Cb

Programming for musicians.

Language Reference Manual

Mehmet Erkilic (me2419)
Marcellin Nshimiyimana (mn2587)
Kyle Rego (kar2150)
Cole Diamond (cid2105)
Matthew Cowan (mpc2145)

Table of Contents

Introduction

The Cb language is designed to be the most intuitive language for a musician to not only write basic music quickly with a focus on chord creation and manipulation, but include more algorithmic music compilation as naturally as possible. This manual describes the syntax for the Cb language.

Lexical Conventions

Whitespace

Spaces, tabs, and newlines (collectively, "white space") are ignored except when used as separators. Separators are white space that is needed to separate otherwise adjacent identifiers, keywords, and constants.

Comments

A comment, whether single or multiline, goes between <- characters, which indicates the start of it and ->, which indicate the end. The comment can be placed anywhere in the program as long as it is between these two characters. Comments do not nest and are ignored.

Ex:

```
<- create a chord with three notes with a duration of 1/8
    note that the duration of the chord overrides that of the notes->
Note c = (C, 0, HALF);
Note g = (G, 0, HALF);
Note e = (E, 0, HALF);
Chord cr = (c, g, e, EIGHTH)
```

Identifiers

In Cb language, an identifier is a sequence of letters, digits, and underscores "_". An identifier must start with a letter or an underscore and may not start with a number. There is no limit on how long an identifiers can be.
Below is the list of characters allowed in creating an identifier:

```
a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z _
0 1 2 3 4 5 6 7 8 9
```

Ex: nice_note, NICE_note, and _NOTE2 are acceptable identifiers. However,

1nicenote and 2nicenote are not acceptable identifiers

Keywords

Keywords are identifiers used for specifying the types of expressions and for including methods from an external packages. These keywords listed below are reserved for Cb, which means that they cannot be used as normal identifiers.

Int	is	isnt
Note	if	meth
Chord	else	return
Scale	while	compose
Stanza	foreach	end
Score	in	elsif
and	or	use

Literals

Cb uses only Integer literals that consist of a sequence one or more digits.

Constants

Integer constant

Cb has a set of Integer constants that are used to represent basic notes and known durations of notes. These can also note be used as normal identifiers. Below is a list of Integer constants:

A	B	C	D	E	F	G
A#	B#	C#	D#	E#	F#	G#
Ab	Bb	Cb	Db	Eb	Fb	Gb

A rest pitch constant is "R"
Note rest = (R, 0, HALF); <-Create a half note rest ->
<- the octave here doesn't matter ->

SIXTEENTH	EIGHTH	QUARTER	HALF	WHOLE

Operators
An operator specifies an operation to be performed. Operators are described in depth in the Expressions section.

Punctuators
A punctuator is a symbol that adds semantic value to the expression or statement that it belongs to, but does not perform an actual operation. These punctuators are used in declaration and assignment of variables. Below is a list of Punctuators:
[ ] ( ) . ;
Ex:

```
Note asharp = (A#, 0, QUARTER);
<- do re mi song ->
meth Stanza doremi(Int duration)
        Note do = (C, 0, duration);
        Note re = (D, 0, duration);
        Note mi = (E, 0, duration);
        Stanza s = [do, re, mi];
        return s;
end
```

Meaning of Identifiers

Disambiguating Names
A Cb identifier is disambiguated mainly by the following characteristics: lexical uniqueness and function scope

Lexical Uniqueness
Cb identifiers are created with a combination of Latin characters and the underscore character as specified (and constrained) in the section, *Lexical Conventions*. All identifiers are first disambiguated by its lexical name being different from all other identifiers in the file.

Method Scope
Cb identifiers have nearly no scope (brackets have no effect on the life of a variable); all variables are global to the file. However, a single exception is made to any variables declared within a method to avoid the unintentional manipulation of values. This decision was made to address the possibility of unwanted variable mutation. For example:

```
meth Note test()
        Note n = (D, 0, whole*2);
        return n;
end
Note n = (C, 0, 1);
test();
```

Here, a note is declared twice; once while defining a method and once after defining the method. Moreover, the method is subsequently called after the second declaration. In this scenario, having method scope is important since it is likely that the user does not want to let the method call alter the declaration/initialization of identifiers he/she makes beforehand. In this situation, 'n' will still have a value (C, 0, 1).

Types

There is 1 basic type: integers, and there are 5 derived types: a *Note*, *Chord*, *Scale*, *Stanza*, and *Score*. Their identifiers are listed below:

Type:
Int, Note, Chord, Scale, Stanza, Score

Basic Types

An integer specifies a whole, signed integer denoted by the keyword "Int"
Ex. *Int x = 5;*

Derived Types

A note is defined by a string representing a note constant, an integer ([-5, 5]) representing octave displacement, and a positive integer representing duration. The duration value refers to a multiple of 1/64 (the 64th note). So a duration of 2 is equivalent to a 32nd note and a value of 4 is equivalent to a 16th note. You may also use the duration constants (SIXTEENTH, EIGHTH, …) to help you with defining the duration of a note. You can also use the * operator to help ease the 1/64th base multiples.
Ex. *Note n = (A, -3, 72); Note n = ("C", 2, HALF*2)*

A chord is defined by a list of notes and a positive integer representing duration.
Ex. *Chord c = ([n1, n2, n3], 70)*

A scale is defined by a list of notes
Ex. *Scale s = [n1, n2, n3]*

A stanza is defined by a list of both chords and notes
Ex. *Stanza sz = [n1, n2, c1, n3]*

A score is defined by a list of stanzas
Ex. *Score sc = [sz1, sz2, sz3]*

lvalues

Identifiers may serve as an lvalue (short for 'left value'), an expression referring to an object. In the expression X1 = X2, the left operand, X1, is the lvalue (represented by the identifier 'X1'). Utilizing identifiers as an lvalue means that a user is able to modify rvalues, the expression residing on the right side of an assignment statement.

Declarations

Declaration Syntax

Function definitions have the form:

function-definition:
type identifier(parameter-listopt) compound-statement

parameter-list:
type-specifier identifier
parameter-list, type-specifier identifier

Type is one of the following keywords: int, bool, note, chord, stanza, scale, score
Identifier is a non-reserved alpha-numeric sequence as described in section X.X

Compound-statement is any legal code that returns a value of agreeable type with the declaration.

Blocks

A block is a section of code enclosed by *meth* and *end* keywords. 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.

Scope

The scope of an identifier is the subsequent statements within the block of code where it is declared including blocks nested in that block. Declarations can appear after certain keywords that open a block of code. These keywords are meth, while, 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.

Identifier Naming

All identifiers within a block of code must be unique and a nested 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.

Expressions

In Cb, expressions consist of one or more operators in tandem with operands. Associativity rules determine precedence, but parentheses can override the default orderings. The two most pervasive expressions in Cb are assignment expressions and operation expressions. The table below outlines the associativity rules of the Cb's built in functions.

Tokens (Descending Priority)	Operators	Class	Associativity
Identifiers, constants, parenthesized expression	Primary expression	Primary	
() [] .	Function calls, subscripting, direct selection	Postfix	L-R
+ - * / % ^	Arithmetic and augmentation	Binary	L-R
is isnt	Equality comparisons	Binary	L-R
< <= >= >	Relational	Binary	L-R
and	Logical And	Binary	L-R
or	Logical Or	Binary	L-R
= += -= *= /=	Assigment	Binary	R-L

Primary expressions

Identifier

An identifier typifies a primary expression. Its declaration calls for the specification of a type of the identifier followed by the value of the identifier. It can refer to an lvalue or a function designator.

Constant

An integer, decimal, character, or floating constant is a primary expression of constant value. The capitalized letters A-G are constant expressions that each represent Notes of default duration having pre-defined values associated with the notes A-G, respectively. Naturally, *Note* constants are the most frequent example of constants in Cb.

Parenthesized expression

A parenthesized expression is a primary expression of the form (expression). It can be used to override precedence. For example, consider the two expressions below.
Expression 1: (note1 > note2) and (note3 < note2 or note3 < note1)
Expression 2: (note1 > note2 and note3 < note2) or (note3 < note1)
While the former will "and" the two subexpressions together, the latter will instead apply the "or" operator to the result.

Postfix

Postfix calls can be function calls, subscripts or direct selection. An example of each, respectively:

```
Note1.arpeggiate(2, 3);
Chord1[3] = Note1;
Chord1.length;
```

Function Calls

A function call is characterized by a primary expression followed by parentheses enclosing an optional comma-seperated list of expressions. These expressions form the arguments to the function. Each and every function in Cb must be declared before it is called. The method signature must consist of [meth return_type method_name (argument-expression-list). The argument expression list may either be

a single argument or a list of arguments. Additionally, the return argument must match the return type in the method signature.
A copy of each parameter is created in advance of the function call. As result, Cb uses argument-passing by value. Although a function may change the values of the parameters, the changes will not affect the values of the parameters. Recursive function calls are honored in Cb.

Subscripting

Only Chords and Stanzas can be operated on using the subscripting operation. For example, subscripting applied to a Chord can be used to select a particular note. The subscript operator allows both retrieval and mutation of elements.

Direct Selection

Pitch and duration in objects of type Note and Chord can be changed through directly accessing the objects. For example, A.pitch += 2 will result in C. The same paradigm applies to duration objects as well. Direct selection can be applied to Stanza as well to access the length.

Unary Operations

Whole-Step Increment/Decrement Operations

Plus-plus (++) and minus-minus (--) operations of the form (expression)++ or (expression)-- can be used for a variety of purposes. When applied to a Note, the plus-plus or minus-minus operator will augment or diminish the Note, respectively. Analogously, the plus-plus or minus-minus operator applied to a Chord will augment or diminish each of the constituent Notes.

Ex:
Note n = (G, 0, 1);
n = n++;
<- Now n is the note with pitch of A ->

Octave Increment/Decrement Operations

Carrot-plus (^+) and Carrot-minus (^-) operations of the form (expression)^+ or (expression)^- will shift a single Note or all constituent Notes of a Chord or Stanza up

or down an octave. Specifically, carrot-plus will shift up an octave while carrot-minus will shift down an octave.

Ex:
Note n = (G, 0, 1);
n = n^+;
<- Now n is the note with pitch of G transposed up one octave ->

Binary Operations

Add and Subtract

Add and subtract binary operations can be applied to a multitude of objects. In general, any object added to another object of the same type will result in the concatenation of the two objects. For example, the plus operator applied to a Chord or Stanza, the result is a concatenated or reduced sequence. When applied to a Note, the Note is augmented or diminished by the argument of the expression. Chords can be added to Stanzas through the add and subtract methods but number literals cannot be added to chords, notes or stanzas.
The syntax is as follows:
Add-expression: add-expression + add-expression
Subtract-expression: subtract-expression – subtract-expression

Multiply, Divide and Modulus

Multiply can applied to Note, Chord, Scale and Stanza objects to create copies of the instance as well as to numbers to apply regular multiplication rules. Division and modulus can only be applied to real numbers.
The syntax for each of these expressions is analogous:
Multiply-expression: multiply-expression * multiply-expression
Divide-expression: divide-expression / divide-expression
Modulus-expression: modoulus-expression % modulus-expression

Augmentation Operator

The augmentation operator (^) can be applied to a note to augment the note by a number of octaves or to a chord to transpose every constituent note by a number of octaves. Note ^ (Number) or Chord ^ (Number) exemplifies the syntax of the carrot operator. The range of allowable octaves for any note to assume the value of is -5 to +5.

Relational Comparisons

Yields a Number result (1 if true, 0 if false) that uses the following syntax:
Relational-expression:

relational-expression < relational-expression
relational-expression > relational-expression
relational-expression >= relational-expression
relational-expression <= relational-expression

Equality Comparisons

Determines if two values are equal. Cb uses 1 to denote true and 0 to denote false. The token "is" denotes equality while "isnt" denotes inequality.

The following rules govern equality relations:
Two Number objects are equal if they have the same value.
Two Note objects are equal if they have the same duration and pitch.
Two Chord objects are equal if they consist of the same notes for the same duration
Two Stanza objects are equal if they have the same chords and notes in the same order.
Equality Comparisons take the following form:

Equality-expression is equality-expression
Equality-expression is not equality-expression

Logical Operators

The symbols "and" and "or" perform a logical and, or operation on two expressions, respectively. If the expression evaluates to false, then a zero is returned. Otherwise, 1 is returned. Lazy evaluations or "short-circuiting" is supported.
Logical-expression:

logical-and-expression and logical-and-expression
logical-or-expression or logical-or-expression

Assignment

Assignment is a right associative operation – the expression on the right is evaluated and then used to set the lvalue. The rvalue must have the same type as the lvalue since no casting is implicitly done.

Commas

Commas are used to separate list elements like parameters in a function or Notes in a Chord. Consider, for example, *Chord chord = ([noteA, noteB], dur)*. Moreover, a pair of expressions separated by a comma is evaluated left-to-right and that the type and value of the result are identical to the type and value of the right operand.

Expressions of the form [Operation]-Equals

The tokens "+=", "-=", "/=", "*=" can be used to modify the state of a variable by a given amount. For example, A += 2 will return a Note of value C with a default duration. Each of the operators uses the pre-defined operations of addition, subtraction, division and multiplication to compute the result.

Statements

Except as indicated, statements are executed in sequence. Statements are executed for their effect, and do not have values. They fall into the following categories:

statement:
    expression;
    return expression;
    conditional-statement;
    while-statement
    foreach-statement

Expression statement

expression ;

Most statements take this form, as assignments or function calls. All side effects from the expression are completed before the next statement is executed.

Compound statement

statement-list:
    statement
    statement-list statement

Inside methods and other structures there is the concept of multiple statements.

Conditional statement

elsif-statement:
    / nothing */*
    elsif-statement elsif (expr) statement-list

if (expr) statement-list elsif-statement END
if (expr) statement-list elsif-statement ELSE statement-list END

In both cases the expression is evaluated and if it is nonzero or the bool value of true, the first substatement is executed. The elsif part is entirely optional. If included the program will continue evaluating the expression specified for each elsif statement and execute the statements of the block that evaluates to true or a nonzero number. If an else clause is included its code will be executed if none of the prior elsif conditions or the if condition were accepted.

While statement

while (expr) statement-list END

The while statement allows for looping over the statement-list as long as the expr evaluates down to true. This means the expr evaluates to either a nonzero integer or the bool value true.

Foreach statement

param-decl:
    DATATYPE ID

foreach (param-decl IN ID) statement-list END

The foreach statement allows for looping over all elements of the specified datatype in the specified item.

Return statement

return expression ;

A function returns to its caller by means of the *return* statement, which must be of the form expressed above. In Cb a value must be returned by all methods.

Appendix

Required built in method "compose"

The compose method on a score object terminates the program and allows the compiler to write all of the contents of the score to the MIDI file. The compose method takes an integer as its only parameter to represent the tempo (beats per minute). All lines of code after this statement are ignored.

Example use of a package:

```
Assume there is a package called practice.pcb
<- create a scale that repeats n times, with increasing pitches.
    return a stanza with all scales comined.
 ->

meth Stanza doremi(Int octave, Int duration, Int n)
        Int o = octave;
        if (o == 5)
                o = 4;
        end
        Note do = (C, o, duration);
        Note re = (D, o, duration);
        Note mi = (E, o, duration);
        Note fa = (F, o, duration);
        Note so = (G, o, duration);
        Note la = (A, o, duration);
        Note ti = (B, o, duration);
        Note upper_do = (C, o+1, duration);
        s = [do, re, mi, fa, so, la, ti, upper_do];
        Stanza st = [];
        Int c = 0;
        Note rest = (R, 0, whole);
        While(c < n)
                st.put(s);
                st.put(rest);
                s^+;
                c = c +1;
        end
```

```
        return st;
end
```

In a program called practice.cb:

```
<- include package.pcb->
use package.pcb
Int dur = quarter;
Int oct = 1;
Stanza drm = doremi(oct, dur, 3);
drm.compose();
```

---

parser.mly

```
%{ open Ast %}

%token <int> INTLITERAL
%token <int> OCTAVE /* integer between -5 and 5 */
%token <int> DURATIONINT /* positive intege x>0 */

%token <string> DURATIONCONST /* whole half etc. */
%token <string> DATATYPE
%token <string> NOTECONST  /* Goes to string A or B or any note*/
%token <string> ID

%token IN
%token IF
%token ELSE NOELSE
%token WHILE FOREACH
%token ASSIGN
%token PLUSEQ
%token MINUSEQ
%token TIMESEQ
%token DIVIDEEQ
%token MOD
%token MODEQ
%token PLUS
%token MINUS
%token TIMES
%token DIVIDE
%token IS
%token ISNT
%token LT
%token LEQ
```

```
%token GT
%token GEQ
%token PLUSPLUS
%token MINUSMINUS
%token SHARP
%token FLAT
%token RAISE
%token LOWER

%token LEFTPAREN RIGHTPAREN LBRAC RBRAC
%token INT NOTE CHORD SCALE STANZA SCORE

%token METH RETURN END
%token PLUS MINUS TIMES DIVIDE

%token ASSIGN
%token VASSIGN          /* Variable Assign  only used for variable declaration */
%token SEMICOLON
%token COMMA DOT

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc ELSIF
%left PLUSEQ MINUSEQ
%left TIMESEQ DIVIDEEQ MODEQ
%right ASSIGN
%left IS ISNT
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%left PLUSPLUS MINUSMINUS RAISE LOWER
%left SHARP FLAT
```

```
%start program
%type <Ast_tmp.program> program                /* ocamlyacc: e - no type has been declared for the start symbol `program'*/
%%

program:
{ [], [] }
| program vdecl { ($2 :: fst $1), snd $1 }
| program methdecl { TODO() }

methdecl:
        METH DATATYPE ID LEFTPAREN meth_params RIGHTPAREN statement_list END { create() }

meth_params:
        { [] }
        | param_list { List.rev($1) }

param_list:
        param_decl { [$1] }
        | param_list COMMA param_decl { $3 :: $1 }

param_decl:
        DATATYPE ID { TODO() }

statement_list:
        { [] }
        | statement_list statement { $2 :: $1 }

statement:
        expr SEMICOLON { TODO() }
        | RETURN expr SEMICOLON { Return($2) }
        | IF LEFTPAREN expr RIGHTPAREN statement_list elsif_statement %prec NOELSE END { TODO() }
        | IF LEFTPAREN expr RIGHTPAREN statement_list elsif_statement ELSE statement_list END { TODO() }
        | WHILE LEFTPAREN expr RIGHTPAREN statement_list END { TODO() }
```

```
        | FOREACH LEFTPAREN param_decl IN ID RIGHTPAREN statement_list END {TODO()}

elsif_statement:
    /* nothing */ { [] }
        | elsif_statement ELSIF LEFTPAREN expr RIGHTPAREN statement_list { TODO() }

vdecl:
        DATATYPE ID SEMICOLON {{ vartype = $1; varname = $2}}
        | NOTE ID VASSIGN LEFTPAREN NOTECONST COMMA OCTAVE COMMA duration_expr RIGHTPAREN SEMICOLON { TODO() }
        | INT ID VASSIGN INTLITERAL SEMICOLON { create($2) }/* int x = 5; */
        | CHORD ID VASSIGN LEFTPAREN LBRAC generic_list RBRAC COMMA duration_expr RIGHTPAREN SEMICOLON  { TODO() }
        | SCALE ID VASSIGN LBRAC generic_list RBRAC { TODO() }
        | STANZA ID VASSIGN LBRAC generic_list RBRAC { TODO() }
        | SCORE ID VASSIGN LBRAC generic_list RBRAC { TODO() }

generic_list:
        { [%1] } /* cannot have empty */
        | generic_list COMMA ID { $3 :: $1 } /* Depends on the type of id */
        | generic_list COMMA ID TIMES INTLITERAL { TODO() }

duration_expr:
        DURATIONINT { $1 }
        | DURATIONCONST { $1 }
        | duration_expr PLUS duration_expr { Binop($1, Add, $3)  }
        | duration_expr MINUS duration_expr { Binop($1, Sub, $3)  }
        | duration_expr TIMES duration_expr { Binop($1, Mult, $3)  }
        | duration_expr DIVIDE duration_expr { Binop($1, Div, $3)  }

expr:
        ID { Id($1) }
        | ID DOT ID { TODO() }
        | INTLITERAL { TODO() }
        | ID LBRAC expr RBRAC { ElemOp($1, $3) }
```

```
        | ID LBRAC expr RBRAC ASSIGN expr { LElemOp($1, $3, $6) }
        | ID ASSIGN expr { TODO() }
        | expr PLUSEQ expr { Assign($1, BinOp($1, Add, $3)) }
        | expr MINUSEQ expr { Assign($1, BinOp($1, Sub, $3)) }
        | expr TIMESEQ expr { Assign($1, BinOp($1, Mult, $3)) }
        | expr DIVIDEEQ expr { Assign($1, BinOp($1, Div, $3)) }
        | expr MODEQ expr { Assign($1, BinOp($1, Mod, $3)) }
        | expr PLUS expr { BinOp($1, Add, $3) }
        | expr MINUS expr { BinOp($1, Sub, $3) }
        | expr TIMES expr { BinOp($1, Mult, $3) }
        | expr DIVIDE expr { BinOp($1, Div, $3) }
        | expr MOD expr { BinOp($1, Mod, $3) }
        | expr IS expr { BinOp($1, Eq, $3) }
        | expr ISNT expr { BinOp($1, NEq, $3) }
        | expr LT expr { BinOp($1, Less, $3) }
        | expr LEQ expr { BinOp($1, LEq, $3) }
        | expr GT expr { BinOp($1, Greater, $3) }
        | expr GEQ expr { BinOp($1, GEq, $3) }
        | expr PLUSPLUS { Assign($1, BinOp($1, Add, IntLiteral(1))) }
        | expr MINUSMINUS { Assign($1, BinOp($1, Sub, IntLiteral(1))) }
        | expr SHARP { TODO() }
        | expr FLAT { TODO() }
        | expr RAISE { TODO() }
        | expr LOWER { TODO() }
        | LEFTPAREN expr RIGHTPAREN { $2 }
        | ID LEFTPAREN actuals_opt RIGHTPAREN { TODO() }

actuals_opt:
        { [] }
        | actuals_list { List.rev $1 }

actuals_list:
        expr { [$1] }
```

```
| actuals_list COMMA expr { $3 :: $1 }
```