ASML Language Reference Manual

Tim Favorite (tuf1) & Frank Smith (fas2114) - Team SoundHammer

Columbia University COMS W4115 - Programming Languages & Translators

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`
`const`	`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 either an integer or a float followed by the suffix Hz.

4) Amplitude constants: Represented by a float followed by the suffix a.

5) Time constants: Represented by either an integer or a float followed by the suffix ms.

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

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";`

Functions are defined by the header "`fun`", a return type, a series of 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 at 440Hz = 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 is assumed to have at least one argument, "input" (of type wave), representing the data in the input .WAV file and specified by a -i option in the command line. Further command line arguments are possible, and will be referred to by keywords arg_0, arg_1, arg_2,...,arg_n where there are n arguments other than input. Functions and variables must be declared before they are referenced, so a main function must be created below any functions that it uses. ASML does not support prototypes.

Main should always return a value of type wave, and so it should always be of return type wave. It will return the 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)
At expressions (used to retrieve information from sound waves)
Function calls
Top expressions (parentheses, variables, and constants)

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 w;
freq f;
time t;
ampl a;
f = 550Hz;
f = f + 100Hz;
t = 20000ms;
w at f = input at (f-100Hz);
a = w at f at t;
```

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 either 0 or 1 - if they do 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).
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 as well as floats and integers (when frequencies, times, or amplitudes are added and subtracted to or by floats and integers, the result will be a frequency, time, or amplitude, respectively).

The functionality when waves are added to waves, or subtracted from 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 */
```

When a wave is subtracted from another wave, it is making a new mix without that wave in it. Essentially, the level on the mixer for that channel being subtracted was just set to zero.

```
band = band - bass; /* band doesn't like the bassist anymore */
```

Waves may not be added or subtracted to values of any other type.

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 other values of like types as well as floats and integers (when frequencies, times, or amplitudes are multiplied or divided by floats and integers, the result will be a frequency, time, or amplitude, respectively).

As with addition and subtraction, wave multiplication and division works differently. When a wave is multiplied by another wave, the result is a "convolution" of the two waves, or filtering of the original wave by the second wave. Effects such as band-pass/band-reject and reverb can be applied to a wave using filters assuming a filter wave exists. For example:

```
guitar = guitar * rev_filter; /* reverb effect added to guitar */
```

A wave being divided by another wave undoes any filtering that the second wave (the filter), would have done. Using the previous example, we can remove the reverb from the guitar as follows:

```
guitar = guitar / rev_filter; /* no more reverb */
```

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 an amplitude (times, frequencies, and waves are illegal). It is the functional equivalent of multiplying the legal values by -1.

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*/
```

H. Function Calls

Function calls work as they would in a language like C or Java. Since functions are required to return a value, function calls should be used in assignment statements. Function call expressions evaluate to an identifier with a type corresponding to the function's return type, 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 declare the intended type for the identifier they specify (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 bound by the fun keyword.

ASML requires that functions return a value- a lack of a default return value will result in a compile time error. 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.

A.1 Parameters

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

A call to a function must supply arguments that match the number of parameters specified as well as their individual type, in order. The const option prevents variables from being changed by statements within the function as a part of the standard call-by-reference process.

B. Variable Declarations

```
decl:
    const TYPE expr;
      TYPE 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. For variable declarations, the const modifier means that the value assigned to the identifier at definition time cannot be altered.

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 (methods). 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 only entered 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.

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 block, 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;
for(int 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 STRING ;

The print statement is intended to be used for debugging purposes. At the moment it can only be used to print string literals to the console.

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 length = in at 0ms to length;
   in = in + in2;
   return in;
end fun

fun wave main()
   time del = arg_0;
     time length = arg_1;
   wave output = delay(input, del, length);
   return output;
end fun
```