Sick Beets

Language Reference Manual

Manager: Courtney Wong (cw2844)
Language Guru: Angel Yang (cy2389)
System Architect: Kevin Shen (ks3206)
Tester: Jin Peng (jjp2172)

Table of Contents

1. Introduction

Sick Beets is a programming language that allows users to compose music by generating .midi files. Sick Beets is inspired by the structured nature of music, which makes it easy to represent a composition piece by defining attributes of notes such as pitch or duration. Using Sick Beets, users can concatenate, overlay, and transpose a series of notes to digitally encode their compositions.

2. Types and Literals

2.1 Primitive Types

Boolean (bool) : May be true or false.

Integer (int) : A literal such as 15 is a 64 bit signed integer.

Floating Point (float) : A floating point literal is a number with a decimal point such as 2.75 or an exponent part such as 1e 5, or both.

String (string) : A sequence of ASCII characters such as "hello world \n!" The string literal is enclosed in quotes, and special characters are escaped using a backslash. The supported escape sequences are:

\n	newline	\r	carriage return
\t	horizontal tab	\v	vertical tab
\\	backslash	\"	double quote

Note: A pitch is specified by the following form:

Accidentals are optional and can be specified by: 'b' for flats and '#' for sharps. For example, **eb(-1)** is an E-flat which is one octave below middle C. Notes without an octave offset are assumed to be in the octave of middle C. Supported notes include a,b,c,d,e,f,g and r for rests.

Duration: The duration of a note is specified by a combination of key letters. We support the following durations:

w:	whole	i:	eighth
h:	half	t:	thirty-second
q:	quarter		

Letters can be combined together to create other durations: For example, **wh** would be a dotted whole duration.

Instrument: Instruments can be specified to play a series of notes, and are specified by $. The supported instruments are piano, violin, flute, trumpet, and guitar. If no instrument is specified, the default instrument is the piano.

2.2 Arrays

Array literals are literals enclosed by hard brackets. There are no colons or semicolons between the items in the array. The following are examples of valid arrays:

```
[ $piano $violin $trumpet ]
[ 1 2 3 4 ]
[ “apple” “orange” ]
```

Arrays are strongly typed, and all arrays can only have items of the same type. For example, [1 w “red”] is not a valid array.

2.2.1 Empty Arrays

To create an empty array, one must specify the type before the array literal: int[] creates an empty array with type int.

3. Operators and Expressions

3.1 Identifiers

Variable and function identifiers are sequences of one or more letters and digits where the first character is a letter. Here are several examples of identifiers:

```
chorus1, printHello, song2
```

The following are invalid identifiers that result in a syntax error:

```
1train, sick-beets, _hi
```

3.2 Assignment Operator

The operator = denotes assignment of an expression to a variable identifier. The variable type does not have to be specified, because of type inference.

3.3 Arithmetic Operators

The arithmetic operators are +, -, *, and /. These are all left to right associative, with * and / have higher precedence than + and -.

Table 3.1 explains what each arithmetic operator does:

Operator	**Explanation**
+	Adds values of left and right operands.
-	Subtracts value of right operand from value of left operand.
*	Multiplies values of left and right operands.
/	Divides value of left operand by value of right operand.

3.4 Relational and Logical Operators

Below the arithmetic operators in precedence are the relational operators: >, >=, <, and <=. These operators all have the same precedence. Just below the relational operators in precedence are the equality operators: ==, !=. Below the equality operators is boolean AND: &&, and then boolean OR: ||.

Table 3.2 explains the relational and logical operators, ordered in decreasing precedence.

Operators	**Explanation**
>, >=,<, <=	The relational operators. These compare the values of the left and right operands and evaluate as true or false.

==, !=	The equality operators. These determine whether the left and right operands are equal in value or not.
&&	Boolean AND. Expects left and right operands to be of type boolean.
||	Boolean OR. Expects left and right operands to be of type boolean.

3.5 Array Access

Arrays are accessed with the following syntax:

```
identifier[index]
```

The index must have type int, and must range from 0 to (array length - 1). Elements of an array can be modified using the assignment operator, or retrieved. For example:

```
fruits = [ “apple” “orange” ]
print fruits[1] // prints “orange”
fruits[0] = “banana”
print fruits[2] // syntax error
```

3.6 Musical Operators

: augment - The : operator applies notes to rhythms (or vice versa) to create a tune. Notes and rhythms can be augmented in a one-to-many relationship.

```
tune = q : [ c d e f ]
tune = [ g a b c ] : w
tune = [ q q q q ] : d
tune = c : [ q h q ]
```

If array is augmented with another array, each array must have the same number of elements, or an error will be thrown.

```
tune = [ q q h ] : [ c e d ]
```

3.7 Tunes

A tune is a series of notes with corresponding durations for a given instrument. The default instrument for a tune is $piano, but a tune can have any instrument.

```
     tune = [ q q h ] : [ c e d ]
     violin_tune = tune $violin
     flute_tune = tune $flute
```

We can concatenate tunes as well using the . operator.

```
     piano_tune = [ q q h ] : [ c e d ] . [ w ] : [ g ]
```

3.8 Phrases

A phrase is a combination of tunes across the same duration. A phrase is indicated with brackets {}. Every tune must have the same duration.

```
     chorus = { piano_tune, violin_tune, flute_tune }
```

3.9 Tracks

A track consists of a series of phrases, which can concatenated via the . operator.

```
     song = intro . verse . chorus . verse . end
```

3.10 Comments

Single-line comments are designated by //. Multi-line comments are enclosed by /* */.

```
// This is a single line comment.

/*
   This is a
   multi-line
   comment.
```

*/

4. Control Flow

4.1 if elif else

Keywords “if”, “elif”, and “else” denote conditional statements in which the expression body associated with each conditional is executed iff the boolean expression evaluates to true:

```
    if ( /* boolean expression */ ) {
        /* expression body */
    } elif ( /* boolean expression */ ) {
        /* expression body */
    } else {
        /* expression body */
    }
```

if statements can be stand-alone, but elif and else must have a preceding if statement. An if else block can also be used.

4.3 for-each loop

Keyword “for” denotes the for-each loop that executes the expression body for each item in the array, with the current item accessible through the identifier:

```
    for ( /* item type */ /* identifier */ : /* array of items */ )
    {
        /* expression body */
    }
```

4.3 while loop

Keyword “while” denotes the while loop that will execute the expression body repeatedly as long as the boolean expression remains true:

```
    while ( /* boolean expression */ ) {
        /* expression body */
    }
```

5. Program Structure

5.1 Functions

5.1.1 Defining a Function

Keyword “function” denotes the definition of a function:

```
     function function_name ( /* list of parameters */ ) {
          /* expression body */
          return /* item or value returned */
     }
```

Following keyword “function” comes the name of the function and (a parameter)* enclosed in parenthesis. All functions must have a return value, or an error will be thrown.

5.1.2 Applying a Function

```
     transposed_song = transpose_song (song, 5)
```

Note: When a function is applied, parameters will be constant.

5.2 Scope

The scope of variables is the outermost level of braces in which it is defined. If a variable is declared and not confined with braces, then the scope is within the whole program.

5.3 Multi-Line Expressions

Lines are separated by the newline character. The continuation character used for multi-line expressions is the ‘\’ character

```
     multiLine = 1 + 2 + \
                 3 + 4
```

6. Standard Library

The standard library allows users to configure the tempo of their songs and contains functions helpful for manipulating tracks.

6.1 Tempo

The global variable **tempo** controls the speed at which the song is played, with an immutable time signature of 4/4 and default tempo of 120 bpm. The song will adopt the latest set tempo and is set this way:

```
tempo = 160
```

6.2 Standard Library Functions

Sick Beets comes with three standard library functions: **print**, **render**, and **play**. Each is outlined below:

print function
prints the string argument to standard out

```
print (“string”)
```

render function
creates a MIDI file of the song

```
render (song_name)
```

play function
plays the tune

```
play (tune_name)
```

7. Context-Free Grammar

```
program → epsilon | program stmt | program fdec

fdec → function id ( params ) { stmts }

params → epsilon | id | params , id

stmts → epsilon | stmts stmt

stmt → id = expr | return expr | if (expr) { stmt } elif_block
else_block | while (expr) { stmt } | print ( expr )

elif_block → epsilon | elif_block elif (expr) { stmt }

else_block → epsilon | else { stmt }

expr → literal | [ elements ] | { elements } | expr + expr | expr -
expr | expr / expr | expr * expre | expr == expr | expr != expr |
expr > expr | expr >= expr | expr < expr | expr <= expr | expr : expr
| expr . expr | id ( params )

elements → epsilon | literal | elements literal
```