod(tMismatches[i]);
                fail("exception not thrown for mismatch: " + i);
            } catch (ASMLSemanticException e) {
                numFails++;
            }
        }
        assertEquals(tMismatches.length, numFails);
    }

    public void testRelate() {
        Value tRHS, tResult;
        String tRHSType = "";

        String tOps[] = {"<", "<=", ">", ">=", "==", "!="};
        double compareF[] = {4.9, 5.0, 5.1};
        int tResult4[] = {0, 0, 1, 1, 0, 1};
        int tResult5[] = {0, 1, 0, 1, 1, 0};
        int tResult6[] = {1, 1, 0, 0, 0, 1};
        int tResults[][] = {tResult4, tResult5, tResult6};

        int i=-1, j=-1;

        //Legal operations
        try {
            //Integers
            tRHSType = "Integers";
            for(i=4; i<=6; i++){
                tRHS = new ASMLInteger(i);
                for(j=0; j<tOps.length; j++){
                    tResult = mLHS.relate(tRHS, tOps[j]);
                    assertNotNull(tResult);
                    assertEquals(Type.INT, tResult.getType());
                    assertEquals(tResults[i-4][j], ((ASMLInteger)tResult).getValue());
                }
            }

            //Floats
            tRHSType = "Floats";
            for(i=0; i<compareF.length; i++){
                tRHS = new ASMLFloat(compareF[i]);
                for(j=0; j<tOps.length; j++){
                    tResult = mLHS.relate(tRHS, tOps[j]);
                    assertNotNull(tResult);
                    assertEquals(Type.INT, tResult.getType());
                    assertEquals(tResults[i][j], ((ASMLInteger)tResult).getValue());
                }
            }
        } catch (ASMLSemanticException e) {
            fail("Legal Ops throw Semantic Exception: " +
                    tRHSType + " i: " + i + " j: " + j +
                    e.getMessage());
        }

        //Illegal operations
```

```
      Value tMismatches[] = {new ASMLAmplitude(1), new ASMLFrequency(1),
            new ASMLTime(1), new ASMLString(" times a lady")};
      int numFails = 0;
      for(i=0; i<tMismatches.length; i++){
         try {
            tResult = mLHS.relate(tMismatches[i], "<");
            fail("exception not thrown for mismatch: " + i);
         } catch (ASMLSemanticException e) {
            numFails++;
         }
      }
      assertEquals(tMismatches.length, numFails);

      try {
         tResult = mLHS.relate(new ASMLInteger(1), "q");
         fail("exception not thrown for bad op");
      } catch (ASMLSemanticException e) {
      }
   }

   public void testLogic() {
      Value tRHS, tResult;
      String tRHSType = "";

      //Legal operations
      try {
         tRHSType = "T && T";
         tRHS = new ASMLInteger(3);
```

```
/* 1
 * for testing a simple function call:*/
fun int foo()
        print "foo";
end fun
fun wave main()
        print "pre foo";
        foo();
        print "post foo";
end fun
/* 2
 * testing the ids and param passing*/
fun int foo(time t, int i)
        print "foo " + t;
        print "int " + i;
end fun
fun wave main(freq f, time t)
        print "frequency is " + f;
        foo(t, 3);
end fun
/* 3
 * simple test of if statements*/
fun wave main()
        /*if one*/
        if(1 == 1)
                print "one works";
        end else
                print "one fails";
```

```
        end if
        /*if two*/
        if(1 == 2)
                print "two fails";
        end else
                print "two works";
        end if
        /*if three*/
        if(3!=3)
                print "three fails";
                print "don't listen to the next print statement!";
        end if
        print "three works";
end fun
/* 4
 * Test for testing scope, declare, and assign*/
fun wave main()
        int i;
        i = 1;
        print "should be 1: " + i;
        if(1)
                int i=10;
                print "should be 10: " + i;
                i = i + 5;
                print "should be 15: " + i;
        end if

        i = i + 5;
        print "should be 6: " + i;
end fun
/* 5
 * Test for while statement*/
fun wave main()
        int i = 1;
        print "should print 1 to 10";
        while(i <= 10)
                print "-->" + i;
                i = i + 1;
        end while

        print "did it work?";
        while(i > 11)
                print "Fail";
        end while
        print "no infinite loop, so non entry works.";
end fun
/* 6
 * test the for statement
 * TODO also test by messing with each expression and scoping (define new i in
loop)*/
fun wave main()
        int i;
        print "should print 1 to 10 (for)";
        for(i=1; i<=10; i=i+1)
                print "-->" + i;
        end for

        print "did it work?";
        for(i=20; i<=10; i=i+1)
                print "Fail";
        end for
        for(i=1; i>=10; i=i+1)
                print "Fail";
        end for
```

```
        print "should print 1 to 10 every 3rd value (for)";
        for(i=1; i<=10; i=i+3)
                print "-->" + i;
        end for
end fun
/* 7
 * test return trying to make sure that the scope pops and the right value is
returned */
fun wave main()
        int i = 1;
        print "foo should return 15Hz: " + foo();
        print "value of i should be 1: " + i;
end fun
fun freq foo()
        print "in foo";
        int i = 2;
        freq f = 5Hz;
        print "should print f 5 -> 15";
        while (f <= 60Hz)
                print "f: " + f;
                if(f == 15Hz)
                        int i = 3;
                        print "about to return";
                        return f;
                end if
                f = f + 5Hz;
        end while
        print "FAILURE";
end fun
/* 8
 * Writing a simple fibonacci... */
fun wave main(int fib)
        print "fib value: " + doFib(fib);
end fun
fun int doFib(int num)
        if(num == 0)
                return 0;
        end else
                if(num == 1)
                        return 1;
                end else
                        return doFib(num-1) + doFib(num-2);
                end if
        end if
end fun
/* 9
 * some test for not and negative */
fun wave main()
    print "should be -51: " + -5 + 1;
        print "should be -4: " + (-5+1);
        print "should be 4: " + (5-1);
        if(1 == 2)
                print "fail";
        end if
        if(!(1==2))
                print "pass";
        end if
end fun
/* 10
 * some simple tests of operators*/
fun wave main()
        print "should be 5.0Hz: " + (2Hz + 3Hz);
        print "should be 4.0Hz: " + (12Hz - 8Hz);
        print "should be 16.0a: " + (4.0a * 4.0);
```

```
        print "should be 4.0a: " + (16.0a / 4);
        print "should be 2: " + (2%3);
        print "should be -4.0a: " + (16.0a / -4);
        /*relations are covered by if, while and for tests*/
        if(1 || 0)
                print "Pass"; end
        else
                print "Fail"; end if
        if(1 && 0)
                print "Fail"; end
        else
                print "Pass"; end if
        if(!(1 || 0))
                print "Fail"; end
        else
                print "Pass"; end if
        end fun
```

2q. TestPrograms.txt

```
@if tests
//1 simple if
fun int main ()
  if(a)
    b=c;
  end if
end fun
#-------------
//2 simple if/else
fun int main ()
  if(a)
    b=c;
  end
  else
    d=e;
  end if
end fun
#--------------
//3 nested if and else
fun int main()
        if(a)
                if(b)
                        c = d;
                end if
        end
        else if(e)
                f = g;
        end if end if
end fun
#---------------
//4 ifs side by side
fun int main()
        if(a)
                b = c;
        end if
        if (d)
                e = f;
        end if
end fun
#---------------
//5 if with complex expression
fun int main ()
  if((a = b+c)/funcall(x+34/2) <= 5)
```

```
      b=c;
   end if
end fun
#---------------
//6 if/else inside of an if
fun wave main()
            if(a == b)
                        if(c <= d)
                                    print c;
                        end
                        else
                                    print d;
                        end if
            end if
end fun
@bad ifs
fun int main()
            if(a == b)
            b = b + 1;
            end
      if else
            b = b - 1;
            end
end fun
#---------------
fun int main()
            if(int a = x)
                        a = b;
            end if
end fun
#---------------
fun int main()
            if(a)
                        a = b
            end if
end fun
#---------------
fun int main()
            if(a)
                        a = b;
            if
end fun
#---------------
fun int main()
            if(a) a = b;
end fun
@fun tests
fun ampl main()
end fun
#---------------
fun float a()
end fun
fun int b()
end fun
#---------------
fun wave main (int a)
end fun
#---------------
fun wave main (int i, float f, ampl c)
end fun
```

```
@bad funs
//1 no type for the fun
fun main (int a)
end fun
#---------------
//2 bounding fun turns into a new function, followed by another function...
fun wave main()
end
fun int next()
end fun
fun int nextnext()
end fun
#---------------
//3 two ends
fun main()
end end fun
#---------------
//4 no param type
fun main(a)
end fun
#---------------
//5 no param id
fun main(int)
end fun
#---------------
//6 no param comma
fun main(int i float f)
end fun
#---------------
//7 param comma missing
fun main(int i, float f ampl a)
end fun
#---------------
//8 nested fun declarations
fun main()
		fun next()
		end fun
end fun
#---------------
//9 statement outside of a block
if(a==1)
		a;
end if
fun main()
end fun
#---------------
//10 expression in argument field.
fun main(a = x + b)
end fun
#----------------
//11 removing const from the language.
fun wave main (const int a)
end fun
#---------------
//12 bounding fun turns into a new function
fun wave main()
end
fun int next()
end fun
#---------------
```

```
//13 no bounding fun
fun wave main()
end
@good whiles
fun wave main()
		while(a == b)
			a;
		end while
end fun
#---------------
fun wave main()
		while(a*b+c || (funcall(fc) <= b+34.8) + b)
			a;
		end while
end fun
#---------------
fun wave main()
		while(a == b)
			a;
		end while
		while(c == d)
			c;
		end while
end fun
#---------------
fun wave main()
		while(a==b)
			while(c == d)
				a + c;
			end while
		end while
end fun
#---------------
fun wave main()
		while(a=b)
			a = b;
		end while
end fun
@bad whiles
//missing semi
fun wave main()
		while(a==b)
			a
		end while
end fun
#---------------
//missing end
fun wave main()
		while(a==b)
			a;
		while(b==c)
			b;
		end while
end fun
#---------------
//missing while
fun wave main()
		while(a==b)
			a;
		end
```

```
end fun
#---------------
//non-expression
fun wave main()
	while(int a=b)
		a;
	end while
end fun
#---------------
//missing while
fun wave main()
	while(a==b)
		a;
	end
	while(b==c)
		b;
	end while
end fun
#---------------
//statement in parens
fun wave main()
	while(if(a==b) a; end if)
		a;
	end while
	while(b==c)
		b;
	end while
end fun
@good fors
fun wave main()
	for(a; b; c)
		a;
	end for
end fun
#---------------
fun wave main()
	for(a*b+c || (funcall(fc) <= b+34.8) + b; ((a = b+c)/funcall(x+34/2)<=5); c)
		a;
	end for
end fun
#---------------
fun wave main()
	for(a; b; c)
		a;
	end for
	for(e; f; g)
		c;
	end for
end fun
#---------------
fun wave main()
	for(a; b; c)
		for(d; e; f)
			a + c;
		end for
	end for
end fun
@bad fors
//missing semi
fun wave main()
```

```
		for(a; b; c)
			a
		end for
end fun
#---------------
//missing end
fun wave main()
		for(a; b; c)
			a;
		for(d; e; f)
			b;
		end for
end fun
#---------------
//missing for
fun wave main()
		for(a; b; c)
			a;
		end
end fun
#---------------
//non-expression 1
fun wave main()
		for(int a=b; c; d)
			a;
		end while
end fun
#---------------
//non-expression 2
fun wave main()
		for(b; int a=c; d)
			a;
		end while
end fun
#---------------
//missing semis
fun wave main()
		for(b c d)
			a;
		end while
end fun
#---------------
//not enough exprs
fun wave main()
		for(b; c;)
			a;
		end while
end fun
#---------------
//missing for
fun wave main()
		for(a; b; c)
			a;
		end
		for(b; c; d)
			b;
		end for
end fun
#---------------
//statement in parens
```

```
fun wave main()
		for(if(a==b) a; end if)
			a;
		end
		for(a; b; c)
			b;
		end while
end fun
#----------------
// no expressions, but semis
fun wave main()
		for(;;)
			a;
		end for
end fun
@good returns
fun wave a()
		wave b;
		return b;
end fun
#--------------
fun int b()
  int a;
  int c;
  return a && b;
end fun
#--------------
fun wave main()
		return funcall(input);
end fun
#--------------
fun int c()
		return a = (b || c == d) + e * -f + !g(h at 9 to 10);
end fun
@bad returns
//1 returning a declaration
fun int a()
		return int a = 5;
end fun
#---------------
//2 returning outside of a function
return var;
fun int main()
  /* stuff */
end fun
#----------------
//3 returning expression list
fun int a()
		return a,b;
end fun
#----------------
//4 returning a statement
fun int a()
		return return x;
end fun
#----------------
//5 no semicolon
fun int a()
		return x
end fun
```

```
@good prints
fun wave a()
		print b;
end fun
#--------------
fun int b()
		print "test " + b;
end fun
#--------------
fun wave main()
		print funcall(input);
end fun
#--------------
fun int c()
		print a = (b || c == d) + e * -f + !g(h at 9 to 10);
end fun
@bad prints
//1 printing a declaration
fun int a()
		print int a = 5;
end fun
#---------------
//2 printing outside of a function
print var;
fun int main()
  /* stuff */
end fun
#----------------
//3 printing expression list
fun int a()
		print a,b;
end fun
#----------------
//4 printing a statement
fun int a()
		print return x;
end fun
#----------------
//5 no semicolon
fun int a()
		print x
end fun
#----------------
//6 two expressions
fun wave main()
		print wooo hoo;
end fun
@good exprs
//1 assign
fun wave main()
		a = b;
		int a = b;
		a = b = c = d;
		1 = 2 = 3 = 4;
		1 = b = 3 = d;
end fun
#----------------
//2 logicals
fun wave main()
		a || b;
```

```
        a && b;
        a || b && c;
        a && b || c;
        1 || 2 && 3;
        1 && 2 || 3;
        a || 2 && c;
        a && 2 || c;
end fun
#----------------
//3 relational
fun wave main()
        a < b;
        a > b;
        a <= b;
        a >= b;
        a == b;
        a != b;
        a < b > c <= d >= e == f != g;
        1 < 2 > 3 <= 4 >= 5 == 6 != 7;
        a < 2 > c <= 4 >= e == 6 != g;
end fun
#----------------
//4 add/sub
fun wave main()
        a + b;
        a - b;
        a + b - c;
        1 + 2 - 3;
        a + 2 - 3;
end fun
#----------------
//5 mult/div/mod
fun wave main()
        a * b;
        a / b;
        a % b;
        a * b / c % d;
        1 * 2 / 3 % 4;
        a * 2 / 3 % d;
end fun
#----------------
//6 unaries
fun wave main()
        !a == b;
        a == !b;
        a == -1ms;
        -2 == b;
        a == -2 + !c;
        !a || -2.654968451 - -3;

        amplof a == 2.6a;
        2Hz == amplof b;
        !a || -2 - -3 > amplof b;
end fun
#----------------
//7 at expression
fun wave main()
        a at b;
        a at 1;
        a at b to c;
```

```
		a at 1 to c;
		a at (b at c to d);
end fun
#----------------
//8 at expression 2
fun wave main()
		w at f1 to f2 at t1 to t2;
end fun
#----------------
//9 fun call
fun wave main()
		a();
		a(b, c);
		a = b(c, d);
		a = b(c+1, -2+!b);
		a() = b(c+1, -2+!b);
end fun
#----------------
//10 literals
fun wave main()
		1;
		a;
		"happy happy joy joy";
end fun
#----------------
//11 mixed
fun wave main()
		a = (b || c == d) + e * -f + !g(h at 9 to 10);
		h at (i at 9 to 10 * !g(-f) * e + (b && c > d)) = a;
end fun
@bad exprs
//bad fun calls: decl argument
fun wave main()
		a(int x);
end fun
#----------------
//bad fun calls: stmt argument
fun wave main()
		a(if(x) a = 1; end if);
end fun
#----------------
//bad fun calls: missing separator
fun wave main()
		a(x y-b);
end fun
@good includes
include "howzit goin??";
fun wave main ()
end fun
#-----------------
include "howzit goin??";
include "whoa two includes";
fun wave main ()
end fun
@bad includes
//1 missing semi
include "howzit goin??"
fun wave main ()
end fun
#-----------------
```

```
//2 no string
include ;
fun wave main ()
end fun
#-----------------
//3 non string
include howzit;
fun wave main ()
end fun
#-----------------
//4 include after a function declaration
include "howzit goin??";
fun wave main ()
end fun
include "whoa two includes";
@good decls
fun wave main()
		int i;
end fun
#-----------------
fun wave main()
		int i;
		float f;
		ampl a;
		freq fr;
		time t;
		wave w;
end fun
#-----------------
fun wave main()
		int i = 1;
end fun
#-----------------
fun wave main()
		int i = (i + 1) / amplof w at f1 to f2 at t1 to t2;
		int t = a + b = c / 2;
end fun
@bad decls
//1 no expression
fun wave main()
		int i =;
end fun
#-----------------
//2 declaring a literal
fun wave main()
		int 1;
end fun
#-----------------
//3 non assign expression
fun wave main()
		int i + 3 / 4;
end fun
#-----------------
//4 assign has expr lhs
fun wave main()
		int a + b / 3 = amplof h / 1;
end fun
#-----------------
//5 at expression lhs
fun wave main()
```

```
            wave w2 at 5Hz to 500Hz = w1;
end fun
#-----------------
//6 Const declarations are not allowed.
fun wave main()
            const wave w;
end fun
#-----------------
//7 Const has been removed from the language.
fun wave main()
            const wave w = 1 + 2;
end fun
@statements
//1 if statement
fun wave main()
            if(a != b)
                        int i = a * b;
                        return i;
            else
                        return a;
end fun
//2 two if statements
fun wave main
```

2r. wavetests.txt

```
/*fun wave main(wave added)
            return input + added;
end fun
fun wave main()
            return input at 10000Hz to 20000Hz;
end fun
*/
fun wave main()
            wave a = input at 20Hz to 500Hz;
            wave b = input at 15000Hz to 20000Hz;
            return a + b;
end fun
/*
fun wave main()
            return input at 100000ms to 150000ms;
end fun
fun wave main()
            wave a = input;
            a at 250ms to 12250ms = input at 0ms to 12000ms;
            return a + input;
end fun
fun wave main()
            return 2.5*input;
end fun
fun wave main(wave added)
            return 2*input + added;
end fun
fun wave main()
            return input - .5;
end fun
fun wave main()
            print " " + amplof input;
            return input;
            end fun

fun wave main()
```

```
		return -(input);
	end fun
	*/
```

3a. ASMLAmplitude.java

```
/**
 *
 */
package asml.walker;
/**
 * @author Frank Smith and Tim Favorite
 *
 */
public class ASMLAmplitude extends Value {
	/** the value of this ASMLAmplitude object */
	protected double mValue;

	/**
	 * Creates a non-storable ASMLAmplitude object from a double value
	 * Example: 2.1a
	 * @param aValue the value to set to this ASMLAmplitude object
	 */
	public ASMLAmplitude(double aValue) {
		mType = Type.AMPL;
		mValue = aValue;
		mIsInitialized = true;
	}

	/**
	 * Creates an ASMLAmplitude object from a double value, name, and constant
boolean
	 * Example: const ampl a = 2.0a
	 * @param aValue the value to set to this ASMLAmplitude object
	 * @param aName the name to set to this ASMLAmplitude object
	 */
	public ASMLAmplitude(double aValue, String aName){
		this(aValue);
		mName = aName;
		mIsStorable = true;
	}

	/**
	 * Creates an ASMLAmplitude with just a name (this is a declaration)
	 * Example: ampl b
	 * @param aName the name of the object
	 * @param aIsConst
	 */
	public ASMLAmplitude(String aName){
		mType = Type.AMPL;
		mName = aName;
		mIsInitialized = false;
		mIsStorable = true;
	}

	@Override
	public Value add(Value rhs) throws ASMLSemanticException {
		if(rhs.getType() == Type.AMPL)
			return new ASMLAmplitude(mValue +
((ASMLAmplitude)rhs).getValue());
		if(rhs.getType() == Type.STRING)
			return new ASMLString(Double.toString(mValue) + "a" +
```

```
                                        ((ASMLString)rhs).getValue());

                        return super.add(rhs);
            }
            @Override
            public Value divide(Value rhs) throws ASMLSemanticException {
                        switch(rhs.getType()){
                                    case Type.INT:
                                                return new ASMLAmplitude(mValue /
((ASMLInteger)rhs).getValue());
                                    case Type.FLOAT:
                                                return new ASMLAmplitude(mValue /
((ASMLFloat)rhs).getValue());
                                    case Type.AMPL:
                                                return new ASMLAmplitude(mValue /
((ASMLAmplitude)rhs).getValue());
                                    default:
                                                return super.divide(rhs);

                        }
            }
            @Override
            public Value multiply(Value rhs) throws ASMLSemanticException {
                        switch(rhs.getType()){
                                    case Type.INT:
                                                return new ASMLAmplitude(mValue *
((ASMLInteger)rhs).getValue());
                                    case Type.FLOAT:
                                                return new ASMLAmplitude(mValue *
((ASMLFloat)rhs).getValue());
                                    case Type.AMPL:
                                                return new ASMLAmplitude(mValue *
((ASMLAmplitude)rhs).getValue());
                                    default:
                                                return super.multiply(rhs);

                        }
            }
            @Override
            public Value relate(Value rhs, String op) throws ASMLSemanticException {
                        double tValue;
                        if(rhs.getType() != Type.AMPL)
                                    return super.relate(rhs,op);
                        tValue = ((ASMLAmplitude)rhs).getValue();
                        if (op.equals("<"))
                                    if (mValue < tValue) return new ASMLInteger(1);
                                    else return new ASMLInteger(0);
                        else if (op.equals("<="))
                                    if (mValue <= tValue) return new ASMLInteger(1);
                                    else return new ASMLInteger(0);
                        else if (op.equals(">"))
                                    if (mValue > tValue) return new ASMLInteger(1);
                                    else return new ASMLInteger(0);
                        else if (op.equals(">="))
                                    if (mValue >= tValue) return new ASMLInteger(1);
                                    else return new ASMLInteger(0);
                        else if (op.equals("=="))
                                    if (mValue == tValue) return new ASMLInteger(1);
                                    else return new ASMLInteger(0);
                        else if (op.equals("!="))
```

```
                        if (mValue != tValue) return new ASMLInteger(1);
                        else return new ASMLInteger(0);
                return super.relate(rhs, op);
        }
        @Override
        public Value subtract(Value rhs) throws ASMLSemanticException {
                if(rhs.getType() != Type.AMPL)
                        return super.subtract(rhs);

                return new ASMLAmplitude(mValue - ((ASMLAmplitude)rhs).getValue());
        }

        /**
         * Gets value of this ASMLAmplitude object
         * @return the value
         * @throws ASMLSemanticException
         */
        public double getValue() throws ASMLSemanticException {
                if(this.mIsInitialized)
                        return mValue;
                throw new ASMLSemanticException("Cannot return value for uninitialized
identifier");
        }
}
```

3b. ASMLControl.java

```
/**
 *
 */
package asml.walker;
import java.util.*;
import org.antlr.runtime.RecognitionException;
import org.antlr.runtime.tree.CommonTree;
import org.antlr.runtime.tree.CommonTreeNodeStream;
import asml.ASMLWalker;
import asml.walker.streams.SubwaveFloatAudioInputStream;
/**
 * @author Frank A. Smith and Tim Favorite
 *
 */
public class ASMLControl {
        Stack<FunctionRecord> mActivationRecord        = null;
        HashMap<String, FunctionRecord> mFunctionMap = null;
        ArrayList<String> mCmdLnArgs                   = null;
        String mInput                                  = null;
        String mOutput                                 = null;
        CommonTreeNodeStream mTNStream                 = null;
        ASMLWalker mWalker                             = null;

//      public ASMLControl(HashMap<String, FunctionRecord> aFunctionMap,
//                      String input, String output){
//              this(aFunctionMap, new ArrayList<String>(), input, output);
//      }

        public ASMLControl(HashMap<String, FunctionRecord> aFunctionMap,
                        ArrayList<String> aCmdLnArgs, String input, String output,
ASMLWalker walker){
                mFunctionMap = aFunctionMap;
                mCmdLnArgs   = aCmdLnArgs;
                mActivationRecord = new Stack<FunctionRecord>();
                mInput = input;
```

```
			mOutput = output;
			mWalker = walker;
		}

		public void doCallMain() throws ASMLSemanticException{
			if(!mFunctionMap.containsKey("main"))
				throw new ASMLSemanticException("There is no main function
and therefore program" +
						"cannot be run.");
			FunctionRecord tMain = mFunctionMap.get("main");
			tMain.passParamString(mCmdLnArgs);
			tMain.addSymbol(new ASMLWave("input", mInput));
			mActivationRecord.push(tMain);
			mTNStream.push(mTNStream.getNodeIndex(tMain.getBlockRt()));
			try {
				mWalker.block();
			} catch (RecognitionException e) {
				throw new ASMLSemanticException(e.getMessage());
			}
			mTNStream.pop();
		}

		public Value doCallFunction(String name, ArrayList<Value> aActualParams) throws
ASMLSemanticException{
			if(!mFunctionMap.containsKey(name))
				throw new ASMLSemanticException("There is no " + name + "
function.");
			FunctionRecord tFun = new FunctionRecord(mFunctionMap.get(name));
			tFun.passParamValue(aActualParams);
			mActivationRecord.push(tFun);
			mTNStream.push(mTNStream.getNodeIndex(tFun.getBlockRt()));
			try {
				mWalker.block();
			} catch (RecognitionException e) {
				throw new ASMLSemanticException(e.getMessage());
			}
			mTNStream.pop();
			mActivationRecord.pop();
			return tFun.getRetVal();
		}

		public Value doAssign(Value aLHS, Value aRHS) throws ASMLSemanticException{
			if (aLHS.getType() != aRHS.getType())
				throw new ASMLSemanticException("Type of lhs value does not
match"+
				" type of rhs value.");
			if (!aLHS.isStorable())
				throw new ASMLSemanticException("Left hand value is not
storable.");

			Value tVal = null;
			String tName = aLHS.getName();

			if(aRHS.getType() == Type.WAVE && ((ASMLWave)aRHS).isAtResult()){
				ASMLWave tWave = (ASMLWave)aRHS;
				if(tWave.getEndFreq().getValue() <
tWave.getStartFreq().getValue() ||
						tWave.getEndTime().getValue() <
tWave.getStartTime().getValue())
							throw new
```

```
ASMLSemanticException("Second argument must be greater than or equal to first argument
in at expression!");

                              if(tWave.getStartFreq().getValue() > -1 &&
tWave.getStartTime().getValue() > -1)
                                        aRHS = tWave.at(tWave.getStartFreq(),
tWave.getEndFreq()).at(tWave.getStartTime(), tWave.getEndTime());
                              else if (tWave.getStartFreq().getValue() > -1)
                                        aRHS = tWave.at(tWave.getStartFreq(),
tWave.getEndFreq());
                              else if (tWave.getStartTime().getValue() > -1)
                                        aRHS = tWave.at(tWave.getStartTime(),
tWave.getEndTime());
                    }
                    switch(aLHS.getType()){
                    case Type.INT:          tVal = new
ASMLInteger(((ASMLInteger)aRHS).getValue(), tName);
                                                                      break;
                    case Type.FLOAT:        tVal = new
ASMLFloat(((ASMLFloat)aRHS).getValue(), tName);
                                                                      break;
                    case Type.AMPL:         tVal = new
ASMLAmplitude(((ASMLAmplitude)aRHS).getValue(), tName);
                                                                      break;
                    case Type.FREQ:         tVal = new
ASMLFrequency(((ASMLFrequency)aRHS).getValue(), tName);
                                                                      break;
                    case Type.TIME:         tVal = new
ASMLTime(((ASMLTime)aRHS).getValue(), tName);
                                                                      break;
                    case Type.WAVE:         if (((ASMLWave)aLHS).isAtResult()){
                                                                                ASMLWave
tWave = (ASMLWave)aLHS;
                                                                                if
(tWave.getStartFreq().getValue() > -1)

System.out.println("At expressions on left hand side of "+

"expressions are not currently supported, at will be ignored.");
                                                                                if
(tWave.getEndTime().getValue() < tWave.getStartTime().getValue()){
                                                                                          throw
new ASMLSemanticException("Second argument must be greater than or equal to first
argument in at expression!");
                                                                                } else if
(tWave.getStartTime().getValue() > -1){
                                                                                          tVal =
new ASMLWave(new SubwaveFloatAudioInputStream(tWave.getValue().getFormat(),
tWave.getValue(), ((ASMLWave)aRHS).getValue(),

tWave.getStartTime().getValue(), tWave.getEndTime().getValue()), tWave.getName());
                                                                                }
                                                                      }
                                                                      else tVal = new
ASMLWave(((ASMLWave)aRHS).getValue(), tName);
                    }

                    editSymbol(tVal);
                    return tVal;
          }
```

```
	public void doDeclare(String aName, String aType, Value aVal) throws
ASMLSemanticException{
		Value tVal = null;
		if (Type.getType(aType) != aVal.getType())
			throw new ASMLSemanticException("Type of assigned value
does not match"+
					" type declarator.");

		if(aVal.getType() == Type.WAVE && ((ASMLWave)aVal).isAtResult()){
			ASMLWave tWave = (ASMLWave)aVal;
			if(tWave.getEndFreq().getValue() <
tWave.getStartFreq().getValue() ||
					tWave.getEndTime().getValue() <
tWave.getStartTime().getValue())
						throw new
ASMLSemanticException("Second argument must be greater than or equal to first argument
in at expression!");

			if(tWave.getStartFreq().getValue() > -1 &&
tWave.getStartTime().getValue() > -1)
				aVal = tWave.at(tWave.getStartFreq(),
tWave.getEndFreq()).at(tWave.getStartTime(), tWave.getEndTime());
			else if (tWave.getStartFreq().getValue() > -1)
				aVal = tWave.at(tWave.getStartFreq(),
tWave.getEndFreq());
			else if (tWave.getStartTime().getValue() > -1)
				aVal = tWave.at(tWave.getStartTime(),
tWave.getEndTime());
		}

		switch(Type.getType(aType)){
		case Type.INT:		tVal = new
ASMLInteger(((ASMLInteger)aVal).getValue(), aName);
							break;
		case Type.FLOAT:	tVal = new
ASMLFloat(((ASMLFloat)aVal).getValue(), aName);
							break;
		case Type.AMPL:		tVal = new
ASMLAmplitude(((ASMLAmplitude)aVal).getValue(), aName);
							break;
		case Type.FREQ:		tVal = new
ASMLFrequency(((ASMLFrequency)aVal).getValue(), aName);
							break;
		case Type.TIME:		tVal = new
ASMLTime(((ASMLTime)aVal).getValue(), aName);
							break;
		case Type.WAVE:		tVal = new ASMLWave((ASMLWave)aVal,
aName);
		}

		addSymbol(tVal);
	}

	public void doDeclare(String aName, String aType) throws
ASMLSemanticException{
		Value tVal = null;

		switch(Type.getType(aType)){
		case Type.INT:		tVal = new ASMLInteger(aName);
```

```
                                                        break;
            case Type.FLOAT:        tVal = new ASMLFloat(aName);
                                                        break;
            case Type.AMPL:         tVal = new ASMLAmplitude(aName);
                                                        break;
            case Type.FREQ:         tVal = new ASMLFrequency(aName);
                                                        break;
            case Type.TIME:         tVal = new ASMLTime(aName);
                                                        break;
            case Type.WAVE:         tVal = new ASMLWave(aName);
            }

            addSymbol(tVal);
        }

        /**
          * The pre-condition for doWhile is that the while condition is initially valid.  That
          * is, the aExpr is tested in the walker first and this function is only called if it
          * is true (non-zero).
          * We must also know that aExpr evaluates to an integer.  That is something that
must
          * be checked in the walker.
          * @param aExpr Pointer to the expr subtree that is checked with each loop.  Must
          * evaluate to an integer.
          * @param aBlock Pointer to the block subtree that is executed with every loop
          * @throws Exception
        */
        public void doWhile(CommonTree aExpr, CommonTree aBlock) throws Exception{
            ASMLInteger tVal = null;

            do{
                mTNStream.push(mTNStream.getNodeIndex(aBlock));
                mWalker.block();
                mTNStream.pop();

                //return has been called
                if(isCurrentFunctionLocked())
                    break;

                mTNStream.push(mTNStream.getNodeIndex(aExpr));
                tVal = (ASMLInteger) mWalker.expr();
                mTNStream.pop();
            }while(tVal.getValue() != 0);
        }

        /**
          * The pre-condition for doFor is that the while condition is initially valid.  That
          * is, the aExpr is tested in the walker first and this function is only called if it
          * is true (non-zero).
          * We must also know that aExpr evaluates to an integer.  That is something that
must
          * be checked in the walker.
          * @param aCheck Pointer to the expr subtree that is checked with each loop.
Must
          * evaluate to an integer.
          * @param aUpdate Pointer to the expr subtree that is executed at the very end of
          * each loop.
          * @param aBlock Pointer to the block subtree that is executed with every loop
          * @throws Exception
        */
```

```
	public void doFor(CommonTree aCheck, CommonTree aUpdate, CommonTree
aBlock)
			throws Exception{
			ASMLInteger tVal = null;

			do{
				mTNStream.push(mTNStream.getNodeIndex(aBlock));
				mWalker.block();
				mTNStream.pop();

				//return has been called
				if(isCurrentFunctionLocked())
					break;

				mTNStream.push(mTNStream.getNodeIndex(aUpdate));
				mWalker.expr();
				mTNStream.pop();
				mTNStream.push(mTNStream.getNodeIndex(aCheck));
				tVal = (ASMLInteger) mWalker.expr();
				mTNStream.pop();
			}while(tVal.getValue() != 0);
	}

	public void doReturn(Value aVal) throws ASMLSemanticException{
			Value tVal = null;

			if(aVal.getType() == Type.WAVE && ((ASMLWave)aVal).isAtResult()){
				ASMLWave tWave = (ASMLWave)aVal;
				if(tWave.getEndFreq().getValue() <
tWave.getStartFreq().getValue() ||
							tWave.getEndTime().getValue() <
tWave.getStartTime().getValue())
								throw new
ASMLSemanticException("Second argument must be greater than or equal to first argument
in at expression!");

				if(tWave.getStartFreq().getValue() > -1 &&
tWave.getStartTime().getValue() > -1)
					aVal = tWave.at(tWave.getStartFreq(),
tWave.getEndFreq()).at(tWave.getStartTime(), tWave.getEndTime());
				else if (tWave.getStartFreq().getValue() > -1)
					aVal = tWave.at(tWave.getStartFreq(),
tWave.getEndFreq());
				else if (tWave.getStartTime().getValue() > -1)
					aVal = tWave.at(tWave.getStartTime(),
tWave.getEndTime());
			}

			switch(aVal.getType()){
			case Type.INT:
				tVal = new ASMLInteger(((ASMLInteger)aVal).getValue());
				break;
			case Type.FLOAT:
				tVal = new ASMLFloat(((ASMLFloat)aVal).getValue());
				break;
			case Type.AMPL:
				tVal = new
ASMLAmplitude(((ASMLAmplitude)aVal).getValue());
				break;
			case Type.FREQ:
```

```
				tVal = new
ASMLFrequency(((ASMLFrequency)aVal).getValue());
				break;
			case Type.TIME:
				tVal = new ASMLTime(((ASMLTime)aVal).getValue());
				break;
			case Type.WAVE:
				tVal = new ASMLWave(((ASMLWave)aVal).getValue());
			}

			mActivationRecord.peek().setRetVal(tVal);

			mActivationRecord.peek().setCanExecute(false);
			if(mActivationRecord.peek().getName().equals("main"))
				((ASMLWave)tVal).write(mOutput);
		}

		public Value doAt(Value aWave, Value aValueStart, Value aValueEnd) throws
ASMLSemanticException {
			if(aWave.getType() != Type.WAVE)
				throw new ASMLSemanticException("First argument of At
expression can only be a wave.");

			ASMLWave tWave = (ASMLWave) aWave;

			if (aValueStart.getType() == Type.FREQ && aValueEnd.getType() ==
Type.FREQ)
				tWave.setIsAtResult((ASMLFrequency)aValueStart,
(ASMLFrequency)aValueEnd);
			else if (aValueStart.getType() == Type.TIME && aValueEnd.getType()
== Type.TIME)
				tWave.setIsAtResult((ASMLTime)aValueStart,
(ASMLTime)aValueEnd);
			else
				throw new ASMLSemanticException("Second and third
arguments of at expression can " +
						"only be frequencies or times and must
match.");
			return tWave;
		}

		public Value doAt(Value aWave, Value aValue) throws ASMLSemanticException {
			return doAt(aWave, aValue, aValue);
		}

		public void enterScope() throws ASMLSemanticException {
			if(mActivationRecord.empty())
				throw new ASMLSemanticException("Activation Record is
currently empty.");
			mActivationRecord.peek().enterScope();
		}

		public void exitScope() throws ASMLSemanticException{
			if(mActivationRecord.empty())
				throw new ASMLSemanticException("Activation Record is
currently empty.");
			mActivationRecord.peek().exitScope();
		}
		public void setStream(CommonTreeNodeStream stream) {
			mTNStream = stream;
		}
```

```
	public void addSymbol(Value aVal) throws ASMLSemanticException{
		mActivationRecord.peek().addSymbol(aVal);
	}

	public void editSymbol(Value aVal) throws ASMLSemanticException{
		mActivationRecord.peek().editSymbol(aVal);
	}

	public Value getSymbol(String aName) throws ASMLSemanticException{
		return mActivationRecord.peek().getSymbol(aName);
	}

	public boolean isCurrentFunctionLocked(){
		return !mActivationRecord.peek().canExecute();
	}
}
```

3c. ASMLError.java

```
package asml.walker;
/**
 *
 * @author Frank Smith and Tim Favorite
 *
 */
public class ASMLError extends Value {
	private String mValue;

	public ASMLError(String aValue){
		mValue = aValue;
	}

	public String getValue(){
		return mValue;
	}
}
```

3d. ASMLFloat.java

```
/**
 *
 */
package asml.walker;
/**
 * @author Frank Smith and Tim Favorite
 *
 */
public class ASMLFloat extends Value {
	/** The value representing the ASMLFloat value */
	protected double mValue;

	/**
	 * Constructs an ASMLFloat object based on a float value (this is a literal)
	 * Example: 2.5
	 * @param aValue the value of this ASMLFloat
	 */
	public ASMLFloat(double aValue) {
		mType = Type.FLOAT;
		mValue = aValue;
		mIsInitialized = true;
	}

	/**
	 * Constructs an ASMLFloat object based on a float value, a name, and a boolean
```

```
	 * representing its constancy.
	 * Example: const float a = 2.5
	 * @param aValue the value of this ASMLFloat
	 * @param aName the name of this ASMLFloat
	 */
	public ASMLFloat(double aValue, String aName){
		this(aValue);
		mName = aName;
		mIsStorable = true;
	}

	/**
	 * Constructs an ASMLFloat object based on a name and a boolean representing
	 * its constancy (declaration)
	 * Example: float a
	 *
	 * @param aName the name of this ASMLFloat
	 * @param aIsConst
	 */
	public ASMLFloat(String aName){
		mType = Type.FLOAT;
		mName = aName;
		mIsInitialized = false;
		mIsStorable = true;
	}

	/**
	 * Adds rhs value to this ASMLFloat. Only ints, floats, strings, and waves may be
added.
	 * In case of a string rhs type it concatenates the value of this ASMLFloat to the
rhs
	 * string. In case of a wave rhs type it shifts the wave up by this ASMLFloat's
value.
	 * @param rhs the right hand side of the add operation
	 * @return the result of the add expression
	 */
	public Value add(Value rhs) throws ASMLSemanticException {
		switch(rhs.getType()){
			case Type.INT:
				return new ASMLFloat(mValue +
((ASMLInteger)rhs).getValue());
			case Type.FLOAT:
				return new ASMLFloat(mValue +
((ASMLFloat)rhs).getValue());
			case Type.STRING:
				return new ASMLString(Double.toString(mValue) +
						((ASMLString)rhs).getValue());
			case Type.WAVE:
				return ((ASMLWave)rhs).add(this);
			default:
				return super.add(rhs);

		}
	}
	/**
	 * Does a division operation - only ints and floats allowed for rhs value.
	 * @param rhs the right hand side of the operation
	 * @return the result of the operation
	 */
	public Value divide(Value rhs) throws ASMLSemanticException {
```

```
			switch(rhs.getType()){
				case Type.INT:
					return new ASMLFloat(mValue /
((ASMLInteger)rhs).getValue());
				case Type.FLOAT:
					return new ASMLFloat(mValue /
((ASMLFloat)rhs).getValue());
				default:
					return super.add(rhs);

			}
		}
		/**
		 * Does a multiplication operation - ints, floats, frequencies, times, amplitudes,
and
		 * waves are allowed as right hand side operands. For waves, this is a
multiplication
		 * of its amplitude by a factor of this ASMLFloat's value.
		 * @param rhs The right hand side operand
		 * @return The result of the multiplication
		 */
		public Value multiply(Value rhs) throws ASMLSemanticException {
			switch(rhs.getType()){
				case Type.INT:
					return new ASMLFloat(mValue *
((ASMLInteger)rhs).getValue());
				case Type.FLOAT:
					return new ASMLFloat(mValue *
((ASMLFloat)rhs).getValue());
				case Type.FREQ:
					return new ASMLFrequency(mValue *
((ASMLFrequency)rhs).getValue());
				case Type.TIME:
					return new ASMLTime(mValue *
((ASMLTime)rhs).getValue());
				case Type.AMPL:
					return new ASMLAmplitude(mValue *
((ASMLAmplitude)rhs).getValue());
				case Type.WAVE:
					ASMLWave tW = (ASMLWave) rhs;
					return tW.multiply(this);
				default:
					return super.multiply(rhs);

			}
		}
		/**
		 * Negates this ASMLFloat's value.
		 * @return the negative of this ASMLFloat
		 */
		public Value negate() throws ASMLSemanticException {
			return new ASMLFloat(-mValue);
		}
		/**
		 * Does a relational operation on this ASMLFloat and another number
		 * @param rhs the right hand side of the relational operation (must be int or float)
		 * @param op the operator (<, <=, >, >=, ==, !=)
		 * @return an ASMLInteger with a value of 0 (false) or 1 (true)
		 * @throws ASMLSemanticException if rhs was not an int or float or if operator
was invalid
```

```
	 */
	public Value relate(Value rhs, String op) throws ASMLSemanticException {
		double tValue;
		switch(rhs.getType()){
			case Type.INT:
				tValue = ((ASMLInteger)rhs).getValue();
				break;
			case Type.FLOAT:
				tValue = ((ASMLFloat)rhs).getValue();
				break;
			default:
				return super.relate(rhs,op);
		}
		if (op.equals("<"))
			if (mValue < tValue) return new ASMLInteger(1);
			else return new ASMLInteger(0);
		else if (op.equals("<="))
			if (mValue <= tValue) return new ASMLInteger(1);
			else return new ASMLInteger(0);
		else if (op.equals(">"))
			if (mValue > tValue) return new ASMLInteger(1);
			else return new ASMLInteger(0);
		else if (op.equals(">="))
			if (mValue >= tValue) return new ASMLInteger(1);
			else return new ASMLInteger(0);
		else if (op.equals("=="))
			if (mValue == tValue) return new ASMLInteger(1);
			else return new ASMLInteger(0);
		else if (op.equals("!="))
			if (mValue != tValue) return new ASMLInteger(1);
			else return new ASMLInteger(0);
		return super.relate(rhs, op);
	}
	/**
	 * Performs a subtraction operation
	 * @param rhs the right hand side of the operation (must be int or float)
	 * @return the result of the subtraction
	 * @throws ASMLSemanticException if rhs was not an int or float
	 */
	public Value subtract(Value rhs) throws ASMLSemanticException {
		switch(rhs.getType()){
			case Type.INT:
				return new ASMLFloat(mValue - 
((ASMLInteger)rhs).getValue());
			case Type.FLOAT:
				return new ASMLFloat(mValue - 
((ASMLFloat)rhs).getValue());
			default:
				return super.subtract(rhs);

		}
	}
	/**
	 * Get the value of this ASMLFloat
	 * @return the value of this ASMLFloat
	 */
	public double getValue() throws ASMLSemanticException {
		if(this.mIsInitialized)
			return mValue;
		throw new ASMLSemanticException("Cannot return value for uninitialized
```

```
identifier");
          }
}
```

3e. ASMLFrequency.java

```
/**
 *
 */
package asml.walker;
/**
 * @author Frank Smith and Tim Favorite
 *
 */
public class ASMLFrequency extends Value {

          /** the value of this ASMLFrequency object */
          protected double mValue;

          /**
           * Constructs a non-storable ASMLFrequency object
           * Example: 450Hz
           * @param aValue the vale of this ASMLFrequency object
           */
          public ASMLFrequency(double aValue) {
                    mType = Type.FREQ;
                    mValue = aValue;
                    mIsInitialized = true;
          }

          /**
           * Constructs an ASMLFrequency object with value, name, and constant specifier
           * (declaration and assignment)
           * Example freq f = 450Hz
           * @param aValue the value of this ASMLFrequency object
           * @param aName the name of this ASMLFrequency object
           */
          public ASMLFrequency(double aValue, String aName){
                    this(aValue);
                    mName = aName;
                    mIsStorable = true;
          }

          /** Constructs an ASMLFrequency object with a name (declaration)
           * Example: freq f
           * @param aName the name of the object
           * @param aIsConst
           */
          public ASMLFrequency(String aName){
                    mType = Type.FREQ;
                    mName = aName;
                    mIsInitialized = false;
                    mIsStorable = true;
          }

          @Override
          public Value add(Value rhs) throws ASMLSemanticException {
                    if(rhs.getType() == Type.FREQ)
                              return new ASMLFrequency(mValue +
((ASMLFrequency)rhs).getValue());
                    if(rhs.getType() == Type.STRING)
                              return new ASMLString(Double.toString(mValue) + "Hz" +
```

```
                                        ((ASMLString)rhs).getValue());
                return super.add(rhs);
        }
        @Override
        public Value divide(Value rhs) throws ASMLSemanticException {
                switch(rhs.getType()){
                        case Type.INT:
                                return new ASMLFrequency(mValue /
((ASMLInteger)rhs).getValue());
                        case Type.FLOAT:
                                return new ASMLFrequency(mValue /
((ASMLFloat)rhs).getValue());
                        case Type.FREQ:
                                return new ASMLFloat(mValue /
((ASMLFrequency)rhs).getValue());
                        default:
                                return super.divide(rhs);

                }
        }
        @Override
        public Value multiply(Value rhs) throws ASMLSemanticException {
                switch(rhs.getType()){
                        case Type.INT:
                                return new ASMLFrequency(mValue *
((ASMLInteger)rhs).getValue());
                        case Type.FLOAT:
                                return new ASMLFrequency(mValue *
((ASMLFloat)rhs).getValue());
                        default:
                                return super.multiply(rhs);

                }
        }
        @Override
        public Value relate(Value rhs, String op) throws ASMLSemanticException {
                double tValue;
                if(rhs.getType() != Type.FREQ)
                        return super.relate(rhs,op);
                tValue = ((ASMLFrequency)rhs).getValue();
                if (op.equals("<"))
                        if (mValue < tValue) return new ASMLInteger(1);
                        else return new ASMLInteger(0);
                else if (op.equals("<="))
                        if (mValue <= tValue) return new ASMLInteger(1);
                        else return new ASMLInteger(0);
                else if (op.equals(">"))
                        if (mValue > tValue) return new ASMLInteger(1);
                        else return new ASMLInteger(0);
                else if (op.equals(">="))
                        if (mValue >= tValue) return new ASMLInteger(1);
                        else return new ASMLInteger(0);
                else if (op.equals("=="))
                        if (mValue == tValue) return new ASMLInteger(1);
                        else return new ASMLInteger(0);
                else if (op.equals("!="))
                        if (mValue != tValue) return new ASMLInteger(1);
                        else return new ASMLInteger(0);
                return super.relate(rhs, op);
        }
```

```
		@Override
		public Value subtract(Value rhs) throws ASMLSemanticException {
			if(rhs.getType() != Type.FREQ)
				return super.subtract(rhs);

			return new ASMLFrequency(mValue - ((ASMLFrequency)rhs).getValue());
		}
		/** Gets the value of this ASMLFrequency object */
		public double getValue() throws ASMLSemanticException {
			if(this.mIsInitialized)
				return mValue;
			throw new ASMLSemanticException("Cannot return value for uninitialized
identifier");
		}
}
```

3f. ASMLInteger.java

```
/**
 *
 */
package asml.walker;

/**
 * This class represents the Integer type.
 * It extends Value and all applicable operations.
 * @author Frank A. Smith & Tim Favorite
 *
 */
public class ASMLInteger extends Value {

   /** The value of this ASMLInteger object. */
   protected int mValue;

   /** Constructs an ASMLInteger object with only a value.
    * Example: 5
    * @param aValue the value of this ASMLInteger object.
    */
   public ASMLInteger(int aValue) {
	mType          = Type.INT;
	mValue         = aValue;
	mIsInitialized = true;
   }

   /**
    * Copy constructor.
    * @param val the value (must be ASMLInteger) to set to this ASMLInteger object.
    * @throws ASMLSemanticException if val is not of type ASMLInteger
    */
   public ASMLInteger(Value val) throws ASMLSemanticException{
	if (val.getType() != Type.INT)
		throw new ASMLSemanticException("Cannot assign non-integer value to an
integer.");
	else {
		mType = Type.INT;
		mValue = ((ASMLInteger)val).getValue();
		mIsInitialized = val.isInitialized();
		mIsStorable = val.isStorable();
		mName = val.getName();
	}
   }
```

```
/**
 * Constructs an ASMLInteger with specified value, name and constant specifier.
 * (declaration and assignment)
 * Example: const int a = 3
 * @param aValue the value of this ASMLInteger object.
 * @param aName the name of this ASMLInteger object.
 */
public ASMLInteger(int aValue, String aName){
        this(aValue);
        mName       = aName;
        mIsStorable = true;
}

/**
 * Constructs an ASMLInteger with specified name (declaration only)
 * @param aName the name of this ASMLInteger object.
 * @param aIsConst
 */
public ASMLInteger(String aName){
        mType          = Type.INT;
        mName          = aName;
        mIsStorable    = true;
        mIsInitialized = false;
}


@Override
public Value add(Value rhs) throws ASMLSemanticException {
        switch(rhs.getType()){
                case Type.INT:
                        return new ASMLInteger(mValue + ((ASMLInteger)rhs).getValue());
                case Type.FLOAT:
                        return new ASMLFloat(mValue + ((ASMLFloat)rhs).getValue());
                case Type.STRING:
                        return new ASMLString(Integer.toString(mValue) +
                                        ((ASMLString)rhs).getValue());
                case Type.WAVE:
                        return ((ASMLWave)rhs).add(this);
                default:
                        return super.add(rhs);

        }
}

@Override
public Value divide(Value rhs) throws ASMLSemanticException {
        switch(rhs.getType()){
                case Type.INT:
                        return new ASMLInteger(mValue / ((ASMLInteger)rhs).getValue());
                case Type.FLOAT:
                        return new ASMLFloat(mValue / ((ASMLFloat)rhs).getValue());
                default:
                        return super.divide(rhs);

        }
}

@Override
/**
```

```
	 * Zeroes are false, nonzeroes are true.
	 */
	public Value logic(Value rhs, String op) throws ASMLSemanticException {
		if (rhs.getType() == Type.INT){
			if (op.equals("||")){
				if (mValue == 0 && ((ASMLInteger)rhs).getValue() == 0)
					return new ASMLInteger(0);
				else return new ASMLInteger(1);
			}
			else if (op.equals("&&")){
				if (mValue != 0 && ((ASMLInteger)rhs).getValue() != 0)
					return new ASMLInteger(1);
				else return new ASMLInteger(0);
			}
		}
		return super.logic(rhs, op);
	}

	@Override
	public Value mod(Value rhs) throws ASMLSemanticException {
		if (rhs.getType() == Type.INT){
			return new ASMLInteger(mValue % ((ASMLInteger)rhs).getValue());
		}
		return super.mod(rhs);
	}

	@Override
	public Value multiply(Value rhs) throws ASMLSemanticException {
		switch(rhs.getType()){
			case Type.INT:
				return new ASMLInteger(mValue * ((ASMLInteger)rhs).getValue());
			case Type.FLOAT:
				return new ASMLFloat(mValue * ((ASMLFloat)rhs).getValue());
			case Type.FREQ:
				return new ASMLFrequency(mValue *
((ASMLFrequency)rhs).getValue());
			case Type.TIME:
				return new ASMLTime(mValue * ((ASMLTime)rhs).getValue());
			case Type.AMPL:
				return new ASMLAmplitude(mValue *
((ASMLAmplitude)rhs).getValue());
			case Type.WAVE:
				ASMLWave tW = (ASMLWave) rhs;
				return tW.multiply(this);
			default:
				return super.multiply(rhs);

		}
	}

	@Override
	public Value negate() throws ASMLSemanticException {
		return new ASMLInteger(-mValue);
	}

	@Override
	public Value not() throws ASMLSemanticException {
		if (mValue == 0) return new ASMLInteger(1);
		else return new ASMLInteger(0);
	}
```

```
    @Override
    public Value relate(Value rhs, String op) throws ASMLSemanticException {
            double tValue;
            switch(rhs.getType()){
                    case Type.INT:
                            tValue = ((ASMLInteger)rhs).getValue();
                            break;
                    case Type.FLOAT:
                            tValue = ((ASMLFloat)rhs).getValue();
                            break;
                    default:
                            return super.relate(rhs,op);
            }

            if (op.equals("<"))
                    if (mValue < tValue) return new ASMLInteger(1);
                    else return new ASMLInteger(0);
            else if (op.equals("<="))
                    if (mValue <= tValue) return new ASMLInteger(1);
                    else return new ASMLInteger(0);
            else if (op.equals(">"))
                    if (mValue > tValue) return new ASMLInteger(1);
                    else return new ASMLInteger(0);
            else if (op.equals(">="))
                    if (mValue >= tValue) return new ASMLInteger(1);
                    else return new ASMLInteger(0);
            else if (op.equals("=="))
                    if (mValue == tValue) return new ASMLInteger(1);
                    else return new ASMLInteger(0);
            else if (op.equals("!="))
                    if (mValue != tValue) return new ASMLInteger(1);
                    else return new ASMLInteger(0);
            return super.relate(rhs, op);
    }

    @Override
    public Value subtract(Value rhs) throws ASMLSemanticException {
            switch(rhs.getType()){
                    case Type.INT:
                            return new ASMLInteger(mValue - ((ASMLInteger)rhs).getValue());
                    case Type.FLOAT:
                            return new ASMLFloat(mValue - ((ASMLFloat)rhs).getValue());
                    default:
                            return super.subtract(rhs);

            }
    }

    /** Gets the value of this ASMLInteger object. */
    public int getValue() throws ASMLSemanticException {
            if(this.mIsInitialized)
                    return mValue;
            throw new ASMLSemanticException("Cannot return value for uninitialized
identifier");
    }

}
```

3g. ASMLSemanticException.java

```
package asml.walker;

/**
 * This class represents all semantic exceptions encountered while walking the ASML tree.
Any
 * semantic error should throw this exception.
 * @author Frank Smith and Tim Favorite
 *
 */
public class ASMLSemanticException extends Exception{
   private static final long serialVersionUID = 1837430462181439821L;

   /** Constructs an ASMLSemanticException with the given message string. */
   public ASMLSemanticException(String msg){
           super(msg);
   }
}
```

3h. ASMLString.java

```
/**
 *
 */
package asml.walker;

/**
 * Represents string literals
 * @author Frank Smith and Tim Favorite
 *
 */
public class ASMLString extends Value {
   /** This contains the value of the ASMLString object.     */
   protected String mValue;

   /** Constructs an ASMLString
    *  */
   public ASMLString(String aValue) {
           mType = Type.STRING;
           mValue = aValue;
           mIsInitialized = true;
   }

   public ASMLString(Value aValue) throws ASMLSemanticException{
           if(aValue.getType() != Type.STRING){
                   throw new ASMLSemanticException("Cannot assign non-string value to a
string.");
           } else {
                   mType = Type.STRING;
                   mValue = ((ASMLString)aValue).getValue();
                   mIsInitialized = aValue.isInitialized();
                   mIsStorable = aValue.isStorable();
                   mName = aValue.getName();
           }
   }

   public Value add(Value rhs) throws ASMLSemanticException{
           switch(rhs.getType()){
           case Type.AMPL:
                   return new ASMLString(mValue +
Double.toString(((ASMLAmplitude)rhs).getValue()) + "a");
           case Type.FLOAT:
```

```
			return new ASMLString(mValue +
Double.toString(((ASMLFloat)rhs).getValue()));
		case Type.FREQ:
			return new ASMLString(mValue +
Double.toString(((ASMLFrequency)rhs).getValue()) + "Hz");
		case Type.INT:
			return new ASMLString(mValue +
Integer.toString(((ASMLInteger)rhs).getValue()));
		case Type.STRING:
			return new ASMLString(mValue + ((ASMLString)rhs).getValue());
		case Type.TIME:
			return new ASMLString(mValue +
Double.toString(((ASMLTime)rhs).getValue()) + "ms");
		default:
			return super.add(rhs);
		}
	}

	public String getValue() throws ASMLSemanticException {
		if(this.mIsInitialized)
			return mValue;
		throw new ASMLSemanticException("Cannot return value for uninitialized
identifier");
	}
}
```

3i. ASMLTime.java

```
/**
 *
 */
package asml.walker;

/**
 * @author Frank Smith and Tim Favorite
 *
 */
public class ASMLTime extends Value {

	protected double mValue;

	public ASMLTime(double aValue) {
		mType = Type.TIME;
		mValue = aValue;
		mIsInitialized = true;
	}

	public ASMLTime(double aValue, String aName){
		this(aValue);
		mName = aName;
		mIsStorable = true;
	}

	public ASMLTime(String aName){
		mType          = Type.TIME;
		mName          = aName;
		mIsInitialized = false;
		mIsStorable    = true;
	}
```

```
@Override
public Value add(Value rhs) throws ASMLSemanticException {
        if(rhs.getType() == Type.TIME)
                return new ASMLTime(mValue + ((ASMLTime)rhs).getValue());
        if(rhs.getType() == Type.STRING)
                return new ASMLString(Double.toString(mValue) + "ms" +
                                ((ASMLString)rhs).getValue());

        return super.add(rhs);
}

@Override
public Value divide(Value rhs) throws ASMLSemanticException {
        switch(rhs.getType()){
                case Type.INT:
                        return new ASMLTime(mValue / ((ASMLInteger)rhs).getValue());
                case Type.FLOAT:
                        return new ASMLTime(mValue / ((ASMLFloat)rhs).getValue());
                case Type.TIME:
                        return new ASMLFloat(mValue / ((ASMLTime)rhs).getValue());
                default:
                        return super.divide(rhs);

        }
}

@Override
public Value multiply(Value rhs) throws ASMLSemanticException {
        switch(rhs.getType()){
                case Type.INT:
                        return new ASMLTime(mValue * ((ASMLInteger)rhs).getValue());
                case Type.FLOAT:
                        return new ASMLTime(mValue * ((ASMLFloat)rhs).getValue());
                default:
                        return super.multiply(rhs);

        }
}

@Override
public Value relate(Value rhs, String op) throws ASMLSemanticException {
        double tValue;
        if(rhs.getType() != Type.TIME)
                return super.relate(rhs,op);

        tValue = ((ASMLTime)rhs).getValue();
        if (op.equals("<"))
                if (mValue < tValue) return new ASMLInteger(1);
                else return new ASMLInteger(0);
        else if (op.equals("<="))
                if (mValue <= tValue) return new ASMLInteger(1);
                else return new ASMLInteger(0);
        else if (op.equals(">"))
                if (mValue > tValue) return new ASMLInteger(1);
                else return new ASMLInteger(0);
        else if (op.equals(">="))
                if (mValue >= tValue) return new ASMLInteger(1);
                else return new ASMLInteger(0);
        else if (op.equals("=="))
                if (mValue == tValue) return new ASMLInteger(1);
```

```
			else return new ASMLInteger(0);
		else if (op.equals("!="))
			if (mValue != tValue) return new ASMLInteger(1);
			else return new ASMLInteger(0);
		return super.relate(rhs, op);
	}

	@Override
	public Value subtract(Value rhs) throws ASMLSemanticException {
		if(rhs.getType() != Type.TIME)
			return super.subtract(rhs);

		return new ASMLTime(mValue - ((ASMLTime)rhs).getValue());
	}

	public double getValue() throws ASMLSemanticException {
		if(this.mIsInitialized)
			return mValue;
		throw new ASMLSemanticException("Cannot return value for uninitialized
identifier");
	}
}
```

3j. ASMLWave.java

```
package asml.walker;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collection;

import javax.sound.sampled.AudioFileFormat;
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.UnsupportedAudioFileException;

import org.tritonus.dsp.ais.AmplitudeAudioInputStream;
import org.tritonus.share.sampled.FloatSampleBuffer;

import asml.walker.streams.*;

/**
 *
 * @author Tim Favorite
 *
 */
public class ASMLWave extends Value {

	/**
	 * This is the representation of the wave.
	 */
	protected AudioInputStream mValue;
	protected boolean mIsAtResult = false;
	protected ASMLTime mStartTime = new ASMLTime(-1);
	protected ASMLTime mEndTime = new ASMLTime(-1);
	protected ASMLFrequency mStartFreq = new ASMLFrequency(-1);
	protected ASMLFrequency mEndFreq = new ASMLFrequency(-1);

	/**
	 * Constructs an ASMLWave object from an existing AudioInputStream. This will be
```

```
	 * considered a literal and not a storable object.
	 * Example: input at 500Hz
	 * TODO is this going to be used ever?
	 * @param aValue the wave stream
	 */
	protected ASMLWave(AudioInputStream aValue) {
		mType = Type.WAVE;
		mValue = aValue;
		mIsInitialized = true;
	}

	/**
	 * Constructs an ASMLWave object from an existing AudioInputStream, assigns it a
	 * name.
	 * Example: wave fido
	 * @param aValue the wave stream
	 * @param aName the name of the object
	 */
	protected ASMLWave(AudioInputStream aValue, String aName){
		this(aValue);
		mName = aName;
		mIsStorable = true;
	}

	/**
	 * Constructs an ASMLWave object from a name and a boolean constant value
(declaration)
	 * Example: wave b
	 * @param aName the name of the object
	 */
	public ASMLWave(String aName){
		mType = Type.WAVE;
		mName = aName;
		mIsInitialized = false;
		mIsStorable = true;
	}

	public ASMLWave(String aName, String file) throws ASMLSemanticException {
		this(aName);
		try {
			mValue = AudioSystem.getAudioInputStream(new File(file));
		} catch (UnsupportedAudioFileException e) {
			throw new ASMLSemanticException("Unsupported file type, cannot create a
wave!");
		} catch (IOException e) {
			throw new ASMLSemanticException("Cannot read from file, please check
whether it exists.");
		}
		mIsInitialized = true;
	}

	/**
	 * Copy constructor
	 * @param fileName
	 * @return
	 * @throws ASMLSemanticException
	 * @throws ASMLSemanticException
	 */
	public ASMLWave(Value aValue) throws ASMLSemanticException{
		if(aValue.getType() != Type.WAVE)
```

```
			throw new ASMLSemanticException("Cannot set a non-wave value to a
wave.");
		mType = Type.WAVE;
		mName = aValue.getName();
		mIsStorable = aValue.isStorable();
		mIsInitialized = aValue.isInitialized();
		mValue = ((ASMLWave)aValue).getValue();
	}

	public ASMLWave(Value aValue, String aName) throws ASMLSemanticException{
		if(aValue.getType() != Type.WAVE)
			throw new ASMLSemanticException("Cannot set a non-wave value to a
wave.");
		mType = Type.WAVE;
		mName = aName;
		mIsStorable = aValue.isStorable();
		mIsInitialized = aValue.isInitialized();
		mValue = ((ASMLWave)aValue).getValue();
	}

	public static ASMLWave createWaveFromFile(String fileName, String name) throws
ASMLSemanticException {
		try {
			return new ASMLWave(AudioSystem.getAudioInputStream(new
File(fileName)), name);
		} catch (UnsupportedAudioFileException e) {
			throw new ASMLSemanticException("Unsupported file type, cannot create a
wave!");
		} catch (IOException e) {
			throw new ASMLSemanticException("Cannot read from file, please check
whether it exists.");
		}
	}

	/**
	 * Adds a value (rhs) to this wave. Rhs can either be an int, a float, or a wave.
	 * If it is an int or float it essentially acts as a vertical shift operation. If it
	 * is a wave, it will mix this wave and the rhs wave together.
	 * @param rhs the right hand side of the addition
	 * @return the result of the add operation
	 */
	public Value add(Value rhs) throws ASMLSemanticException{
		double scalar;
		switch(rhs.getType()){
			case Type.INT:
				scalar = ((ASMLInteger)rhs).getValue();
				break;
			case Type.FLOAT:
				scalar = ((ASMLFloat)rhs).getValue();
				break;
			case Type.WAVE:
				Collection<AudioInputStream> coll = new
ArrayList<AudioInputStream>();
				coll.add(mValue);
				coll.add(((ASMLWave)rhs).getValue());
				return new ASMLWave(new
MixingFloatAudioInputStream(mValue.getFormat(), coll));
			default: return super.add(rhs);
		} return new ASMLWave(new ScalarFloatAudioInputStream(mValue.getFormat(),
mValue, scalar));
```

```
    }


    /**
     * Multiplies this wave by an int or a float, changing the amplitude of the wave
     *  by that factor.
     * @param rhs the int or float value by which the wave is multiplied.
     * @return modified wave
     */
    @Override
    public Value multiply(Value rhs) throws ASMLSemanticException {
            float scalar;
            switch (rhs.getType()){
                    case Type.INT:
                            scalar = ((ASMLInteger)rhs).getValue();
                            break;
                    case Type.FLOAT:
                            scalar = (float)(((ASMLFloat)rhs).getValue());
                            break;
                    default:
                            return super.multiply(rhs);
            }
            AmplitudeAudioInputStream aais = new AmplitudeAudioInputStream(mValue);
            aais.setAmplitudeLinear(scalar);
            return new ASMLWave(aais);

    }

    /**
     * Returns the amplitude of this wave.
     * @return the amplitude
     */
    public Value amplof() throws ASMLSemanticException {
            // TODO 8192 is a totally arbitrary size for the buffer - maybe we'll try different
values to see which one gives best performance
            int samplecount = 8192;
            AmplitudeAudioInputStream aais = new AmplitudeAudioInputStream(mValue,
mValue.getFormat());
            FloatSampleBuffer buffer = new
FloatSampleBuffer(mValue.getFormat().getChannels(), samplecount,
mValue.getFormat().getSampleRate());
            float[] channel;
            float peak = 0;
            aais.read(buffer);
            while(buffer.getSampleCount() > 0){
                    for(int i=0; i<buffer.getChannelCount(); i++){
                            channel = buffer.getChannel(i);
                            for(int j=0; j<channel.length; j++){
                                    if(Math.abs(channel[j]) > peak) peak =
Math.abs(channel[j]);
                            }
                    }
                    aais.read(buffer);
            }
            return new ASMLAmplitude((double)peak);
    }

    /**
     * Constructs a wave made up of a range of samples in the time or frequency domain.
     * rhs1 and rhs2 must be of same type, and can only be time or frequency values.
```

```
	 * @param rhs1 the beginning of range
	 * @param rhs2 the end of the range
	 * @return the range of samples
	 */
	public Value at(Value rhs1, Value rhs2) throws ASMLSemanticException {
		if(rhs1.getType() == Type.TIME && rhs2.getType() == Type.TIME){
			double start = ((ASMLTime)rhs1).getValue();
			double end = ((ASMLTime)rhs2).getValue();
			float frameSize = mValue.getFormat().getFrameSize() *
mValue.getFormat().getFrameRate();
			long fstart = (long)(start/1000 * frameSize);
			long flength = (long)((end/1000 - start/1000)*frameSize);
			try {
				mValue.skip(fstart);
			} catch (IOException e) {
				throw new ASMLSemanticException("Cannot read from wave file!");
			}
			return new ASMLWave(new AudioInputStream(mValue,
mValue.getFormat(), flength), mName);
		} else if(rhs1.getType() == Type.FREQ && rhs2.getType() == Type.FREQ){
			/* Windowed-sinc implementation */
			float[] filter = windowedSinc(((ASMLFrequency)rhs1).getValue(),
((ASMLFrequency)rhs2).getValue());
			return new ASMLWave(new
ConvolvingFloatAudioInputStream(mValue.getFormat(), mValue, filter), mName);
		}
		return super.at(rhs1,rhs2);
	}

	/**
	 * Constructs a wave made up of one sample in the time or frequency domain.
	 * Rhs must be of type time or frequency.
	 * @param rhs the time or frequency specifier
	 * @return a wave with that one sample
	 */
	public Value at(Value rhs) throws ASMLSemanticException {
		if(rhs.getType() == Type.TIME){
			double time = ((ASMLTime)rhs).getValue();
			float frameSize = mValue.getFormat().getFrameSize();
			long fstart = (long)(time * frameSize);
			try {
				mValue.skip(fstart);
			} catch (IOException e) {
				throw new ASMLSemanticException("Cannot read from wave file!");
			}
			return new ASMLWave(new AudioInputStream(mValue,
mValue.getFormat(), 1));
		} else if(rhs.getType() == Type.FREQ){
			// This too, is a range, but a range of 1 - rhs to rhs + 1...
			double freq = ((ASMLFrequency)rhs).getValue();
			float[] filter = windowedSinc(freq, freq + 1.0);
			return new ASMLWave(new
ConvolvingFloatAudioInputStream(mValue.getFormat(), mValue, filter));
		}
		return super.at(rhs);
	}

	/**
	 * Multiplies wave by -1, essentially performing an inversion.
	 * @return the inverted wave
```

```
     */
    public Value negate() throws ASMLSemanticException {
            return multiply(new ASMLInteger(-1));
    }

    /**
     * Subtracts a value (rhs) from this wave. Rhs can either be an int, a float, or a wave.
     * If it is an int or float it essentially acts as a vertical shift operation. If it
     * is a wave, it will unmix the rhs wave from this wave.
     * @param rhs the right hand side of the addition
     * @return the result of the subtract operation
     */
    @Override
    public Value subtract(Value rhs) throws ASMLSemanticException {
            double scalar;
            switch(rhs.getType()){
                    case Type.INT:
                            scalar = ((ASMLInteger)rhs).getValue();
                            break;
                    case Type.FLOAT:
                            scalar = ((ASMLFloat)rhs).getValue();
                            break;
                    default: return super.add(rhs);
            }
             return new ASMLWave(new ScalarFloatAudioInputStream(mValue.getFormat(),
mValue, -scalar));
    }

    /** Gets the AudioInputStream from this wave.
     *
     * @return the AudioInputStream representing this wave
     */
    public AudioInputStream getValue()throws ASMLSemanticException {
            if(this.mIsInitialized)
                    return mValue;
            throw new ASMLSemanticException("Cannot return value for uninitialized
identifier");
    }

    /* Windowed-sinc filter generator. The filter generation would likely be more efficient
     * using Fast Fourier Transform (FFT), but as this was easier to learn for a DSP
     * novice, here's this instead.
     */
    private float[] windowedSinc(double start, double end){
            double sum;
            int windowSize = 800;
            if (windowSize % 2 == 1) windowSize += 1;

            double lower[] = new double[windowSize + 1];
            double upper[] = new double[windowSize + 1];
            double filter[] = new double[windowSize + 1];
            double fstart = start/mValue.getFormat().getSampleRate();
            double fend = end/mValue.getFormat().getSampleRate();

            //calculate lower - it's a low pass filter with cutoff of start
            for (int i=0; i<=windowSize; i++){
                    if(i == windowSize/2) lower[i] = 2 * Math.PI * fstart;
                    else lower[i] = Math.sin(2*Math.PI*fstart*(i-windowSize/2))/(i-
windowSize/2);
```

```
			lower[i] = lower[i] * (0.42-
0.5*Math.cos(2*Math.PI*i/windowSize)+0.08*Math.cos(4*Math.PI*i/windowSize));
		}

		// normalize lower
		sum = 0.0;
		for (int i=0; i<=windowSize; i++) sum +=lower[i];
		for (int i=0; i<=windowSize; i++) lower[i] = lower[i]/sum;

		// same thing for upper now, cutoff is end instead of start
		for (int i=0; i<=windowSize; i++){
			if(i == windowSize/2) upper[i] = 2 * Math.PI * fend;
			else upper[i] = Math.sin(2*Math.PI*fend*(i-windowSize/2))/(i-
windowSize/2);

			upper[i] = upper[i] * (0.42-
0.5*Math.cos(2*Math.PI*i/windowSize)+0.08*Math.cos(4*Math.PI*i/windowSize));
		}

		// normalize upper
		sum = 0.0;
		for (int i=0; i<=windowSize; i++) sum +=upper[i];
		for (int i=0; i<=windowSize; i++) upper[i] = upper[i]/sum;

		// now change upper into a high-pass filter using spectral inversion
		for (int i=0; i<=windowSize; i++) upper[i] = -upper[i];
		upper[windowSize/2] += 1.0;

		// now add lower and upper to make a band-reject filter
		for (int i=0; i<=windowSize; i++) filter[i] = lower[i] + upper[i];

		//do spectral inversion on band-reject filter to make our band-pass filter!
		for (int i=0; i<=windowSize; i++) filter[i] = -filter[i];
		filter[windowSize/2] += 1.0;

		// Convert to float array for use with ConvolvingFloatAudioInputStream
		float[] floatfilter = new float[windowSize + 1];
		for (int i=0; i<=windowSize; i++) floatfilter[i] = (float)filter[i];
		return floatfilter;
	}

	/**
	 * @return the mIsAtResult
	 */
	public boolean isAtResult() {
		return mIsAtResult;
	}

	/**
	 * @param isAtResult the mIsAtResult to set
	 */
	public void setIsAtResult(ASMLTime start, ASMLTime end) {
		mIsAtResult = true;
		mStartTime = start;
		mEndTime = end;
	}

	public void setIsAtResult(ASMLFrequency start, ASMLFrequency end) {
		mIsAtResult = true;
		mStartFreq = start;
```

```
		mEndFreq = end;
	}
	/**
	 * @return the mEndFreq
	 */
	public ASMLFrequency getEndFreq() {
		return mEndFreq;
	}

	/**
	 * @return the mEndTime
	 */
	public ASMLTime getEndTime() {
		return mEndTime;
	}

	/**
	 * @return the mStartFreq
	 */
	public ASMLFrequency getStartFreq() {
		return mStartFreq;
	}

	/**
	 * @return the mStartTime
	 */
	public ASMLTime getStartTime() {
		return mStartTime;
	}

	public void write(String file) throws ASMLSemanticException{
		try {
			AudioSystem.write(mValue, AudioFileFormat.Type.WAVE, new File(file));
		} catch (IOException e) {
			throw new ASMLSemanticException("Could not write to output file - please
check" +
					" to make sure file exists.");
		}
	}
}
```

3k. FunctionRecord.java

```
package asml.walker;

import java.util.ArrayList;
import java.util.Stack;
import org.antlr.runtime.tree.*;

/**
 * @author Frank A Smith and Tim Favorite
 * */

//TODO Remove anything to do with scope depth.  Doesn't work!
public class FunctionRecord {
	protected int mType;
	protected String mName  = null;
	protected Value mRetVal = null;

	protected Stack<SymbolTable> mSTStack    = null;
	protected ArrayList<Value> mFormalParams = null;
```

```
   protected CommonTree mBlockRt = null;
   protected SymbolTable mBottom = null;
   protected boolean mCanExecute = true;

   public FunctionRecord(int aType, String aName,
                  ArrayList<Value> aFormalArgs, CommonTree aBlockRt) throws
ASMLSemanticException{
           mType = aType;
           mName = aName;
           mFormalParams = aFormalArgs;
           mBlockRt = aBlockRt;
           mRetVal = new ASMLError(aName + " failed on call");

           mSTStack = new Stack<SymbolTable>();
           SymbolTable st = new SymbolTable();
           for(int i=0; i<aFormalArgs.size(); i++){
                  st.declare(mFormalParams.get(i));
           }
           mSTStack.push(st);
           mBottom = st;
   }

   /*
    * Because this is a copy constructor, I'll just use direct references to the copied
    * FunctionRecords protected fields.  I think this is a legitimate use of the new
    * functionality of protected Java members.
    * This still uses a lot of references, but the constants in here should never get
    * changed.  The most important features are resetting the symbol table, return
    * value and mCanExecute, which should all happen.*/
   public FunctionRecord(FunctionRecord aOrig) throws ASMLSemanticException{
           mType = aOrig.mType;
           mName = aOrig.mName;
           mFormalParams = aOrig.mFormalParams;
           mBlockRt = aOrig.mBlockRt;
           mRetVal = new ASMLError(mName + " failed on call");

           mSTStack = new Stack<SymbolTable>();
           SymbolTable st = new SymbolTable();
           for(int i=0; i<mFormalParams.size(); i++){
                  st.declare(mFormalParams.get(i));
           }
           mSTStack.push(st);
           mBottom = st;
   }

   //Argument handling methods

   public int getNumArgs(){
           return mFormalParams.size();
   }

   public void passParamValue(ArrayList<Value> aActualParams) throws
ASMLSemanticException{
           if (aActualParams.size() != mFormalParams.size())
                  throw new ASMLSemanticException("Number of arguments in call differs
from " +
                                 "number of arguments in function declaration.  Call to
function "+
                                 mName+".");
```

```
		Value tFormal, tActual = null;
		for(int i=0; i<aActualParams.size(); i++){
			tFormal = mFormalParams.get(i);
			tActual = aActualParams.get(i);
			if(!tActual.isInitialized())
				throw new ASMLSemanticException("Cannot assign uninitialized
actual argument to formal argument '"
						+tFormal.getName()+"'.");
			if(tActual.getType() == tFormal.getType()){
				String tFormalName = tFormal.getName();
				switch(tActual.getType()){
					case Type.INT:
					{
						ASMLInteger tVal = (ASMLInteger)tActual;
						mBottom.update(tFormalName, new
ASMLInteger(tVal.getValue(), tFormalName));
						break;
					}
					case Type.FLOAT:
					{
						ASMLFloat tVal = (ASMLFloat)tActual;
						mBottom.update(tFormalName, new
ASMLFloat(tVal.getValue(), tFormalName));
						break;
					}
					case Type.AMPL:
					{
						ASMLAmplitude tVal = (ASMLAmplitude)tActual;
						mBottom.update(tFormalName, new
ASMLAmplitude(tVal.getValue(), tFormalName));
						break;
					}
					case Type.FREQ:
					{
						ASMLFrequency tVal = (ASMLFrequency)tActual;
						mBottom.update(tFormalName, new
ASMLFrequency(tVal.getValue(), tFormalName));
						break;
					}
					case Type.TIME:
					{
						ASMLTime tVal = (ASMLTime)tActual;
						mBottom.update(tFormalName, new
ASMLTime(tVal.getValue(), tFormalName));
						break;
					}
					case Type.WAVE:
					{
						ASMLWave tVal = (ASMLWave)tActual;
						mBottom.update(tFormalName, new
ASMLWave(tVal.getValue(), tFormalName));
						break;
					}
				}
			}
			else
				throw new ASMLSemanticException("Type of actual argument does
not match formal argument '"+
						tFormal.getName() + "'.");
```

```
		}
	}

	public void passParamString(ArrayList<String> aActualParams) throws
ASMLSemanticException{
		if (aActualParams.size() != mFormalParams.size())
			throw new ASMLSemanticException("Number of arguments in call differs
from " +
					"number of arguments in function declaration.  Call to
function "+
					mName+".");

		Value tFormal  = null;
		String tActual = null;
		for(int i=0; i<aActualParams.size(); i++){
			tFormal = mFormalParams.get(i);
			tActual = aActualParams.get(i);

			switch(tFormal.getType()){
			case Type.AMPL:
				try {
						mBottom.update(tFormal.getName(), new
ASMLAmplitude(Double.valueOf(tActual), tFormal.getName()));
						break;
					} catch (NumberFormatException e) {
						throw new ASMLSemanticException("Type of actual
argument does not match formal argument '"+
								tFormal.getName() + "'.");
					}
			case Type.FLOAT:
				try {
					mBottom.update(tFormal.getName(), new
ASMLFloat(Double.valueOf(tActual), tFormal.getName()));
					break;
				} catch (NumberFormatException e) {
					throw new ASMLSemanticException("Type of actual argument
does not match formal argument '"+
							tFormal.getName() + "'.");
				}
			case Type.FREQ:
				try {
					mBottom.update(tFormal.getName(), new
ASMLFrequency(Double.valueOf(tActual), tFormal.getName()));
					break;
				} catch (NumberFormatException e) {
					throw new ASMLSemanticException("Type of actual argument
does not match formal argument '"+
							tFormal.getName() + "'.");
				}
			case Type.TIME:
				try {
					mBottom.update(tFormal.getName(), new
ASMLTime(Double.valueOf(tActual), tFormal.getName()));
					break;
				} catch (NumberFormatException e) {
					throw new ASMLSemanticException("Type of actual argument
does not match formal argument '"+
							tFormal.getName() + "'.");
				}
			case Type.INT:
```

```
                        try {
                                mBottom.update(tFormal.getName(), new
ASMLInteger(Integer.valueOf(tActual), tFormal.getName()));
                                break;
                        } catch (NumberFormatException e) {
                                throw new ASMLSemanticException("Type of actual argument
does not match formal argument '"+
                                                tFormal.getName() + "'.");
                        }
                case Type.WAVE:
                        ASMLWave wave = ASMLWave.createWaveFromFile(tActual,
tFormal.getName());
//                      wave.mName = tFormal.getName();
                        mBottom.update(tFormal.getName(), wave);
                        break;
                default:
                        throw new ASMLSemanticException("Unknown excecption reached
when passing parameters to function '"+
                                        mName + ".'");
                }// end switch
        }
    }

    //Value get/set

    public Value getRetVal() {
        return mRetVal;
    }

    public void setRetVal(Value aRetVal) throws ASMLSemanticException {
        if(mType != aRetVal.getType())
            throw new ASMLSemanticException("Type mismatch, function with type " +
                    mType + " cannot return type " + aRetVal.getType());
        mRetVal = aRetVal;
    }

    //Get Block root

    public CommonTree getBlockRt() {
        return mBlockRt;
    }

    //SymbolTable Handling methods

    public void enterScope(){
        SymbolTable st = new SymbolTable(mSTStack.peek());
        mSTStack.push(st);
    }

    public void exitScope(){
        mSTStack.pop();
    }

    public void addSymbol(Value aVal)throws ASMLSemanticException{
        mSTStack.peek().declare(aVal);
    }

    public void editSymbol(Value aVal)throws ASMLSemanticException{
        mSTStack.peek().update(aVal);
    }
```

```
    public Value getSymbol(String aName)throws ASMLSemanticException{
            return mSTStack.peek().retrieve(aName);
    }

    public boolean canExecute() {
            return mCanExecute;
    }

    public void setCanExecute(boolean canExecute) {
            mCanExecute = canExecute;
    }

    public String getName() {
            return mName;
    }
}
```

3l. SymbolTable.java

```
/**
 *
 */
package asml.walker;
import java.util.HashMap;

/**
 * @author Frank A. Smith & Tim Favorite
 *
 */
public class SymbolTable {
    protected SymbolTable mParent = null;
    protected HashMap<String, Value> mST         = null;

    public SymbolTable(){
            mST = new HashMap<String, Value>();
    }

    public SymbolTable(SymbolTable aParent){
            this();
            mParent = aParent;
    }

    public void declare(Value aVal)throws ASMLSemanticException{
            if(!aVal.isStorable())
                    throw new ASMLSemanticException("Cannot create entry for non-storable
value.");

            if(mST.containsKey(aVal.mName))
                    throw new ASMLSemanticException("Cannot re-declare value '"+
aVal.getName()+
                                    "' within the same scope.");
            mST.put(aVal.getName(), aVal);
    }

    public void update(String aName, Value aVal)throws ASMLSemanticException{
            if(!aVal.isStorable())
                    throw new ASMLSemanticException("Cannot create entry for non-storable
value.");
            if(!aVal.isInitialized())
                    throw new ASMLSemanticException("Cannot assign uninitialized value to '"
```

```
+ aName +"'.");

        if(mST.containsKey(aName)){
            if(mST.get(aName).getType() != aVal.getType())
                throw new ASMLSemanticException("Type mismatch- cannot assign
to '"+
                        aName + "'.");
            mST.put(aName, aVal);
        }
        else{
            if(mParent == null)
                throw new ASMLSemanticException("Undeclared value '"+ aName +
                    "'- cannot assign.");
            else
                mParent.update(aName, aVal);
        }
    }

    public void update(Value aVal)throws ASMLSemanticException{
        if(!aVal.isStorable())
            throw new ASMLSemanticException("Cannot create entry for non-storable
value.");

        this.update(aVal.getName(), aVal);
    }

    public Value retrieve(String aName)throws ASMLSemanticException{
        if(mST.containsKey(aName))
            return mST.get(aName);

        if(mParent == null)
            throw new ASMLSemanticException("Undeclared value '"+ aName +
                "'- cannot retrieve.");
        else
            return mParent.retrieve(aName);
    }
}
```

3m. Type.java

```
/**
 *
 */
package asml.walker;

/**
 * A simple way for us to keep track of values of the different types
 * in implementing the ASML walker.
 * @author Frank A. Smith & Tim Favorite
 *
 */
public final class Type {
    public static final int INT    = 0;
    public static final int FLOAT  = 1;
    public static final int AMPL   = 2;
    public static final int FREQ   = 3;
    public static final int TIME   = 4;
    public static final int WAVE   = 5;
    public static final int STRING = 6;
    public static final int ERROR = -1;
```

```
    public static int getType(String typeStr){
            if (typeStr.equals("int")) return INT;
            else if(typeStr.equals("float")) return FLOAT;
            else if(typeStr.equals("ampl")) return AMPL;
            else if(typeStr.equals("freq")) return FREQ;
            else if(typeStr.equals("time")) return TIME;
            else if(typeStr.equals("wave")) return WAVE;
            return ERROR;
    }
}
```

3n. Value.java

```
/**
 *
 */
package asml.walker;

/**
 * This is the parent class for an abstraction of every major type
 * that is used by ASML.  Value is the return type of all expressions.
 * Children of Value are expected to implement the way their individual
 * type handles various operations (e.g. addition, multiplication).
 * @author Frank A. Smith & Tim Favorite
 *
 */
public abstract class Value {
    /** The Type of this Value (see Type class) */
    protected int mType;
    /** The name of this Value object. */
    protected String mName        = null;
    /** Represents whether this Value object is storable or not (is it a variable?) */
    protected boolean mIsStorable = false;
    /** Represents whether this Value object has been initialized */
    protected boolean mIsInitialized = false;

    //public accessors
    /**
     * Gets the type.
     * @return the type (see Type class)
     */
    public int getType(){return mType;}
    /**
     * Is this object storable?
     * @return whether it's storable
     */
    public boolean isStorable(){return mIsStorable;}
    /**
     * Gets object's name.
     * @return the name of this object
     */
    public String getName() {return mName;}
    /**
     * Has this object been initialized?
     * @return if it's been initialized
     */
    public boolean isInitialized() {return mIsInitialized;}

    /**
     * Performs an add operation
     * @param rhs The right hand side of the operation
```

```
     * @return the value of the add operation
     * @throws ASMLSemanticException if operand types were incompatible with this
operation
     */
    public Value add(Value rhs) throws ASMLSemanticException{
            throw new ASMLSemanticException("Illegal Operation: add");
    }

    /**
     * Performs a subtraction operation
     * @param rhs the right hand side of the operation
     * @return the value of the subtraction operation
     * @throws ASMLSemanticException if operand types were incompatible with this
operation
     */
    public Value subtract(Value rhs) throws ASMLSemanticException{
            throw new ASMLSemanticException("Illegal Operation: subtract");
    }

    /**
     * Performs a multiplication operation
     * @param rhs the right hand side of the operation
     * @return the value of the multiplication operation
     * @throws ASMLSemanticException if operand types were incompatible with this
operation
     */
    public Value multiply(Value rhs) throws ASMLSemanticException{
            throw new ASMLSemanticException("Illegal Operation: multiply");
    }

    /**
     * Performs a division operation
     * @param rhs the right hand side of the operation
     * @return the value of the division operation
     * @throws ASMLSemanticException if operand types were incompatible with this
operation
     */
    public Value divide(Value rhs) throws ASMLSemanticException{
            throw new ASMLSemanticException("Illegal Operation: divide");
    }

    /**
     * Performs a modulus operation
     * @param rhs the right hand side of the operation
     * @return the value of the modulus operation
     * @throws ASMLSemanticException if operand types were incompatible with this
operation
     */
    public Value mod(Value rhs) throws ASMLSemanticException{
            throw new ASMLSemanticException("Illegal Operation: mod");
    }

    /**
     * Performs a relational expression
     * @param rhs the right hand side of the operation
     * @param op the operator (<, <=, >, >=, ==, !=)
     * @return the result of the relational operation - an ASMLInteger with the value 0
     * (false) or 1 (true)
     * @throws ASMLSemanticException if operand types were incompatible with this
operation
```

```
 */
public Value relate(Value rhs, String op) throws ASMLSemanticException{
        throw new ASMLSemanticException("Illegal Operation: '" + op + "'");
}

/**
 * Carries out a logical operation (logical 'and' or logical 'or')
 * with the accessed Value as the LHS of the operation, and the
 * formal parameter as the RHS of the operation
 * @param rhs a value to be tested logically
 * @param op the logical operator being used: '&&' or '||'
 * @return an ASMLInteger of 1 if the operation results in true
 * or 0 if the operation results in false*/
public Value logic(Value rhs, String op) throws ASMLSemanticException{
        throw new ASMLSemanticException("Illegal Operation: '" + op + "'");
}

public Value not() throws ASMLSemanticException{
        throw new ASMLSemanticException("Illegal Operation: '!'");
}

/**
 * Negates the value of this object
 * @return the negative of this object's value
 * @throws ASMLSemanticException if object could not be negated
 */
public Value negate() throws ASMLSemanticException{
        throw new ASMLSemanticException("Illegal Operation: 'negative'");
}

/**
 * Gets the amplitude of this object's value (note: this value must be a wave)
 * @return the amplitude of this wave
 * @throws ASMLSemanticException if the object was not a wave
 */
public Value amplof() throws ASMLSemanticException{
        throw new ASMLSemanticException("Illegal Operation: amplof");
}

/**
 * Gets a sample of this object (must be wave) at the time or frequency specified
 * @param rhs the time or frequency being specified
 * @return the sample of the wave
 * @throws ASMLSemanticException if the object was not a wave
 */
public Value at(Value rhs) throws ASMLSemanticException{
        throw new ASMLSemanticException("Illegal Operation: at");
}

/**
 * Gets a sample range of this object (must be wave) at the times or frequencies
specified
 * @param rhs the time or frequency range being specified
 * @return the sample range of the wave
 * @throws ASMLSemanticException if the object was not a wave
 */
public Value at(Value rhs1, Value rhs2) throws ASMLSemanticException{
        throw new ASMLSemanticException("Illegal Operation: at");
}
```

```
    /**
     * This method is intended to be used on the NUMBER token in the tree walker.  For this
     * reason it does not need to input check as it is impossible for a number defying
     * the rules below to be assigned to the NUMBER token.
     * @param val The string representation of a legal Integer, Float, Frequency, Amplitude,
     * or Time in the ASML language.
     * @return The value of the string cast into its appropriate ASML wrapper type.
     * */
    public static final Value valueOf(String val){
            val = val.trim();
            if(val.endsWith("Hz"))
                    return new ASMLFrequency(Double.valueOf(val.substring(0,val.length()-
2)));
            if(val.endsWith("ms"))
                    return new ASMLTime(Double.valueOf(val.substring(0,val.length()-2)));
            if(val.endsWith("a"))
                    return new ASMLAmplitude(Double.valueOf(val.substring(0,val.length()-
1)));
            if(val.contains("."))
                    return new ASMLFloat(Double.valueOf(val));

            return new ASMLInteger(Integer.valueOf(val));
    }
}
```

4a. ConvolvingFloatAudioInputStream.java

```
package asml.walker.streams;

/**
 * A class modeled on the MixingFloatAudioInputStream written by Florian Bomers
 * and Mathias Pfisterer of the Tritonus team, it takes an AudioInputStream and
 * an array of floats representing a filter and then convolves the data from the
 * stream with the filter as the user reads from it.
 * @author Tim Favorite
 */
import java.io.ByteArrayInputStream;
import java.io.IOException;

import javax.sound.sampled.AudioFormat;
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;

import org.tritonus.lowlevel.dsp.FIR;
import org.tritonus.share.sampled.FloatSampleBuffer;

public class ConvolvingFloatAudioInputStream extends AudioInputStream {

    private FIR filter;
    private FloatSampleBuffer wavebuffer;
    private AudioInputStream ais;

    /**
     * Creates a ConvolvingFloatAudioInputStream
     * @param audioFormat format of the AudioInputStream
     * @param ais the AudioInputStream
     * @param coeffs the float array which is the filter
     */
    public ConvolvingFloatAudioInputStream(AudioFormat audioFormat, AudioInputStream
ais, float[] coeffs) {
            super(new ByteArrayInputStream(new byte[0]), audioFormat,
```

```
                        AudioSystem.NOT_SPECIFIED);
            filter = new FIR(coeffs);
            wavebuffer = new FloatSampleBuffer(audioFormat.getChannels(), 0,
                        audioFormat.getSampleRate());
            this.ais = ais;
    }

    /**
     * Reads from stream
     * @param data byte array to be read into
     * @param offset where in byte array to start reading into
     * @param length how many bytes to read
     * @returns number of bytes read
     * @throws IOException if there was a problem reading from stream
     */
    public int read(byte[] data, int offset, int length) throws IOException {
            //initialize wavebuffer
            wavebuffer.changeSampleCount(length/getFormat().getFrameSize(), false);
            wavebuffer.makeSilence();

            // how many bytes are we going to need to read in? samples * size of frame (in
bytes)
            int needRead = wavebuffer.getSampleCount() * ais.getFormat().getFrameSize();
            byte[] tempbuff = new byte[needRead];

            int bytesRead = ais.read(tempbuff, 0, needRead);
            if (bytesRead == -1) return -1;

            wavebuffer.initFromByteArray(tempbuff, 0, bytesRead, ais.getFormat());
            int count = wavebuffer.getSampleCount();

            // manipulating samples will actually manipulate wavebuffer as well
            float[] samples = wavebuffer.getChannel(0);
            for (int i=0;i<samples.length;i++){
                  samples[i] = filter.process(samples[i]);
            }
            wavebuffer.convertToByteArray(0, count, data, offset, getFormat());
            return count * getFormat().getFrameSize();
    }
}
```

4b. ScalarFloatAudioInputStream.java

```
package asml.walker.streams;

import java.io.ByteArrayInputStream;
import java.io.IOException;

import javax.sound.sampled.AudioFormat;
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;

import org.tritonus.share.sampled.FloatSampleBuffer;

/**
 * Class modeled off of MixingAudioInputStream written by Tritonus team, multiplies
 * all samples in an AudioInputStream by a given scalar.
 * @author Tim Favorite
 *
 */
public class ScalarFloatAudioInputStream extends AudioInputStream {
```

```
    private FloatSampleBuffer readBuffer;

    /**
     * A buffer for byte to float conversion.
     */
    private byte[] tempBuffer;

    private AudioInputStream mAis;
    private float mScalar;

    public ScalarFloatAudioInputStream(AudioFormat format, AudioInputStream ais, double
scalar){
            super(new ByteArrayInputStream(new byte[0]), format,
                            AudioSystem.NOT_SPECIFIED);
            mAis = ais;
            mScalar = (float)scalar;
            readBuffer = new FloatSampleBuffer(format.getChannels(), 0,
                            format.getSampleRate());
    }

    public int read(byte[] abData, int nOffset, int nLength) throws IOException {
            readBuffer.changeSampleCount(nLength / getFormat().getFrameSize(), false);

            // calculate how many bytes we need to read from this stream
            int needRead = readBuffer.getSampleCount()
                            * mAis.getFormat().getFrameSize();

            tempBuffer = new byte[needRead];

            // read from the source stream
            int bytesRead = mAis.read(tempBuffer, 0, needRead);

            // now convert this buffer to float samples
            readBuffer.initFromByteArray(tempBuffer, 0, bytesRead,
                            mAis.getFormat());

            for (int channel = 0; channel < readBuffer.getChannelCount(); channel++) {
                    // get the arrays of the normalized float samples
                    float[] readSamples = readBuffer.getChannel(channel);
                    for (int sample = 0; sample < readSamples.length; sample++) {
//                          float newOne = readSamples[sample] + (float)mScalar;
//                          System.out.println(readSamples[sample] + " " + newOne);
                            readSamples[sample] += mScalar;
                    }


            }

            readBuffer.convertToByteArray(0, readBuffer.getSampleCount(), abData, nOffset,
getFormat());
            return readBuffer.getSampleCount() * getFormat().getFrameSize();
    }


}
```

4c. SubwaveFloatAudioInputStream.java

```
package asml.walker.streams;
```

```
import java.io.ByteArrayInputStream;
import java.io.IOException;

import javax.sound.sampled.AudioFormat;
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;

/**
 * Class modeled off of SequenceAudioInputStream written by Tritonus team, works
somewhat akin
 * to a "substring" function where only part of the object is modified with new values.
 * @author Tim Favorite
 *
 */
public class SubwaveFloatAudioInputStream
   extends                  AudioInputStream
{
   private AudioInputStream origStream;
   private AudioInputStream newStream;
   private int start;
   private int len;
   private int frameMark = 0;
   private AudioFormat format;
   private AudioInputStream currentStream;

   public SubwaveFloatAudioInputStream(AudioFormat audioFormat, AudioInputStream
origStream, AudioInputStream newStream,
                    double start, double end)
   {
          super(new ByteArrayInputStream(new byte[0]), audioFormat,
AudioSystem.NOT_SPECIFIED);
          this.origStream = origStream;
          this.newStream = newStream;
          this.format = audioFormat;
          this.start = (int)((format.getFrameRate())*format.getFrameSize()*(start/1000));
          this.len = (int)((format.getFrameRate())*format.getFrameSize()*(end/1000) -
this.start);
          currentStream = origStream;
   }

   public int read()
          throws IOException
   {
          if (frameMark >= start && frameMark < start + len &&
!currentStream.equals(newStream))
                 currentStream = newStream;
          else if (frameMark >= start + len && !currentStream.equals(origStream))
                 currentStream = origStream;
          int      nByte = currentStream.read();
          if (nByte == -1){
                 if (currentStream.equals(newStream))
                 {
                        currentStream = origStream;
                        frameMark = start + len;
                        return read();
                 } else return -1;
          }
          frameMark ++;
          return nByte;
```

```
    }



    public int read(byte[] abData, int nOffset, int nLength) throws  IOException
    {
            int bytesRead = 0;
            if (frameMark >= start && frameMark < start + len &&
!currentStream.equals(newStream))
                    currentStream = newStream;
            else if (frameMark >= start + len && !currentStream.equals(origStream))
                    currentStream = origStream;

            if (currentStream.equals(origStream)){
                    if (frameMark + nLength < start || frameMark >= start + len){
                            bytesRead = currentStream.read(abData, nOffset, nLength);
                            frameMark += bytesRead;
                    }
                    else if (frameMark < start){
                            bytesRead = currentStream.read(abData, nOffset, start-frameMark);
                            frameMark += bytesRead;
                            if(bytesRead == -1) return -1;
                            return bytesRead + read(abData, bytesRead, nLength-bytesRead);
                    }
            } else {
                    if (frameMark + nLength < start + len){
                            bytesRead = currentStream.read(abData, nOffset, nLength);
                            frameMark += bytesRead;
                    } else {
                            bytesRead = currentStream.read(abData, nOffset, start + len -
frameMark);
                            frameMark += bytesRead;
                            if(bytesRead == -1) return -1;
                            return bytesRead + read(abData, bytesRead, nLength-bytesRead);
                    }
            }
            return bytesRead;
    }


    public int available()
            throws IOException
    {
            return origStream.available() + newStream.available();
    }



    public void close()
            throws IOException
    {
            // TODO: should we close all streams in the list?
    }

    public boolean markSupported()
    {
            return false;
    }
}
```

```
/*** SequenceAudioInputStream.java ***/
```