Tonedef

Language Reference Manual

Team Members

Curtis Henkel	cah2196@columbia.edu
Chatura Atapattu	cpa2116@columbia.edu
Matt Duane	md2835@columbia.edu
Kevin Ramkishun	kr2418@columbia.edu

Submitted in fulfillment of the requirements for the Programming Languages and Translators Project
November 3rd, 2010

Contents

1. Introduction 1
2. Lexical Convention 1
2.1 Comments 1
2.2 Identifiers 1
2.3 Operators 2
2.4 Keywords 2
2.5 Punctuators 3
2.6 Literals 3
2.6.1 Integer Literals 3
2.6.2 String Literals 3
2.6.3 Pitch Literals 3
3. Data Types 4
3.1 Integers 4
3.2 Boolean 4
3.3 String 4
3.4 Beat 5
3.5 Pitch 5
3.6 Note 5
3.7 Sequence 6
3.8 Chord 6
3.9 Rhythm 6
3.10 Phrase 7
3.11 Void 7
3.12 Type Conversion 7
4. Declarations 8
4.1 Declaration Syntax 8
4.2 Blocks 8
4.3 Scope 8
4.4 Identifier Naming 9
5. Functions 10

5.1 Function Declaration/Definition 10
5.2 Function Calling 10
5.5 Main 11
5.5 Play 11
5.6 Print 11
6. Operators and Expressions 12
6.1 Expressions 12
6.1.1 Identifiers and Literals 12
6.1.2 Operators and Function Calls 12
6.1.3 Parentheses 12
6.2 Arithmetic and Boolean Operators 13
6.2.1 Comparison Operators (== , != , < , <= , > , >=) 13
6.2.2 Multiplicative Operators (* , / , % , //) 13
6.2.3 Additive Operators (+ , -) 14
6.2.4 Boolean Logic Operators (&& , || , !) 14
6.3 Musical Operators 14
6.3.1 Raise Note/Pitch by Steps (^) 14
6.3.2 Raise Note/Pitch by Octaves (^^) 15
6.3.3 Difference between Pitches (-) 15
6.3.4 Note Creation from Pitch and Beat (:) 15
6.3.5 Chord Creation from Note and Sequence (::) 15
6.3.6 Addition of Notes/Chords (+) 15
6.3.7 Sequence Application (<<) 16
6.3.8 Rhythm Application (<<) 16
6.3.9 Shift Phrase Right (>>) 16
6.3.10 Combine Phrases (**) 17
6.3.11 Append Phrases (@@) 17
6.4 Assignment 17
6.5 Operator Precedence 18
7. Statements 19
7.1 Compound Statements 19
7.2 Expression Statements 19

7.3 If statement 19
7.4 While statement 19
7.5 For statement 20
7.6 Foreach statement 21
7.7 Return statement 21

1. Introduction

Tonedef is an imperative programming language designed to represent and manipulate the components of musical score. Its basic data types are chosen from the lexicon of music and its operators chosen to provide basic transformations on these types. Tonedef aims to provide a platform for constructing programs using abstractions that are familiar and useful to developers with at least a basic knowledge of music theory.

The following manual is intended for programmers of the Tonedef language. It explains the available components of the language and how they can be combined to build a Tonedef program.

2. Lexical Convention

Tonedef parses characters from source files into five types of tokens: identifiers, operators, keywords, punctuators and literals. Blanks, tabs, newlines, and comments (collectively, "Whitespace character(s)") are generally ignored except insofar as they are used to separate tokens. At least one Whitespace character is required to separate identifiers or literals. Where applicable, appropriate usage of Whitespace character(s) shall be included in the definition for the given lexical token.

2.1 Comments

Comments begin with the first character sequence /* and end with the first character sequence */ that is encountered.

Example:

```
/* This
is a
comment */
int i = 1 /* So is this.  */
```

2.2 Identifiers

An identifier is a sequence of alphanumeric characters, with the underscore character "_" included in the alphabet. Identifiers must begin with an alphabetic character, and may be followed by an optional series of one or more alphanumeric characters. Identifiers are case-sensitive, thereby making identifiers with different cases distinctive. Keywords may be incorporated into identifiers, but may not be used alone as identifiers.

There are three special identifiers: main, print, and play. They are all function names. Print and play are system-defined functions. Main is the name of the function that is the entry point for a Tonedef program and needs to be defined in the program.

Examples:

```
good_idt = 1 /* Acceptable identifier */
Good_idt = 2 /* Acceptable identifier, and different than good_idt */

_good_idt = 3 /* "_" is an acceptable starting character for an identifier */
4good_idt = 4 /* Not acceptable identifier because it starts with a number */

bool = 1 /* Not an acceptable identifier, because boolean is a keyword */
my_bool = 2 /* Acceptable identifier even with keyword incorporated */
```

2.3 Operators

Tonedef has a closed set of one and two character operator tokens.

The following are the one-character operators:

```
+ - / * % = < > : ^
```

The following are the two character operators:

```
// ** ^^ :: @@ >> << == != <= >= || &&
```

2.4 Keywords

There are twenty-one (21) reserved keywords in Tonedef that have semantic meaning in a program and may not be used as identifiers. They are,

Type names:

```
int bool string beat pitch note sequence chord phrase rhythm void
function
```

Control words:

```
if else while for foreach in return
```

Constant values:

```
true false
```

2.5 Punctuators

Some characters in Tonedef are not operators but have syntactical significance within an expression. None of these characters are allowed in identifiers.

Punctuator	Use
;	Ends a statement
[ ]	Begins and ends a sequence
( )	Expression grouping and function parameter/argument list grouping
,	List separator
{ }	Groups statements into a block
" "	Groups characters into a string literal
$	Begins a pitch literal

2.6 Literals

In Tonedef, certain sequences of characters are tokenized as literal values of various types.

2.6.1 Integer Literals

An integer literal is any continuous sequence of one or more digits [0-9] that is not part of an identifier.

2.6.2 String Literals

A string literal is comprised of all the characters between quotation marks ". A string literal is opened when a " character is found and continues until the next " is met, with the exception that the character sequence \" has special meaning within the string literal and does not close the string.

2.6.3 Pitch Literals

A pitch literal is a sequence of characters that begins with $ immediately followed by one letter from the set { A, B, C, D, E, F, G}, followed by an optional flat or sharp character from the set { #, b }, followed by a one digit from the set {0 - 9}. There is one special pitch literal that does not meet these rules and is the two-character sequence $_.

3. Data Types

Tonedef has 11 data types,

```
int bool string beat pitch note sequence chord rhythm phrase void
```

3.1 Integers

`int` - Integers are composed of a sequence of one or more digits to represent a whole number. The digits of an Integer may not be separated by Whitespace, and can be negated by placing the unary negation operator "-" before the number. The range of integers is -230 to 230-1 (or -262 to 262-1 for 64-bit systems).

3.2 Boolean

`bool` – A boolean has two possible values, true or false, which correlate to their respective logical values. Booleans may be cast to an Integer value, with `true` returning a value of 1 and `false` returning a value of 0.

3.3 String

`string` – A string is a sequence of 1 or more ASCII characters contained within quotation marks ("..."). Strings may extend across multiple lines of code, but non-explicit Whitespace characters beyond blanks (i.e. tabs and newlines) will not be included in the string. Non-printable characters may be represented in the String by using the following escape sequences.

Sequence	Description
\"	Double quotation mark
\n	Newline
\t	Tab
\r	Carriage return
\\	Backslash

3.4 Beat

`beat` – A beat value represents the duration of a note, where the beat value 1 represents a whole note. More generally, it is a rational number (i.e. can be expressed by n/d where n, d are integers). Beats can be added and subtracted to produce new beat values. There is a special beat-divide operator (//) that produces a beat representing the ratio of the two values of the operands with the left operand being the numerator and the right operand as the denominator.

Examples:

```
beat a = 1 ; /* a is a whole-note beat */
beat b = 1//4 ; /* b is a quarter-note beat */
beat c = b + 1//2; /* c is a dotted half-note beat */
beat d = 3 ; /* d is a 3 tied whole-notes beat */
```

3.5 Pitch

`pitch` – A pitch value represents the musical pitch of a note. Pitch values are comprised of exactly one letter from { A, B, C, D, E, F, G}, an optional flat or sharp { #, b } and a one-digit octave number {0 - 9}. Progression of pitches in an octave follow this sequence { C, C#, D, D#, E, F, F#, G, G#, A, A#, B }, where each pitch is a half step above the previous, and this sequence is equal to this sequence of alternate pitch names { B#, Db, D, Eb, Fb, F, Gb, G, Ab, Bb, Cb }. Note that D, F and G do not have an alternate pitch name. This sequence repeats for all the octaves so $B5 is a half step lower than $C6. The octaves are numbered from lowest to highest with C4 being middle C pitch of the traditional music staff. Pitch literals in code must begin with the dollar sign ($) character. Additionally, there is a special null pitch ($_) that does not fall in the ordered progression of pitches and represents no pitch. Pitches form an ordered set can be compared by the comparison operators (==, !=, <, >, <=, >=) with the null pitch being the less than all the lettered pitches.

Examples:

```
pitch pX = $C0; /* pX is the C of octave 0 */
pitch pY = $F#5; /* pY is the F sharp of octave 5 */
pitch pZ = $_; /* pZ is a null pitch */
```

3.6 Note

`note` – A note value is an abstraction for musical notes. They are composed of a pitch and a beat. Notes can be constructed from a just a pitch or from a pitch and a beat with the semi-colon operator (:) between them.

Examples:

```
note a = $C0; /* a has a pitch of $C0 and a duration of zero */
note b = $D1 : 1/4; /* b is a quarter-note at pitch $D1 */
note c = pX : bY; /* Notes can be constructed from variables */
```

3.7 Sequence

`sequence` - A sequence is a list of integers. They are denoted in code by a comma-separated list of integer values or expressions inside brackets [ ]. Sequences are mainly used to represent changes in pitch through time, or the distance between notes in a chord, where the integer values represent the number of half-step changes in pitch. These distinctions occur when a sequence is used as the right operand of either the (::) or (<<) operators.

Examples:

```
sequence a = [ 0, 2, 4, 5, 7, 9, 11, 12 ];
sequence b = [ 0, 4, 7 ] ;
chord majorC = ($C4:1) :: b ; /* a whole-note C-major chord specifically,
                                 the notes $C4, $E4, $G4 */
phrase majorscaleF = ($F3:1//4) << a; /* an ascending F-major scale*/
```

3.8 Chord

`chord` - A chord is a set of zero or more notes that occur simultaneously. A chord can be built empty, built from a single note, or from multiple notes either by adding (+) notes or by the (::) operator.

Examples:

```
chord a; /* empty chord - no notes */
chord b = $G5:1 ; /* chord containing a single note */
chord c = note_x + note_y + note_z ; /* chord containing 3 notes */
chord d = note_x :: [ 0, 7, 12 ] ; /* chord containing 3 notes relative
                                      to note_x */
```

3.9 Rhythm

`rhythm` - A rhythm represents a series of beats and rests. They are constructed from strings containing only the characters {1, 0, -, \s}. The sequence of these characters describes the rhythm where

- 1 signifies the initial playing a note or chord
- signifies the sustaining of a note or chord
- 0 signifies a rest
- \s separates the other characters into groups

Each group represents a total duration of a whole note so the size of the groups determine the time of each character in that group. The group sizes should be 1, 2, 4, 8, 16, which correspond to each character having length of a whole-note, a half-note, a quarter-note, an eighth-note and a sixteenth-note, respectively. If a group of characters is not equal to any of these values, the first n characters that form a complete group are used and the rest ignored.

Examples:

```
rhythm a = “1 11 1111” ; /* one whole-note, followed by 2 half-notes,
                            followed by 4 quarter notes */
rhythm b = “1--- 1-1- 1111”; /* b is equal to a but written differently
```

```
                                    as a string */
rhythm c = “10101010 1111000011110000”; /* c is (an eighth note then eighth
                                           rest) x 4, then (4 sixteenth notes,
                                           4 sixteenth rests) x 2 */
```

3.10 Phrase

`phrase` - A phrase is an ordered collection of chords, where the ordering represents time. A phrase can be built from a single chord, or from other phrases using the append phrases (@@) or combine phrases (**) operators, or using the apply operator (<<). The musical sequence represented by a phrase value can be played by sending the phrase to the play function.

Examples:

```
phrase a; /* empty phrase */
chord x = ($D3:1) :: [ 0, 4, 7 ];
phrase b = x; /* phrase that contains just chord x */
phrase c = b @@ b ; /* c is chord x played twice */
phrase d = $E5 << [ 0, 2, 2, 4 ] << “11 0 11” ; /* phrase built using the
                                                   apply operator using a
                                                   sequence and a rhythm */
```

3.11 Void

`void` - Void is a special type for functions that return no value. Non-function identifiers cannot have type void.

3.12 Type Conversion

Tonedef will perform certain conversions of types when values do not exactly meet the expected type of a function argument or operand. The following table shows the allowed conversions.

Type Conversion	Description
bool -> int	true -> 1 and false -> 0
int -> bool	0 -> false and everything else becomes a true value
beat -> int	the value of beat is rounded down to an integer
int -> beat	beats are rational numbers so the value of the int is preserved as a beat
pitch -> note	pitch is promoted to a note with duration = 0
beat -> note	beat is promoted to a note with that duration and pitch = $_
note -> pitch	note is demoted to only its pitch value
note -> beat	note is demoted to only its beat value
note -> chord	note is promoted to a chord contain just that one note
chord -> phrase	chord is promoted to a phrase contain just that one chord

4. Declarations

Declarations are used to create new variables within a block of code of specified type and optionally initialize their values.

4.1 Declaration Syntax

A declaration is an expression in one of the two following forms:

```
type identifier
type identifier = initialization-expression
```

Where

- `type` is one of the type keywords: `int`, `bool`, `string`, `beat`, `pitch`, `note`, `chord`, `sequence`, `phrase` or `rhythm`.
- `identifier` is a non-reserved alpha-numeric sequence as described in section 2.2
- `initialization-expression` is any legal Tonedef code that returns a value of agreeable type with the declaration.

4.2 Blocks

A block is a section of code enclosed by braces { }. Blocks can be nested within other blocks. Identifiers visible in an outer block are visible in the inner block, but identifiers declared in the inner block will not be visible in the outer block when the inner block ends.

Example:

```
void function f ()
{ /* start of a block for this function */
      int x = 0;
      while ( x < 10 )
      { /* start of a sub-block */

      } /* end of the sub-block */
} /* end of the block for function f */
```

4.3 Scope

The scope of an identifier is the subsequent statements within the block of code where it is declared including sub-blocks of that block. Declarations can appear after certain keywords that open a block of code. These keywords are function, for, and foreach. When identifiers are declared in these expressions, the scope of the identifiers is the block opened by the keyword. Scope does not extend to the execution of function calls. At the beginning of a function's execution, its parameters will be the only identifiers in scope.

Example:

```
void function f ( phrase p ) {
      /* p is in scope */
      foreach ( chord c in x) {
            /* c and p are in scope */
            note n = some_other_function();
            /* c, p, and n are not in scope while
            some_other_function executes */
      }
      /* c and n are no longer in scope, but p still is */
      for ( int i = 0; i < 0; i = i + 1){
            /* p and i are in scope */
      }
      /* p is only identifier in scope */
}
```

4.4 Identifier Naming

All identifiers within a block of code must be unique and a sub-block's identifiers must not conflict with the identifier names in its parent block. This means that an identifier is visible over its entire scope and cannot be hidden by a subsequent re-declaration of the identifier.

Example:

```
void function f ( note x ) {
      chord x; /* this is NOT legal because x is already an identifier
                  in this block */
      for ( int i = 0; i < 5 ; i = i + 1 ){ }
      for ( int i = 5; i > 0 ; i = i -1 ){ }
      /* this re-use of identifier i is legal because the first i is no
       longer in scope when the second is declared */
}
```

5. Functions

A Tonedef program consists of a collection of functions. Functions are re-usable segments of code that can be invoked from within the code of other functions. This provides a way to break up a program into smaller tasks. Each function consists of a name, a list of parameters, a return type and a block of execution code.

5.1 Function Declaration/Definition

A Tonedef function is declared like this:

```
type function identifier ( parameter-list ) { code }
```

Where

- `type` is the type of the return value of function
- `identifier` is the name of the function used to call it from other code
- `parameter-list` is a possibly empty and comma-separated list of parameter declarations where each parameter declaration is of the form `type identifier`
    - No two parameter names are the same within one function, and no parameter name is the same as its function's name
- `code` is a possible empty sequence of Tonedef code to be executed when the function is called

Functions cannot be declared within other functions. They must all be declared at the program level. Function names are visible throughout the program where they are declared. This means that a function can be called from any other function's execution code and that the ordering of the function declarations within a program is inconsequential.

5.2 Function Calling

A function call is an expression of the following form:

```
function-identifier ( arguments-list )
```

Where

- `function-identifier` is a name corresponding to some user-defined or system-defined function
- `arguments-list` is a possibly empty and comma-separated list of arguments to pass that function
- each argument is an expression that resolves to a value (i.e. not void).
- the number, order, and types of the argument values match the parameter types declared in the function declaration

All argument expressions are resolved before the function call is executed. An argument expression can itself be a function call so function calls within the same statement will be executed inside out.

Example:

```
fn_a(fn_b(5)); /* fn_b (5) is executed and its return value is passed as the
                  argument to the execution of fn_a */
```

5.5 Main

Each Tonedef program must include a definition of a function named `main` with type `int` and no parameters. This function is the entry point for execution of the program.

5.5 Play

The function `play` is a system-defined function of type void that takes one parameter of type `phrase`. When called, this function produces audio output from the musical expression represented by the phrase argument passed to it.

5.6 Print

The function `print` is a system-defined function of type void that takes one parameter of type `string`. When called, this function writes its argument string to the output console.

6. Operators and Expressions

Tonedef operators are any of a closed set of one and two character sequence described in section 2.3 - Lexical Convention - Operators. Some operators are unary and take an operand on the right side. Others are binary and take an operand on both the left and right side. The following sections specify the types of operands the various operators take and the types they return.

Tonedef expressions are sequences of literals, identifiers, punctuators and operators, which evaluate to a value of a Tonedef type. The allowed sequencing of these elements is explained in the following section.

6.1 Expressions

6.1.1 Identifiers and Literals

An expression can be any literal or non-function identifier. These expressions evaluate to the value of the literal or the value bound to the identifier. This defines the following syntax rule:

```
expression :=
      literal
      identifier
```

6.1.2 Operators and Function Calls

Any operator or function identifier along with the appropriate operand or argument expressions combines to be an expression. These expressions evaluate to the result of the operation/function. This adds another syntax rule:

```
expression :=
<unop> expression
      expression <binop> expression
function_identifer ( expression list )
```

6.1.3 Parentheses

An expression within parenthesis evaluates to the same value and type as the expression without parentheses. Parentheses can be used to change the precedence of operators within an expression.

```
expression :=
      (expression)
```

6.2 Arithmetic and Boolean Operators

6.2.1 Comparison Operators (== , != , < , <= , > , >=)

The types of the two operands for comparison operators must be the same. Only operands of type int, beat, pitch and note are allowed.

6.2.1(a) `expr1 == expr2`
Returns the Boolean value `true` if expr1 has the same value as expr2, and `false` if the values are different.

6.2.1(a) `expr1 != expr2`
Returns the Boolean value `false` if expr1 has the same value as expr2, and `true` if the values are different.

6.2.1(c) `expr1 < expr2`
Returns the Boolean value `true` if the value of expr1 is less than the value of expr2, and `false` if otherwise.

6.2.1(d) `expr1 <= expr2`
Returns the Boolean value `true` if the value of expr1 is less than or equal to the value of expr2, and `false` if otherwise.

6.2.1(e) `expr1 > expr2`
Returns the Boolean value `true` if the value of expr1 is greater than the value of expr2, and `false` if otherwise.

6.2.1(f) `expr1 >= expr2`
Returns the Boolean value `true` if the value of expr1 is greater than or equal to the value of expr2, and `false` if otherwise.

6.2.2 Multiplicative Operators (* , / , % , //)

These operators take operands of type int.

6.2.2(a) `expr1 * expr2`
Multiplies expr1 with expr1 and return the result. Both expressions must be of the same type.

6.2.2(b) `expr1` / expr2
Divides expr1 by expr2 and returns the integer result (rounding towards zero). Will return an error if expr2 is a null or zero value.

6.2.2(c) `expr1 % expr2`
Yields the remainder of expr1 divided by expr2. Will return an error if expr2 is a null or zero value. Both expressions must be of the same type.

6.2.2(d) `expr1 // expr2`

Divides expr1 by expr2 and returns the beat result. Will return an error if expr2 is zero valued. Both expressions must be of type `int` or `beat`.

6.2.3 Additive Operators (+ , -)

These operators can take operands of type int or beat as described here. The operators are overloaded for other types of operands, but the descriptions of those are in the section on Musical operators.

6.2.3(a) `expr1 + expr2`
Add expr1 to expr2 and return the result. Both expressions must be of the same type.

6.2.3(b) `expr1 - expr2`
Subtract expr2 from expr1 and return the result. Both expressions must be of the same type.

6.2.3(b) `-expr`
The result is the negative of expression, and has the same type.

6.2.4 Boolean Logic Operators (&& , || , !)

These operators take operands of type `bool`.

6.2.4(a) `expr1 && expr2`
Logical AND on two boolean expressions. Returns `true` only if both expressions are true.

6.2.4(b) `expr1 || expr2`
Logical OR on two boolean expressions. Returns `true` if at least one of the expressions is true, and `false` only if they are both false.

6.2.4(a) `!expr2`
Logical NEGATION. Returns `true` if the expression is false, and `false` if the expression is true.

6.3 Musical Operators

6.3.1 Raise Note/Pitch by Steps (^)

```
np ^ x
```

Returns a new note or pitch that is a copy of the left operand(np) that is raised (or lowered for negative int values) by x half-steps. The left operand can be either of type `note` or `pitch`, and the right operand is of type `int`.

6.3.2 Raise Note/Pitch by Octaves (^^)

```
np ^^ x
```

Returns a new note or pitch that is a copy of the left operand(np) with its pitch raised (or lowered for negative int values) by x octaves. The left operand can be either of type `note` or `pitch`, and the right operand is of type `int`.

6.3.3 Difference between Pitches (-)

```
pitch1 - pitch2
```

Returns an integer equal to the distance from the left operand (pitch1) to the right operand (pitch2).

6.3.4 Note Creation from Pitch and Beat (:)

```
pitch : beat
```

Returns a new note that has a pitch equal to the left operand(pitch) and beat equal to the right operand (beat).

6.3.5 Chord Creation from Note and Sequence (::)

```
note :: sequence
```

Returns a new chord that is built relative to the left operand(note) according to the integers in the right operand(sequence). Each integer in the sequence is a number of steps to raise/lower the pitch of note before adding to the resultant chord. The note itself is only added to the chord if 0 is one of the integers of in the sequence. The order of integers in the sequence does not matter for this operator because all integers are interpreted relative to the pitch of the note.

Examples:

```
chord a = ($C4:1) :: [ 0, 4, 7 ] ;  /* a has the notes $C4, $E4, and $G4 with
                                        whole-note duration */
chord b = ($E5:1) :: [ 1, -2, 7 ] ; /* b has notes $F5, $D5, $B5 */
```

6.3.6 Addition of Notes/Chords (+)

```
nc1 + nc2
```

Returns a new chord that contains the notes of the two operands (nc1, nc2) combined. Each operand can be of type note or chord.

6.3.7 Sequence Application (<<)

```
note << sequence
```

Returns a phrase that is built by applying the right operand (sequence) as a series of pitch increases/decreases relative to the pitch of the left operand (note). To include the note in the phrase, the sequence should contain a 0. The phrase will have n notes where n is the number of integers in sequence, and each note in the phrase will have the same duration as the beat of the left operand.

Examples:

```
phrase a = ($C4:1//4) << [ 0 , 4, 7, 12 ]; /* a has the notes $C4, $E4, $G4,
                                $C5 played as quarter-notes in succession */
```

6.3.8 Rhythm Application (<<)

```
phrase << rhythm
```

Returns a new phrase that has the same chords and order of chords as the left operand (phrase) but with modified beats and locations in time according to the right operand (rhythm). The ones in the string representation of rhythms are the locations in time that the non-empty chords of phrase are put at in the new phrase. If the phrase has more chords than the number of ones in rhythm, then those extra chords are omitted from the resultant phrase. If the phrase has fewer chords than the number of ones in rhythm, then those extra time locations are filled with rests.

Examples:

```
phrase a = ($C4:1//4) << [ 0 , 4, 7, 12 ]; /*same as above example */
phrase b = a << “1 11 1”; /* the four notes of a are in b, but with new
                              durations - whole, half, half, whole */
phrase c = ($D3:1) << [0,1,2,3,4,5,6,7,8,9,10,11,12] << “1110111011101110 1”;
/* c is a chromatic scale starting at $D3 played in 16th notes with 16th
rests after every third note, and ending on a whole note at $D4 */
```

6.3.9 Shift Phrase Right (>>)

```
phrase >> beat
```

Returns a new phrase that is equal to the left operand (phrase) with x beats of rest added to the front, where x is the right operand (beat).

Examples:

```
phrase d = c >> 1//2 ; /* d is the chromatic scale from the above example
                          with a half note of rest at the begging */
```

6.3.10 Combine Phrases (**)

```
phrase1 ** phrase2
```

Returns a new phrase that is the two operands (phrase1, phrase2) merged together. The start of each phrase is aligned and chords of the resultant phrase are created from the chords of phrase1 and phrase2 at each moment in time in the phrases. This operator is commutative so the order of the 2 operands does not change the result. If one operand phrase is longer than the other is, then the shorter one is interpreted to have beats of rest its notes during this merging process. The resultant phrase has total duration equal to the maximum of the durations of the two operands.

6.3.11 Append Phrases (@@)

```
phrase1 @@ phrase2
```

Returns a new phrase that is the left operand (phrase1) followed by the right operand (phrase2). There is no overlap of notes from each operand. This operator is not commutative because the order of the operands determines the ordering of their notes in the resultant phrase. This operator is left associative. The resultant phrase has total duration equal to the sum of the durations of the two operands.

6.4 Assignment

```
identifier = expression
```

Tonedef has one assignment operator (=), which takes an identifier as the left operand and an expression as the right operand. This operator evaluates the `expression` to a value and binds that value to the name `identifier`. The type of the value of `expression` must match the type of `identifier` or be a type that can be converted to the type of `identifier`. This operator also returns the value of expression and is right associative. Assignment is done by value so two identifiers can be equal to the same value, but not bound together such that re-assigning one changes the other.

Examples:

```
pitch a = $C4;     /* a is $C4 */
pitch b = a;       /* b is $C4 as well */
a = b ^ 4 ;        /* a is now $E4, and b remains $C4 */
pitch c = b = a;   /* a, b, c are all now $E4 */
```

6.5 Operator Precedence

The following table shows the order of precedence of Tonedef operators along with the associativity (order of evaluation) of each level. The top of the table is the highest precedence.

Operators	Associativity
-(unary) !	right to left
* / % //	left to right
^ ^^	left to right
:	left to right
:: << >>	left to right
**	left to right
@@	left to right
+ -(binary)	left to right
< <= > >=	left to right
== !=	left to right
&&	left to right
||	left to right
=	right to left
,	left to right

7. Statements

This section describes the different types of Tonedef statements. Every function definition is a sequence of statements. Statements are executed in the order they appear in a function body.

7.1 Compound Statements

Statements are grouped together into a block using braces { }. This allows a sequence of statements to be treated as a single statement. All previously visible identifiers remain visible within the block (see section 4.2 - Blocks). Such a grouping creates a new level of scope where new variable declarations inside the block are only in scope within that block (see section 5.3 - Scope). Compound statements are useful following any of the control structures listed in this section.

7.2 Expression Statements

Any valid expression can be used as a statement by following the expression with a semicolon(;).

```
expression ;
```

Example:

```
i = i * 2 ;
```

7.3 If statement

The if statement is a control structure for conditional execution of code. An if statement has the following syntax:

```
if ( expression ) statement1 else statement2
if ( expression ) statement1
```

When the `expression` is true (i.e. evaluates to a boolean value of `true` or non-zero int value), `statement1` is executed; otherwise, `statement2` is executed. The `else statement2` portion of the if-statement is optional, in which case nothing is executed if the `expression` is false.

7.4 While statement

The while statement is a control structure for looping execution of code as long as a control expression is true. A while statement has the following syntax:

```
while ( expression ) statement
```

The expression is evaluated before the potential execution of the statement. If the expression is true, the statement is executed, then this two-step process repeats. If the expression is false, the statement is not executed and the while loop terminates. The statement should perform some computation that

eventually causes the expression to be false and terminate the loop. Otherwise, the loop will infinitely repeat.

```
note x = $C4 ;
while ( x < $C5 ) {
      x = x^1;
}
```

In this example, x is a note and is raised one-step. Since $C5 is a higher pitch than $C4, the loop eventually terminates. If the loop body were x = x^-1, then x would be decreasing in pitch and the loop would not terminate.

7.5 For statement

The `for` statement is a control structure for iterative execution of code. A for statement has the following syntax:

```
      for ( expression1 ; expression2 ; expression3 )  statement
```

The `expression1` is executed as a statement once at the beginning of execution of the for statement. Then, `expression2` is evaluated. If it is true, statement is executed, then `expression3` is executed as a statement, then this process repeats. If `expression2` is false, the loop terminates.

A for statement is equivalent to the following while statement:

```
expression1;
while (expression2) {
      statement
      expression3;
}
```

Like while statements, a for statement needs a control expression (`expression2`) to eventually be false to terminate. This is normally done in expression3, but can be done in statement as well.

```
beat b = 0;
for ( int i = 1 ; i < 10 ; i = i + 1) {
      b = b + i // 4 ;
}
```

In this example, the body of the loop does not modify i, but `expression3` does, so `expression2` is eventually false. Specifically, this body is executed 9 times and b equals the length 45 quarter-notes.

7.6 Foreach statement

The `foreach` statement is a control structure for iterating through the elements of one of Tonedef's ordered data types (sequence, phrase). A foreach statement has the following syntax:

```
foreach ( type identifier in expression ) statement
```

Where

- `type` can be only `chord` or `int`
- when `type` is `chord`, expression must evaluate to a phrase value
- when `type` is `int`, expression must evaluate to a sequence value

The foreach statement begins with evaluating expression once to a phrase or sequence value. Each iteration of the loop begins with `identifier` becoming the next element in this value. Then, the statement is executed.

```
/* phrase p defined previously */
phrase z;
foreach ( chord c in p ) {
      z = c @@ z;
}
```

In this example, the foreach each loop executes once for each chord in the phrase and builds a new phrase z that is the reverse of p.

7.7 Return statement

The `return` statement terminates execution of a function and has the following syntax:

```
return expression ;
return ;
```

The expression is evaluated to a value. This value needs to have the same type as the return type of the function (or be a type that can be promoted/demoted to the return type). At this point, the execution of the function ends and this value becomes the return value of the function call. Program execution continues from the location of the function call.

A function may contain multiple return statements within its body and/or sub-blocks in its body. However, a function (except for void typed functions) must contain at least one return statement at the outer most block of its body. A function of type void should use the expression-less return statement syntax, or use no return statements, in which case it will return when it reaches the end of its body.

```
pitch function f ( sequence s , pitch p) {
      foreach ( int i in s ) {
            if ( i == 0 ) { return p ; }
            else { p = p ^ i ; }
      }
      return p;
}
```

In this example, there is a return statement in a conditional statement in a looping statement. This allows the function to terminate in the middle of the loop; however, the second return statement is needed because there is no guarantee that the if condition is ever true or even that the sequence has any elements to iterate through.