PoPL Appendix

# Intro Lexical Conventions - Agraj

## Comments

The characters <-- mark the start of a comment and it terminates with the characters -->.

Beats++ only supports comments in a multi line fashion. Comments cannot be nested. The occurrence of comments within a string or character literals is restricted.

## Tokens

There are six classes of tokens encompassing identifiers, keywords, constants, string literals, operators, and other separators. Comments as described below are ignored by the compiler. Whitespace in the form of extra spaces, tabs, newlines etc. are also ignored except when they act as separators for tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

## Identifiers

An identifier is a combination of letters, digits and underscores. Identifiers are case sensitive and can be of any length. Identifiers must begin underscore or letter. There are certain rules established for identifiers based on the type of variable they are representing:

* Identifiers for primitive data types like pitch and duration begin with a lowercase letter.
* Identifiers for derived data types like Note, Measure etc. should begin with an uppercase letter.
* Identifiers for functional identifiers must start with an underscore.

## Keywords

The following identifiers are reserved for the use as keywords, and may not be used otherwise: auto   
//double   
int   
//struct

class   
break   
if

Else  
elif   
//Long

//switch   
//case   
//enum   
//register   
//typedef

replace   
//char   
//extern   
Return

returns  
//union

Const  
float   
//short   
//unsigned   
continue   
for   
//signed   
Void  
//Default  
//goto   
//sizeof

new   
//do   
//static   
while

Func

Null

Import

None

auto

From

Pitch

Duration

Note

Measures

Sheet

Constants - Agraj

Integer constants

Integer constants consist of a sequence of digits without any quotes encompassing them. The standard length for the integer data type is 32 bits. For indication of unsigned integer, as is required for our aliased data type pitch, a suffix of u or U after the digits shall suffice.

Floating constants

A floating constant consists of an integer part, a fraction part and an optional type suffix, u or U. The integer and fraction parts both consist of a sequence of digits. Either the integer part, or the fraction part (not both) may be missing; The decimal point must be present. The type is determined by the suffix; U or u makes it unsigned float, a change necessary for the aliased-double data type freq(short for frequency).

# Meaning of identifiers

* An identifier is a user defined name, which may refer to a function, class, or variable. These identifiers cannot be keywords, which are reserved by the compiler, and carry special meaning(Refer to section xx).
* A variable is specified with: A data type, and a storage class
* Naming conventions:
  + All identifiers are case sensitive
  + All identifiers must begin with a letter (or underscore in case of user defined functions) and only include letters, underscores and numbers
  + Identifiers have to start:
    - with an uppercase letter for derived data types (Note, Measure, Sheet)
    - with a lowercase letter for primitive data types (pitch, int, duration)
    - with an underscore character for functions
* Add off examples from the examples wala submission for incorrect and correct variable names
* Storage Class
  + The storage class determines the lifetime of the storage of the variable. By default, a variable has a storage class of ‘local’, which means the related storage will be discarded at the end of the scope in which it was declared(generally, the current function). By using the keyword ‘heap’ while declaring a variable, it will be stored on the heap(rather than the stack), and the variable will retain its memory and value until the variable is explicitly deleted using the ‘delete’ keyword.
  + Ex:

func \_example[None] returns [None] {

int a = 10; <-- This variable will be stored on the stack, destroyed when function exits -->

heap int b = 2; <-- This variable will be stored on the heap, persists across function calls, and can be passed around. Destroyed only when the ‘delete’ keyword is used -->

}

* Basic types (Need to add range and number of bytes stuff)
  + The primitive datatypes are int, pitch, freq, and duration.
  + Variables declared as type ‘int’ can store integer values between (), corresponding to x bytes of storage(
  + Variables declared as type ‘pitch’ can store integer values between (-100,100). This dataype corresponds to relative frequencies in musical notation, with the base frequency(pitch of 0) corresponding to the note ‘C4’ in western musical notation. This relative frequency is automatically converted to an absolute frequency value internally.
  + Variables declared as type ‘freq’(frequency) can store positive floating point(decimal) numbers. This datatype is used to set the frequency(in Hz) for various operations and other datatypes.
  + Variables declared as type ‘duration’ can store positive integer values in the range(), bytes x. This is analogous to the datatype ‘unsigned int’ in languages like C. This datatype is mainly used to specify the length in time that a musical element will correspond to.(Refer to examples?)
  + Sizes corresponding to the basic datatypes are machine and implementation independent.
  + Derived types(Classes??)
    - The derived types are ‘Note, ‘Measure’, and ‘Sheet’.
    - A Note is simply a tuple of (pitch/frequency, duration). This is the fundamental unit in a musical language. A note can be declared using a pitch or a frequency, and this determines the ‘sound’?, while the duration determines the length in time for which the note will produce sound.
    - A Measure contains a collection of notes, which may overlap with each other, and is of a fixed ‘duration’, which is set while declaring the Measure. (Insert table wala example here)
    - A Sheet is a dynamic array of Measures, placed one after the other. A Sheet can be interpreted musically as an entire musical composition, which can be exported in a number of different ways.
* Functions as a datatype/variable
* In Beats++, a function can be assigned to a variable, and this variable will now possess the same behavior as a variable of any other regular datatype. It can be passed and returned from functions. However, the function type variable cannot be used with the io library functions.
* The syntax for defining a variable that contains a function is as follows:
  + f[Parameter type list]->[Return type list] variable\_name = name\_of\_function;
* examples:
  + func \_example[Note] returns[Measure] {

<-- Function body ->

}

f[Note]->[Measure] function\_variable = \_example;

* Classes, strings, A bit on IO

# Conversions

* It is necessary, under some circumstances, for a variable of one data type to be converted to another datatype, during particular operations. This section covers these conversions(Such originality why not fucking copy the manual)
* Add off float type-> Can have float/int conversion too
* Duration & int conversion
  + Since the datatype ‘duration’ is simply an unsigned integer, it is possible to convert a variable of type ‘duration’ to type ‘int’. The value is unchanged and can now be represented in the new type.
* Measure/ Song conversion in concatenation
  + While using the concatenation operator ‘+’, a Measure is automatically converted to a Sheet(dynamic array of Measures) of length 1, to enable concatenation.
    - Exs: Measure + Measure -> Sheet
    - Sheet + Measure -> Sheet
  + Ex1:

Sheet Sheet1 = Measure1 + Measure2;

Sheet Sheet2 = Sheet1 + Measure1;

<-- This is incorrect usage -->

Measure Measure3 = Measure1 + Measure 2;

* Pitch - Frequency Conversion(Normally and during operation, internally in Note as well needs to be specified)
  + The datatypes ‘pitch’ and ‘frequency’ are closely related, as pitch simply corresponds to a relative frequency, relative to a base frequency of 261.63Hz (Note C4 in Western Musical Notation). Hence, variables of type ‘pitch’ can be converted to type of ‘frequency’, by converting the relative frequency value to an absolute frequency. This conversion is what allows the ‘Note’ datatype to be initialized using pitch or frequency.
  + For convenience, the typecasting for pitch to frequency will be implicit. Frequency cannot be typecast into pitch, since all frequencies do not correspond to a relative pitch.  
    Reference: <https://pages.mtu.edu/~suits/notefreqs.html>(Make this a hyperlink in md)
  + Ex1:

pitch sample\_pitch = 0;

freq sample\_freq = sample\_pitch;<-- sample\_freq gets value of 261.63Hz(The base frequency on the assumed relative scale -->

pitch sample\_pitch2 = -3;

freq sample\_freq2 = sample\_pitch2;<-- sample\_freq2 gets value of 220.00Hz -->

* + Ex2:

<-- The pitch-frequency conversion allows a Note to be initialized using either one of the two datatypes -->

duration sample\_duration = 2;

pitch pitch1 = 0;

freq freq1 = 261.63;

<-- All the below Notes are identical -->

Note Note1 = (pitch1,sample\_duration);

Note Note2 = (freq1,sample\_duration);

Note Note3 = (0,2);

Note Note4 = (261,63,2);

# Expressions

## Operators

Relational Operators

< Lesser than

> Greater than

<= Lesser than or equal to

>= Greater than or equal to

Expressions using these operators all evaluate to either 0 (if the expression is false) or 1 (if the expression is true). The result’s datatype is *int* by default. These operators require two operands, are grouped in a left-to-right fashion and multiple operators can’t be used one after the other (i.e., a > b > c is not allowed). The relational operators can only be used only to compare two operands of the same data type. But the . operator helps bypass this particular restriction (as explained later).

Equality Operators

=:= Equal to

=/= Not equal to

These operators are the same as the relational operators in all ways except for the fact that they have lower precedence than their relational counterparts (i.e., a > b =/= c <= d is parsed as (a > b) =:= (c <= d)).

The Dot (.) Operator

The . operator is used as a predecessor to the other relational and equality operators when one wants to compare just the values of the operands regardless of their datatype. This operator facilitates comparing just numerical values by overlooking the data type. For example, 400 .<= 880.00 would result in 1 (true).

Logical Operators

#### AND Operator

This operator groups left-to-right and evaluates to true only when both operands are true, and is false in all other cases.

Beats++ has two kinds of logical AND operators:

* &   
  This operator doesn’t short circuit the expression and evaluates both sides or operands.  
  i.e., (a > b) & (c++ < d) will always increment c regardless of whether a > b evaluates to true or not. Note that this is not the same as the bitwise ‘and’’ as defined in C like languages.
* &&  
  This is the short-circuit operator and evaluates the right-hand side only if the left-hand side evaluates to true first, otherwise, it directly sets the result to false.

i.e., (a > b) && (c++ < d) will only increment c if a > b evaluates to true, else the result will directly be set to false

#### OR Operator

This operator groups left-to-right and evaluates to false only when both operands are false, and is true in all other cases.

Beats++ has two kinds of logical OR operators:

* |   
  This operator doesn’t short circuit the expression and evaluates both sides or operands.  
  i.e., (a > b) | (c++ < d) will always increment c regardless of whether a > b evaluates to false or not. Note that this is not the same as the bitwise ‘or’ as defined in C like languages.
* ||  
  This is the short-circuit operator and evaluates the right-hand side only if the left-hand side evaluates to false first, otherwise, it directly sets the result to true.

i.e., (a > b) || (c++ < d) will only increment c if a > b evaluates to false, else the result will directly be set to true

Assignment Operators and Expressions

There are multiple assignment expressions, all of which are grouped from left to right. The following are the expressions supported:

= \*= /= -= %= += ++=

All the above operators require compatible datatypes on both sides of the expression. The ++ operator is the superimposition operator which can only be used with the Measures data type.

An expression of the form E1 op= E2 is equivalent to E1 = E1 op E2.

For example:  
a += c is equivalent to a = a + c.

a is reassigned the value of a (old) + c.

# Declarations - Everyone (Tarun - Function declaration)

# Function calls

A function call consists of the identifier for the function followed by square brackets, within which a comma-separated list of assignment expressions (possibly empty), which constitute the arguments to the function. If no declaration for the identifier exists within the scope of the call, it is searched for within the standard library as well as any imports made into the program. Example call:

*list-of-variables-to-store-return* = *function-identifier*[*list-of-arguments*];

Note: In case the function doesn’t have a return value, the assignment component of the above statement shall be omitted.

The list of return variables shall be separated by commas and be in the order that is compatible with the return type of the function.

In preparing for the call to a function, a copy is made of each argument of primitive type; all argument-passing for the primitive data types is strictly by value. However, arguments of any derived data type are passed by reference to the original variable. This implies that changes made by the function to primitive arguments are independent of the original variables, whereas those made on derived data types are reflected in the original variables.

Function Declarators

Functions declaration consist of three major components:

* Identifier name
* List of parameters
* List of return values

Each function declaration is preceded by the *func* keyword. This is followed by the identifier for the function.

Immediately after the identifier comes the list-of-arguments the function takes in. Each argument is separated from others by a comma and the list is wrapped in square brackets. The *returns* Keyword is used to separate the argument list from the list of return values. Return values take the same form as the list for arguments i.e. comma separated and wrapped in square brackets. An empty set of square brackets is used to indicate no arguments or no return values. Following the return values are a pair of curly braces that encompass the function body. Each function must mark its end by the keyword *return* which is followed by the return value list, if any. Identifiers used for arguments in the function declaration are simply placeholders and are not bound to any actual variable within the scope and are thus free to use

More on Function Declaration

* A function declaration of a function ‘X’ has the format:func X[parameter list] returns [return type list] { <-- Function Body --> }
* The parameter list has the format: datatype1 variable1, datatype2 variable2, ….
* Beats++ supports multiple return values from a single function. Hence, the need for a return type list is needed. The format for the same is : return\_datatype1, return\_datatype2, ….
* The datatypes of the variables returned via return statement at the end of the function body should match the datatypes mentioned in the return type list.
* If the function takes no parameters, or has no return values, then the keyword ‘None’ should be specified in the corresponding list during declaration.

## Variables

## Class/Struct Declaration

# Statements, External Declarations - Soumi

## Compound Statements

The compound statements in Beats++ works exactly as it does in most other C-like languages.

Multiple statements when wrapped with curly braces {} are called a block and acts like a single statement. All function bodies are compound statements, as are loop and conditional statement bodies when there are more than one statements to be executed in its body.

## Selection Statements

Beats++ provides if, else and elif statements as conditional statements.

if (expression) {

<--statements-->   
}

This is an if statement without a corresponding else statement. The statements inside it will only be executed if the expression evaluates to true, or is a non-zero or non-null value.

if (expression) {

<--statements-->   
}

else {

<--statements-->   
}

This is an if statement coupled with an else statement. The statements inside the if will only be executed if the expression evaluates to true, or is a non-zero or non-null value. If the expression evaluates to false, the statements inside the else will be executed.

if (expression) {

<--statements-->   
}

elif (expression) {

<--statements-->   
}

else {

<--statements-->   
}

This is an if-elif-else ladder. There is no bound on the number of elif’s allowed between the starting if and the ending else. It is also not compulsory to have an else statement in the end.

## Iteration Statements

The syntax for the following loops are the same as C-like languages so there isn’t much of a learning curve for new users of Beats++.

### While Loops

while (expression) {

<--statements-->   
}

In the while loop, the statements in the body of the loop are executed repeatedly as long as the expression evaluates to true (or has a value other than 0 or null). The evaluation of the expression happens before every iteration or execution of the inner statements.

### For Loops

for (expression1; expression2; expression3) {

<--statements-->   
}

Expression 1 specifies the initialisation for the loop and is evaluated only once, right before the first iteration of the loop.

Expression 2 is evaluated before each iteration and is analogous to the single expression in the while loop. Thus, the for loop is terminated if expression 2 evaluates to false.

Expression 3 is evaluated after each iteration, and acts like an update or re-initialisation statement for the loop.

None of the three expressions is compulsory and any of them may be dropped while using a for loop.  
Thus,

for ( ; ; ) {

<--statements-->   
}

is also a perfectly valid loop.

## Jump Statements

Beats++ provides two jump statements which can only be used inside loops: break and continue, and one jump statement which can only be used inside functions: return.

#### break

This keyword is used to break out of a loop regardless of whether the loop’s expression evaluates to false and passes the control to the block enclosing the loop.

#### continue

This keyword is used to force the next iteration of a loop and skips all the statements following it for that particular iteration.

#### return

This keyword is used by functions to return its caller. Beats++ allows functions to return multiple values (elaborated upon in the functions section). Having no return statement is equivalent to having a return statement not followed by any expression. The return keyword is classified as a jump statement because a function can only have one return statement and any statement or blocks following the return statement will be skipped.

Scope Linkage

## Scoping

The same identifier cannot be used for different purposes in the same scope, even if the usage could be different in the scope.

#The lexical scope of a class or function identifier in an external declaration begins at the immediate end of its declaration and persists to the end of the translation unit in which it appears.

The scope of a parameter of a function definition begins at the start of the function and remains till the end of the function (but not outside the function, it ends at the end of the function declarator).

The scope of an identifier declared at the start of a function begins at the immediate end of its declaration, and remains till the end of the function (but not outside the function, it ends at the end of the function declarator).

//If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.

Linkage

Within a translation unit, all declarations of the same object or function identifier with internal linkage refer to the same thing, and the object or function is unique to that translation unit that is destroyed on moving out of its scope. All declarations for the same object or function identifier with external linkage i.e imports refer to the same thing, and the object or function is shared by the entire program. For this purpose imports are made at the start of a program, outside of *start*[]. Imports are made in two ways: from pre-existing libraries or user-created packages (these are just files without a *start[]* and are meant to store custom classes and data type templates created by the user. For import from a standard library, the statement is as follows

import *library-name*;

For imports of specific functions/classes from a library:

import *function-name,class-name* from *library-name;*

Note that either of *function-name and class-name* can be omitted from the above statement. Also, multiple imports from the same library/package can be separated via the comma separator.

For imports from user-created packages, the *library-name* should be replaced by an absolute or relative path to the package from the working file’s directory.4

Compiled vs Interpreted

Beats++ is built around the objective of providing musicians an easy pathway to produce sheet music and customise the “sheet” with great speed. Keeping the above in mind, one of the most important things to keep into account is low runtime. This is based on the motivation to make the execution of programs as fast and efficient as possible. Combined with the relatively lightweight syntax and lack of pointers, Beats++ proves to be a relatively easy language to debug. Combining the previous two conditions, Beats++ is made to be a compiled language, with executables produced on compilation characterised by the “.bpm” extension.

# 

# 

# 

# Preprocessing - Soumi

## Macros (#replace)

# Extras

Expressions intro

An expression is any valid unit of code that resolves to a value. An expression may

consist of one or more 'operands', and zero or more 'operators' to produce a value.

All the 'operators' are defined below in the remainder of this section. Each operator

can support a particular set of types for the operands, and some operators are 'overload

to support different operations. Depending on the types of the operands, each operator

decides the type of operation to apply.

The precedence of evaluation of the various operators is defined in section x.

Beats++ supports the following types of operators:

- Arithmetic

- Logical

- Musical

The 'operands' may be any valid variable, constant, or another sub-expression, as long a

the types of the variable, constant, or the sub-expression (after evaluation) are

supported by the operator in use.