m

A language for music generation.

Language Reference Manual

Yiling Hu (yh2378)
Monica Ramirez-Santana (mir2115)
Jiaying Xu (jx2129)

Table of Contents

Language Reference Manual

Introduction

This manual describes the syntax for the m language, a language for algorithmic music generation.

Lexical Conventions

An m program consists of a single file with the syntax described in this document.

Tokens

There are five types of tokens: identifiers, keywords, operators, literals, and separators.

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 pitch constants.

Comments, as described below, are also ignored.

If the input stream has been separated into tokens up to a given character, the next token is the longest string of characters that could constitute a token.

Comments

There are two methods to insert a comment: (1) the characters /* introduce a comment, which terminates with the characters */ (2) the characters // introduce a comment, which terminates at the following newline character. Comments do not nest.

Identifiers

An identifier is a sequence of letters (including _) and digits, the first character of which is a letter a-z or A-Z. Identifiers may be of any length. In identifiers, upper- and lowercase letters are different.

Keywords

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

part	int	elseif
staff	float	for
note	bool	while
chord	if	true
void	else	false

Literals

There are four types of literals: integer literals, float literals, Boolean literals, and pitch literals.

literal:

integer-literal
floating-literal
boolean-literal
pitch-literal

Integer Literals

An integer literal consists of a sequence of digits is always interpreted as a decimal number.

Floating Literals

A floating literal consists of an integer part, a decimal part, and a fraction part.

[integer].[fraction]

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.

Pitch Literals

A pitch literal is a sequence of characters and digits in one of the two formats outlined below:

Either:

$$[note][modifier]_{opt}[octave]$$

note: accepts the values a-g and A-G.

modifier: accepts the values s, f, S, F. May be omitted.

octave: accepts the values 0-9.

Or:

The value r or R.

Examples of pitch constants: As7, af7, A7, r

Identifiers

Identifiers are names that refer to functions and objects. An object is a named region of storage, and a variable is an identifier that refers to an object.

A variable has a scope and type. A scope determines the lifetime of the storage associated with the variable. The type indicates how the data contained at the storage location is interpreted.

Scope

Scope is the region of the program in which a variable is known.

There are two kinds of scopes: static and automatic. The context of an object's declaration determines its scope. Static variables are all variables that are defined outside of any blocks within the program. Blocks are any segment of code encased in {} braces. Automatic variables are all variables that are defined inside of a block.

Static variables are globally known. Automatic variables are known inside of the block in which they are defined as well as all blocks contained within that block.

All variables are known within their scope only after their declaration.

Types

There are four basic types: void, integer, floating point, and Boolean, and six derived types: note, chord, staff, part, function, and array.

type:

void
int
float
bool
note
chord
staff
part
function
array

Basic Types

The void type specifies an empty set of values and is denoted by the keyword void.

The integer type specifies a signed integer and is denoted by the keyword int.

The floating point type specifies a signed floating point number and is denoted by the keyword float.

The Boolean type specifies a truth value of either true or false and is denoted by the keyword bool.

Derived Types

The note derived type contains three objects: pitch, duration, and intensity. The pitch object accepts pitch constants and integer values 0-128; default value is 128. The duration object accepts floating point values in the range 0-1; default value is 0. The intensity object accepts integer values 0-100; default value is 0.

The chord derived type contains one object: a collection of notes.

The staff derived type contains three objects: BPM, beat signature, and a collection of notes and chords. The BPM object accepts integer values 0-240; default value is 0. The beat signature object accepts floating point values in the range 0-1; default value is 0.

The part derived type contains two objects: instrument and a collection of staffs. The instrument object accepts integer values 0-127; default value is 0.

The array derived type contains objects of a given type.

The function derived type returns objects of a given type.

Objects

Objects, lvalues, and rvalues

An object, also referred to as an rvalue, is a named region of storage; an lvalue is an expression referring to an object. An obvious example of an lvalue expression is an identifier with suitable type and storage class.

The names “lvalue” and “rvalue” come from the assignment expression E1 = E2 in which the left operand E1 must be an lvalue expression and the right operand E2 must be an rvalue expression.

Promotion

Associating an object of type int with a float variable is allowable, since m automatically promotes the int to a float if necessary. Promotion occurs in all of the following cases: (1) direct assignment of int to a float and (2) some arithmetic operations between int and float.

If an operation is being performed on two objects of which one operand is an int and the other is a float, the evaluation of the expression is of type float if it cannot be rounded to an int, otherwise the evaluation is of type int.

Demotion is not supported, so assignment of a float to an int is not allowed.

Expressions

Primary Expressions

Primary expressions are identifiers, literals, or expressions in parentheses.

primary-expression:
identifier
literal
(expression)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression.

Postfix Expressions

The operators in postfix expressions group left to right. The result of each is the value of an object.

postfix-expression:
primary-expression
postfix-expression[expression]
postfix-expression(argument-expression-list$_{opt}$)
postfix-expression.identifier
postfix-expression++
postfix-expression--

argument-expression-list:
expression
argument-expression-list , expression

Function Calls

A function call is a postfix expression, called the function designator, followed by parentheses containing a possibly empty, comma-separated list of assignment expressions which constitute the arguments to the function. The function call has the type of the function return type. A function must be defined before it is called. Parameters are passed by value into the function.

The term argument is used for an expression passed by a function call; the term parameter is used for an input object (or its identifier) received by a function definition, or described in a function declaration. The effect of the call is undefined if the number of arguments disagrees with the number of parameters and if the types of the arguments disagree in the definition of the function. The order of evaluation of arguments is unspecified. However, the arguments and the function designator are completely evaluated, including all side effects, before the function is entered. Recursive calls to any function are permitted.

Unary Operators

Expressions with unary operators group right-to-left.

unary-expression:
postfix-expression
unary-operator expression

unary-operator: one of
+ - !

Unary Plus Operator

The operand of the unary + operator must have arithmetic type, and the result is the value of the operand. The type of the result is the type of the operand.

Unary Minus Operator

The operand of the unary - operator must have arithmetic type, and the result is the negative value of the operand. The type of the result is the type of the operand.

Unary Negation Operator

The operand of the ! operator must have bool type, and the result is true if the value of its operand compares equal to false, and false otherwise. The type of the result is bool.

Multiplicative Operators

The multiplicative operators *, /, and % group left-to-right.

multiplicative-expression:
unary-expression
*multiplicative-expression * unary-expression*
multiplicative-expression / unary-expression
multiplicative-expression % unary-expression

The operands of * and / must have arithmetic type; the operands of % must have integral type. The binary * operator denotes multiplication. The binary / operator yields the quotient, and the % operator the remainder, of the division of the first operand by the second; if the second operand is 0, the result is undefined. The type of the result is float if the decimal portion of the result would not be equal to 0 if it were interpreted as a float, otherwise it is int.

Additive Operators

The additive operators + and - group left-to-right. If the operands have arithmetic type, the usual arithmetic conversions are performed.

additive-expression:
multiplicative-expression
additive-expression + multiplicative-expression
additive-expression - multiplicative-expression

The operands of + and – must have arithmetic type. The result of the + operator is the sum of the operands. The result of the - operator is the difference of the operands. The type of the result is float if the decimal portion of the result would not be equal to 0 if it were interpreted as a float, otherwise it is int.

Relational Operators

The relational operators group left-to-right (i.e. a<b<c is parsed as (a<b)<c). Relational expressions evaluate to either true or false Boolean values. The operators < (less), > (greater), <= (less or equal) and >= (greater or equal) all yield true or false Boolean values.

relational-expression:
additive-expression
relational-expression < additive-expression
relational-expression > additive-expression
relational-expression <= additive-expression
relational-expression >= additive-expression

Equality Operators

equality-expression:
relational-expression
equality-expression == relational-expression
equality-expression != relational-expression

The == (equal to) and the != (not equal to) operators are analogous to the relational operators except for their lower precedence. (Thus a<b == c<d is true whenever a<b and c<d have the same truth value.) The operands to && need to be bool. The result is true or false bool value.

Logical AND Operator

logical-AND-expression:
equality-expression
logical-AND-expression && equality-expression

The && operator takes lower precedence to equality operators and groups left-to-right. It is important to note that && guarantees left-to-right evaluation: the first operand is evaluated, including all side effects. It returns true if both its operands compare unequal to false, false otherwise. The operands to && need to be bool. The result is true or false bool value.

Logical OR Operator

logical-OR-expression:
logical-AND-expression
logical-OR-expression || logical-AND-expression

conditional-expression:
logical-OR-expression

The || operator takes lower precedence to the && operator and groups left-to-right. It returns true if either of its operands compare unequal to false, and false otherwise. || guarantees left-to-right evaluation: the first operand is evaluated, including all side effects. If it is equal to false, the value of the expression is false. Otherwise, the right operand is evaluated, and if it is unequal to false, the expression's value is true, otherwise false. The operands need to be bool. The result is true or false bool values.

Assignment Expressions

There are several assignment operators; all group right-to-left.

expression:

conditional-expression
unary-expression assignment-operator expression

assignment-operator: one of

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

All require an lvalue as left operand, and the lvalue must be modifiable: it must not be an array, and must not have an incomplete type, or be a function. The type of an assignment expression is that of its left operand, and the value is the value stored in the left operand after the assignment has taken place. In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue.

An expression of the form E1 op= E2 is equivalent to E1 = E1 op (E2) except that E1 is evaluated only once.

Declarations

Declarations specify the interpretation given to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations that reserve storage are called definitions; if a variable is being declared, then the definition is also called initialization. Declarations have the form:

declaration-statement:

type-specifier init-declarator;

type-specifier:

type
type[expression]

Only one object may be declared per declaration-statement.

Function Declarators

There are no function declarators. Function declarations not allowed, since functions must be defined the same time they are declared.

Array Declarators

An array declarator is a declaration of type

type-specifier init-declarator;

where type-specifier is of the form :

type[expression]

The type may be any type, and the expression within [] must be arithmetic.

Initialization

When an object is declared, its init-declarator may specify an initial value for the identifier being declared. The initializer is preceded by =, and is either an expression, or a list of initializers nested in braces.

init-declarator:
    identifier
    identifier = initializer

initializer:
    expression
    { initializer-list}

initializer-list:
    expression
    initializer-list , expression

The initializer for an int, float, or bool type is a single expression, perhaps in braces. The expression is assigned to the object.

The initializer for a derived type is either an expression of the same type, or a brace-enclosed list of initializers for its members in order. If there are fewer initializers in the list than members of the structure, the trailing members are initialized with default values as described in the types section, or 0 in the case of float and int, or false in the case of bool. There may not be more initializers than members.

Statements

Except as described, statements are executed in sequence. Statements are executed for their effect, and do not have values. They fall into several groups:

statement:
    expression;
    compound-statement
    selection-statement
    iteration-statement
    return $expression_{opt}$*;*
    declaration-statement

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

Compound Statement

compound-statement:
    { statement-list$_{opt}$ }
statement-list:
    statement
    statement-list statement

So that several statements can be used where one is expected, the compound statement (also called "block") is provided. The body of a function definition is a compound statement.

Selection Statements

Selection statements are used for flow control.

selection-statement:
    if(expression) compound-statement elseif-statement-list$_{opt}$
    if(expression) compound-statement elseif-statement-list$_{opt}$ else compound-statement

elseif-statement-list:
    elseif(expression) compound-statement
    elseif-statement-list elseif(expression) compound-statement

Iteration Statements

Iteration statements specify looping.

iteration-statement:
    while(expression) compound-statement
    for(expression$_{opt}$; expression$_{opt}$; expression$_{opt}$) compound-statement

Function Definitions

Function definitions have the form:

function-definition:
    type identifier(parameter-list$_{opt}$) compound-statement

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

A function may return an arithmetic type, bool type, or the derived types note, chord, staff, and part. It may not return a function or an array.

Grammar

literal:
    integer-literal
    floating-literal
    boolean-literal
    pitch-literal

primary-expression:
    identifier
    literal
    (expression)

postfix-expression:
    primary-expression
    postfix-expression[expression]
    postfix-expression(argument-expression-list$_{opt}$)
    postfix-expression.identifier
    postfix-expression++
    postfix-expression--

argument-expression-list:
    expression
    argument-expression-list , expression

unary-expression:
    postfix-expression
    unary-operator expression

unary-operator: one of
    + - !

multiplicative-expression:
    unary-expression
    *multiplicative-expression * unary-expression*
    multiplicative-expression / unary-expression
    multiplicative-expression % unary-expression

additive-expression:
    multiplicative-expression
    additive-expression + multiplicative-expression
    additive-expression - multiplicative-expression

relational-expression:
    additive-expression
    relational-expression < additive-expression
    relational-expression > additive-expression
    relational-expression <= additive-expression
    relational-expression >= additive-expression

equality-expression:
    relational-expression
    equality-expression == relational-expression
    equality-expression != relational-expression

logical-AND-expression:
    equality-expression
    logical-AND-expression && equality-expression

logical-OR-expression:
    logical-AND-expression
    logical-OR-expression || logical-AND-expression

conditional-expression:
    logical-OR-expression

expression:
    conditional-expression
    unary-expression assignment-operator expression

assignment-operator: one of
    *= *= /= %= += -=*

declaration-statement:
    type-specifier init-declarator;

type-specifier:
    type
    type[expression]

init-declarator:
    identifier
    identifier = initializer

initializer:
    expression
    { initializer-list}

initializer-list:
expression
initializer-list , expression

statement:
expression;
compound-statement
selection-statement
iteration-statement
return $expression_{opt}$*;*
declaration-statement

compound-statement:
{ $statement\text{-}list_{opt}$ *}*

statement-list:
statement
statement-list statement

selection-statement:
if(expression) compound-statement $elseif\text{-}statement\text{-}list_{opt}$
if(expression) compound-statement $elseif\text{-}statement\text{-}list_{opt}$ *else compound-statement*

elseif-statement-list:
elseif(expression) compound-statement
elseif-statement-list elseif(expression) compound-statement

iteration-statement:
while(expression) compound-statement
for($expression_{opt}$*;* $expression_{opt}$*;* $expression_{opt}$*) compound-statement*

function-definition:
type identifier($parameter\text{-}list_{opt}$*) compound-statement*

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

Appendix

scanner.mll

```
{ open Parser } (* Get the token types *)
rule token = parse
      [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
      | "/*" { comment lexbuf } (* Comments *)
      | "//" { singlecomment lexbuf }
      | '(' { LPAREN }
      | ')' { RPAREN } (* punctuation *)
      | '{' { LBRACE }
      | '}' { RBRACE }
      | '[' { LBRACKET }
      | ']' { RBRACKET }
      | ';' { SEMI }
      | ',' { COMMA }
      | '.' { DOT }
      | '+' { PLUS } (* started here *)
      | '-' { MINUS }
      | '*' { TIMES }
      | '/' { DIVIDE }
      | '%' { MOD }
      | "+=" { PLUSEQ }
      | "-=" { MINUSEQ }
      | "*=" { TIMESEQ }
      | "/=" { DIVIDEEQ }
      | "%=" { MODEQ }
      | '=' { ASSIGN }
      | '!' { NOT }
      | "++" { PLUSPLUS }
      | "--" { MINUSMINUS }
      | "==" { EQ }
      | "!=" { NEQ }
      | '<' { LT }
      | "<=" { LEQ }
      | ">" { GT }
      | ">=" { GEQ }
      | "&&" { AND }
      | "||" { OR }
      | "if" { IF }
      | "else" { ELSE }
      | "elseif" { ELSEIF }
      | "for" { FOR }
      | "while" { WHILE }
      | "return" { RETURN }
      | "void" { DATATYPE("void") }
      | "int" { DATATYPE("int") }
      | "float" { DATATYPE("float") }
      | "bool" { DATATYPE("bool") }
      | "note" { DATATYPE("note") }
      | "chord" { DATATYPE("chord") }
      | "staff" { DATATYPE("staff") }
      | "part" { DATATYPE("part") }
      | "true"|"false" as boollit
```

```
      { BOOLLITERAL(bool_of_string boollit) }
| (['a'-'g' 'A'-'G']['s' 'f' 'S' 'F']?['0'-'9'])|('r'|'R')
      as pitchlit { PITCHLITERAL(pitchlit) }
| eof { EOF } (* Endoffile *)
| ['0'-'9']+ as lxm
      { INTLITERAL(int_of_string lxm) } (* integers *)
| ((['0'-'9']+'.'['0'-'9']*)) | (('.'['0'-'9']+))
      as floatlit { FLOATLITERAL(float_of_string floatlit) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
      as lxm { ID(lxm) }
| _ as char { raise (Failure("illegal character " ^
Char.escaped char)) }

and comment = parse
"*/" { token lexbuf } (* Endofcomment *)
| _ { comment lexbuf } (* Eat everything else *)

and singlecomment = parse
['\t' '\r' '\n'] {singlecomment lexbuf }
```

parser.mly

```
%{ open Ast %}

%token LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET
%token SEMI COMMA DOT
%token PLUS MINUS TIMES DIVIDE MOD PLUSPLUS MINUSMINUS
%token PLUSEQ MINUSEQ TIMESEQ DIVIDEEQ MODEQ
%token EQ NEQ LT LEQ GT GEQ AND NOT OR ASSIGN
%token IF ELSE ELSEIF FOR WHILE RETURN
%token INT VOID FLOAT BOOL NOTE CHORD STAFF PART
%token <int> INTLITERAL
%token <float> FLOATLITERAL
%token <bool> BOOLLITERAL
%token <string> ID
%token <string> DATATYPE
%token <string> PITCHLITERAL
%token EOF


%nonassoc ELSE
%nonassoc NOELSE
%nonassoc ELSEIF
%nonassoc NOACTUALS
%nonassoc LPAREN
%left PLUSEQ MINUSEQ
%left TIMESEQ DIVIDEEQ MODEQ
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%right NOT
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right UPLUS UMINUS
%left PLUSPLUS MINUSMINUS
%start program
%type <Ast.program> program

%%

program:
/* nothing */ { [], [] }
| program vdecl { ($2 :: fst $1), snd $1 }
| program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
      DATATYPE ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list
RBRACE
            { {   fname = $2;
                  rettype = $1;
                  formals = $4;
                  locals = List.rev $7;
                  body = List.rev $8 } }
      | VOID ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
```

```
            { {   fname = $2;
                  rettype = "void";
                  formals = $4;
                  locals = List.rev $7;
                  body = List.rev $8 } }

formals_opt:
      /* nothing */ { [] }
      | formal_list { List.rev($1) }

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

p_decl:
      DATATYPE ID
            { {   paramname = $2;
                  paramtype = $1 } }

vdecl_list:
      /* nothing */    { [] }
      | vdecl_list vdecl { $2 :: $1 }

vdecl:
      DATATYPE ID SEMI { { varname = $2; vartype = $1; value = 0 } }
      | DATATYPE ID ASSIGN expr SEMI { { varname = $2; vartype = $1; value =
$4 } }
      | DATATYPE LBRACKET expr RBRACKET ID SEMI { MakeArray($5, []) }
      | DATATYPE LBRACKET expr RBRACKET ID ASSIGN LBRACE array_list RBRACE
SEMI{ MakeArray($5, List.rev($8)) }

array_list:
      expr { [$1] }
      | array_list COMMA expr { $3 :: $1}

stmt_list:
      /* nothing */ { [] }
      | stmt_list stmt { $2 :: $1 }

stmt:
      expr SEMI { Expr($1) }
      | RETURN expr_opt SEMI { Return($2) }
      | LBRACE stmt_list RBRACE { Block(List.rev $2) }
      | IF LPAREN expr RPAREN stmt elseif_stmt %prec NOELSE { If($3, $5, $6,
Block([])) }
      | IF LPAREN expr RPAREN stmt elseif_stmt ELSE stmt { If($3, $5, $6, $8)
}
      | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt { For($3,
$5, $7, $9) }
      | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

elseif_stmt:
      /* nothing */ { [] }
      | elseif_stmt ELSEIF LPAREN expr RPAREN stmt { {elseif_exp = $4;
elseif_action = $6 } }

expr_opt:
```

```
/* nothing */ { Noexpr }
| expr { $1 }

expr:
      INTLITERAL { Intliteral($1) }
      | FLOATLITERAL { Floatliteral($1) }
      | BOOLLITERAL { Boolliteral ($1) }
      | PITCHLITERAL { Pitchliteral ($1) }
      | ID %prec NOACTUALS{ Id($1) }
      | 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 EQ expr { Binop($1, Equal, $3) }
      | expr NEQ 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 AND expr { Binop($1, And, $3) }
      | expr OR expr { Binop($1, Or, $3) }
      | ID ASSIGN expr { Assign($1, $3) }
      | ID PLUSEQ expr { Pluseq($1, $3) }
      | ID MINUSEQ expr { Minuseq($1, $3) }
      | ID TIMESEQ expr { Timeseq($1, $3) }
      | ID DIVIDEEQ expr { Divideeq($1, $3) }
      | ID MODEQ expr { Modeq($1, $3) }
      | NOT expr { Not($2) }
      | MINUS expr %prec UMINUS { $2 }
      | PLUS expr %prec UPLUS { $2 }
      | expr PLUSPLUS { Plusplus($1) }
      | expr MINUSMINUS { Minusminus($1) }
      | ID LBRACKET expr RBRACKET { Getelement($1, $3) }
      | ID DOT ID { Memberaccess($1, $3) }
      | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
      | LPAREN expr RPAREN { $2 }

actuals_opt:
/* nothing */ { [] }
| actuals_list { List.rev $1 }

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