mTunes: A Midi Music Development Language

Language Reference Manual

HeeWook Lee Sharon Price[1] Hideki Sano Huitao Sheng
{hl450, sbp2001, hs2160, hs734}@columbia.edu

[1] Group Leader

2.1 Lexical conventions

2.1.1 Comments

mTunes supports one-line comments only. All comments start with ";" and continue through the end of the line. They can be placed any where within the code.

2.1.2 Identifiers (Names)

An identifier is an alphanumeric sequence. The first character must be a lower-case letter, while the remaining characters can be upper- and lower-case letters, numbers, and the "_" symbol. Upper and lower case letters are considered different.

2.1.3 Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

```
Function
for
if
else
break
Return
Pi
Exp
```

2.1.4 Numbers

A number can be denoted as a fraction "#/#" where # is any integer. If the denominator is 0, a compilation error will occur. Numbers can also be denoted in decimal format, consisting of digits and an optional decimal point. All numbers, whether integer or decimal, will be internally represented as 32-bit IEEE floating point numbers.

2.1.5 Other Tokens

The symbolic characters supported by mTunes are:

```
{ } ( ) , ;
+ - * / % #
= += -= *= /=
>= <= > < == !=
```

```
&& || !
```

2.2 Types

The following data-types are available for use in mTunes.

Note – a musical note containing information about pitch, duration, and octave
Sequence – a series of Notes
File – a pre-existing midi file
Instrument – the type of musical instrument used to play either a Sequence or Note(s)
Function – a user-defined method for generating a Sequence or Sequences
Number – 32-bit IEEE floating point number

2.2.1 Variable Declaration

For all variables except instruments, the following format is used:

```
<type> <identifier> = <expression or value>
```

Notes and Numbers can be used without assigning them to a variable name (hard-coded by the user into the program).

There are sixteen predefined instrument variables available for use. The user cannot create any new instrument variables, but can change the value assigned to any given instrument. They are denoted "I<1-16>". To set the type of instrument:

```
I<1-16> = <instrument name>
```

There are almost two-hundred predefined instruments in mTunes, almost every instrument supported by midi. The full list is included in Appendix A.

2.2.1 File Variables

When a File is created, mTunes will attempt to open the midi file for transformations. If the file was created using mTunes (or was created using the same midi format for placement of tracks, etc), mTunes will automatically create a series of variables for the user to access. Each sequence in the song will be numbered, from 1 through n, and named <identifier><1-n>, where identifier is the variable name given to the file. <identifier>num will also be set to n, the total number of sequences that mTunes was able to extract from the file.

2.2.3 Scope

All variables created within a function or loop are local to that function or loop. If an instrument variable is set within a function, it remains set to that instrument until set otherwise. Any variable created outside of any control-flow statement is considered to have global scope.

2.3 Constants

mTunes has two predefined constants as follows:

```
Pi
```

The value of pi as defined by java.Math.

```
Exp
```

The value of e as defined by java.Math.

2.4 Expressions

2.4.1 Primary Expressions

Primary expressions consist of combinations of identifiers, constants, function calls, and any other expression surrounded by “(“ and “)”.

2.4.2 Identifiers

An identifier is an expression that always evaluates to the value bounded to it.

2.4.3 () Expression

A parenthesized expression is a primary expression which evaluates to the value of the enclosed expression. The parentheses are used to ensure that an expression evaluates to the correct value, despite precedence rules.

2.4.4 Arithmetic Expressions

Arithmetic expressions take primary expressions as operands, with the exception modulo, which can only take primary expressions that evaluate to numbers. The operators “+”, “-“, “*”, “/”, and “%” indicate addition, subtraction, multiplication, division, and modulo. Multiplication, division, and modulo have the highest precedence, then addition and subtraction.

2.4.5 Relational Expressions

Relational expressions can only be used with if statements and take primary expressions as operands. The binary relational operators “>=”, “<=”, “==”, “!=”, “>”, and “<” are used to test whether the first operand is greater than or equal to, less than or equal to, equal to, not equal to, greater than, or less than the second operand. If the expression evaluates to true, the body of the if statement is executed.

2.4.6 Logical Expressions

Logical expressions can only be used with if statements and take relational expressions as operands. The logical operators “&&”, “||”, and “!” are used to test whether the first operand and, or, not the second is true. The “!” operator can be used on just one operand to test if the operand evaluates to false. If the expression evaluates to true, the body of the if statement is executed.

2.5 Statements

Statements in mTunes must be entirely on one line. Everything on any given line, until a “;” is reached is considered to be one statement. Unless there is a comment on a line, statements do not have to terminate with any symbol. Multi-line statements are not supported. A sequence of statements will be executed sequentially, unless the flow-control statements indicate otherwise.

2.5.1 Assignment

All variable assignments take the form:

```
<variable name> = <value or expression>
```

2.5.2 Conditional Statements

Conditional statements use either the “if” keyword or the “if” and “else” keyword. “else if” is not supported. Conditional statements take the form:

```
if (<logical expression>) {
      <statements>
}
```

-or-

```
if (<logical expression>) {
      <statements>
}
else {
```

```
      <statements>
}
```

If the first logical expression evaluates to true, the statements in the "{ }" immediately following the if are executed. If the logical expression evaluates to false and there is an else clause, the statements inside the "{ }" immediately following the else are executed.

2.5.3 Iterative Statement

For statement

There is one basic type of loop in mTunes, the for statement.

```
for (<starting Number>, <ending Number>, <identifier>) {
      <statements>
}
```

Both the starting Number and ending Number will be interpreted as integers, though they can be Number variables or floating point numbers. If a non-integer number is used, the floor of that number will be used instead. The identifier is the variable that will be assigned the number of the current iteration for the life of the loop. The variable is assigned the starting number for the iteration, then incremented by one for each subsequent iteration. The last iteration occurs when the variable is equal to the ending value.

Break statement

The break statement will automatically end the inner-most for loop in which it is located. The statement simply consists of the keyword "break".

2.5.4 Function Invocation and return

Function call

Functions, both user-defined and included, are called the same way:

```
<identifier>(<parameter1, parameter2, ...>)
```

Return Statement

All user-defined functions must return a sequence, which is accomplished through the return statement.

```
Return <sequence name or series of notes>
```

2.6 Function Definition

A user-defined function must take the form:

```
Function <identifier> (<type1 parameter1, type2 parameter2, ...>) {
      <statements>
      <return statement>
}
```

The identifier is the name of the function. There can be zero or more parameters to a function, comma separated if there is more than one. The type of the parameter followed by a space must be preceding the parameter name.

2.7 Internal Functions

2.7.1 Console Output

To print to the console, the Print commands are used.

```
Print <identifiers and strings>
Println <identifiers and strings>
```

The strings must be surrounded by double quotes. The identifiers and strings must be delimited by "+". Print and Println both output to the console, but Println automatically includes a new line after the output has been printed.

2.7.2 Save

To save the music generated by a program as a midi file, the save function is used.

```
Save <path>
```

The path includes the desired file-name for the file, relative to the directory from which mTunes is run. The save command must be the last line of the program, unless the PlaySong command is also being used, in which case the save command must be the next-to-last line. If save is not on the last line (or next-to-last if PlaySong is also being used), it will be ignored by the compiler.

2.7.3 Add

To add a finalized Sequence or Notes to an instrument's queue:

```
Add I<1-16> <Sequence or Notes>
```

There can only be either one Sequence or a series of Notes used with the Add command. To add multiple Sequences, multiple Add commands must be called. When the final song is played, each instrument with Notes in it's queue plays through it's

queue simultaneously from the start of the song. Each instrument plays until it's queue is empty. If the kind of instrument playing is changed at some point in the queue

2.7.4 Insert

To have an instrument start playing from its queue at a specific time, the insert command is used:

```
Insert <measure>:<beat> I<1-16> <Sequence or Notes>
```

The time at which the inserted Sequence or Notes starts playing is based on 4/4 tempo. Therefore, the time started at is equal to (4 * measure + beat). The Insert command is equivalent to using the Add command twice – first for a (4*measure + beat - 1) length rest, then for the Sequence or Notes in the Insert command. Insert can only be used on an instrument number that does not have any Sequences or Notes already in it's queue.

2.7.5 Include

To have any existing midi start playing at a specified location in the song:

```
Include <measure>:<beat> <identifier>
```

where the location is as defined above and the identifier is the name of the variable assigned to the File.

2.7.6 PlaySong

The "PlaySong" command tells the compiler to play the generated song after compilation. This command must be the last line of the program or it is ignored.

2.7.7 Sequence Manipulation

mTunes has a number of built-in Sequence manipulation methods, which can be accessed as follows:

```
duration(<Sequence>, <Number>)
```

Alters the length of each Note in the Sequence by multiplying it by the given Number

```
transpose(<Sequence>, <Number>)
```

Shifts the pitch of each Note in the Sequence by Number pitches (a positive value shifts the pitches up, negative shifts them down)

```
reverse(<Sequence>)
```

Reverses the order of the Notes in Sequence

```
palindrome(<Sequence>)
```

Creates a palindrome by appending the Sequence with the reverse of the Sequence

```
shuffle(<Sequence>)
```

Randomly orders the Notes in the Sequence

```
fadeIn(<Sequence>)
```

Alters the volume of the Sequence such that it starts at a low volume and finishes at the normal volume

```
fadeOut(<Sequence>)
```

Alters the volume of Sequence such that it starts at the normal volume and finishes at a low volume

Like user-defined functions, all of these functions return a Sequence.

2.8 Note Definition

2.8.1 Pitch

The pitch of a note is denoted by “C”, “D”, “E”, “F”, “G”, “A”, “B”, and “R”. R is a rest, while the rest of the pitches correspond to the standard pitches on the staff. The pitch is the only part of a note that must be explicitly defined. All other parts are optional, in which case they automatically default to the current default (if applicable).

2.8.2 Octave

There are ten available octaves, numbered 1 through 10. The default is 6 for both individual notes and globally, which includes middle-C. The octave can be set globally:

```
SetOctave <1 – 10>
```

The number used in the SetOctave command is on an absolute scale. To change the octave for one note, the letter representing the pitch is proceeded by the relative change in pitch from the current global pitch. The relative change is negative to decrease the octave, positive to increase the octave. An integer variable (which may be proceeded by a “-“ to change it’s sign) may be used to denote the value of the octave.

2.8.3 Sharp and Flat

Sharp is denoted by “#” and flat by “$”. If the note is to be sharp or flat, the pitch is immediately followed by the symbol.

2.8.4 Duration

The default duration is a quarter note, though any length can be specified. The shortest supported duration is 1/32; anything smaller than 1/32 will default to 1/32. The duration follows the pitch, and the sharp or flat symbol if present. It can be an integer variable or any number as defined above.