MIDILC: A MIDI Language Compiler for Programmatic Music Composition

Alex “Akiva” Bamberger
Benjamin Mann
Frederic Lowenthal
Ye Liu

Contents

Chapter 1

Introduction

The language, hereafter referred to as MIDILC (pronounced MIDDLE C, standing for MIDI Language Compiler), allows programmers to compose music. It compiles into MIDI format and has syntax that is similar to Java, changing the basic primitives and the meaning of various operators (Fig 1.1).

The langauge is dynamically typed, but introduces preset types for the programmer most comfortable with the syntax of C or Java. Types must be used upon variable declaration, but are left optional for function declarations and arguments. Each type can be safely cast up in the following order: `Number` $\rightarrow$ `Note` $\rightarrow$ `Chord` $\rightarrow$ `Sequence`. The standard library, written in the language itself, supports major and minor chords, arpeggios, repetition, and other such basic and often used concepts. Note durations are specified in terms of whole notes (w), halves (h), quarters (q), eigths (e), and sixteenths (s). Sequences can either be appended to, which advances the "current time" by however long the appended sequence is, or else something can be inserted into a sequence at a given offset using subscripting. Functions are specified in the same way as in C.

Sequences and Chords can be output to the intermediate representation (IR) of CSV format using the play() function. In order to actually write the MIDI files, the CSV is fed to a Java program that interprets the CSV using the javax.sound.MIDI package.

Composing music on a computer is often done using GUIs that allow the user to drag and drop notes or using instrument inputs. This lets the musician hear his compositions as he is creating them, and often gives the musician a simple MP3 or MIDI ouput. As computer scientists, the MIDILC team finds such methods tedious, extraneuous and requiring too much natural music talent. MIDILC appeals to the virtues of any great programmer: laziness,

`Number`	a value between -2^{31} and $2^{31}-1$
`Note`	a musical note with pitch and duration
`Chord`	stores notes with equal durations and start times, represented by an integer list
`Sequence`	a sequence of notes and chords, represented by a list of integer lists

Figure 1.1: Types in the MIDILC language.

hubris, and impatience [1] and propose a language for those wishing to turn their quantitative skills into beautiful music.

MIDILC also attempts to redress other issues of regular music composition. Songs often have frequent recurring themes. Manually reusing these themes requires precision and dedication. If pieces of a song could be manipulated automatically for reuse and slight modification, song production speed could increase dramatically. The MIDILC language allows programmers to algorithmically generate notes, chords, and sequences of notes and chords by writing functions. This allows for writing interesting compositions that minimize time spent rewriting basic MIDI manipulation routines and implementing primitive musical constructs. The language works optimally with compositions that make consistent use of simple motifs as it encourages reuse of simple mathematical operations on notes and chords.

MIDILC is tailored for crafting melody sequences. For a sequence containing a series of whole notes, one could easily manipulate the notes in the melody to create sequences of counterpoints to the melody. Using MIDILC, a `major()` and `minor()` function are easy to craft and chords simple to arpeggiate to make counterpoint. MIDILC allows for the simple concatenation and composition of sequences, and so complicated sequences could be easily made from simple starting blocks.

[1]Wall, Larry. *Programming Perl*, O'Reilly 2000.

Chapter 2

Language Tutorial

MIDILC was designed with the programmer in mind, but uses all the musical notation familiar to the musician. The lanuage is robust enough to fully support the idiomatic construction of Notes and Chords, as well as set tempo and note duration using `Note` and `Number` literals.

2.1 A Simple Melody

2.2 Twinkle Twinkle Little Star

Chapter 3

Language Reference Manual

MIDILC is a C-like language that makes it simpler to algorithmically generate music. It simplifies MIDI music creation by allowing programmers to specify song information in musical terms and write functions that process existing musical information. By building off of simpler musical functions, such as arpeggios and chords, complex musical compositions can easily be programmed.

To eliminate the programming complexities from the MIDILC language, it has limited scope and data management capabilities. MIDILC can be used following an imperative or functional paradigm and reduces hassle for the programmer by forcing static scope.

It compiles into MIDI files that can then be played in any standard media player.

3.1 Lexical Conventions

3.1.1 Tokens

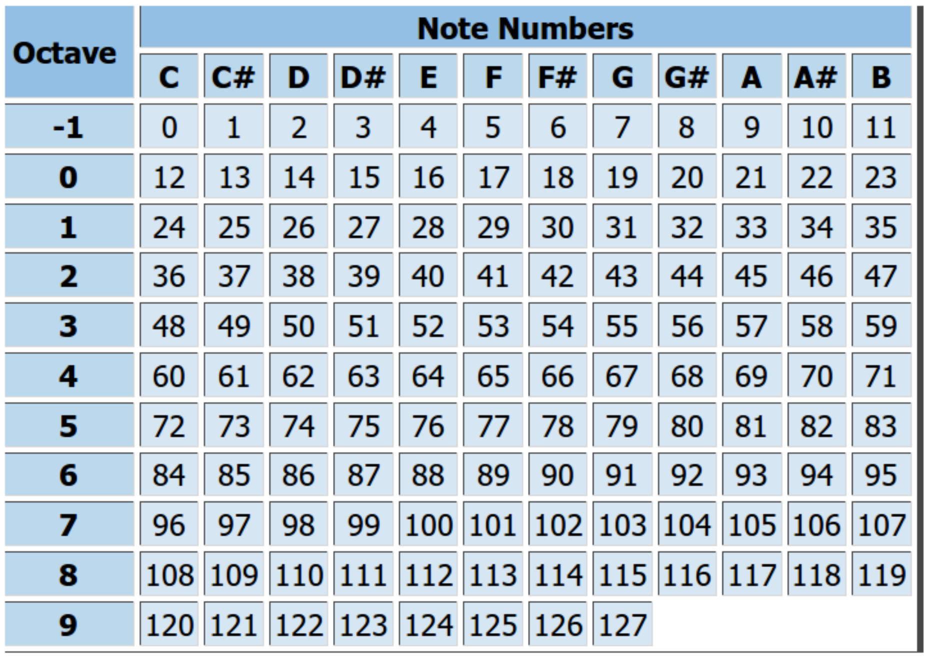
Tokens consist of identifiers, keywords, constants, operators, and separators. As with C, MIDILC is a free-form language and all white space characters are ignored (with the exception of separating tokens), as braces are used to identify the start and end of code blocks and semicolons are used to end statements.

3.1.2 Comments

/* and */ are used to indicate a block of comments (C-style comments). There are no C++-style comments in MIDILC.

3.1.3 Identifiers

These are sequences of letters, digits, and underscores, starting with a letter or underscore. Identifiers cannot be of the format `[A-G R][b#]?[0-9]?[w h q e s]?`, as these are reserved for `Note` literals.



Octave	Note Numbers											
	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
-1	0	1	2	3	4	5	6	7	8	9	10	11
0	12	13	14	15	16	17	18	19	20	21	22	23
1	24	25	26	27	28	29	30	31	32	33	34	35
2	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59
4	60	61	62	63	64	65	66	67	68	69	70	71
5	72	73	74	75	76	77	78	79	80	81	82	83
6	84	85	86	87	88	89	90	91	92	93	94	95
7	96	97	98	99	100	101	102	103	104	105	106	107
8	108	109	110	111	112	113	114	115	116	117	118	119
9	120	121	122	123	124	125	126	127				

Figure 3.1: The correspondence of notes and pitches in MIDI.

3.1.4 Keywords

MIDILC has very few keywords; these include the following:

Types	Control
`Number`	`return`
`Note`	`continue`
`Chord`	`break`
`Sequence`	`if`
`Void`	`else`
	`while`
	`for`

3.1.5 Constants/Literals

MIDILC allows a user to construct `Notes` using pitch and duration (as `Number` types), or using a set of `Note` literals, specified by the note letter, accidental (if any), MIDI octave, and a letter that indicates the notes duration (optional, and defaulting to a quarter note). Duration can be specified by w (for whole note), h (for half note), q (for quarter note), e (for eigth note), and s (for sixteenth note). Rests are indicated by using R instead of a note. Any `Note` object constructed using a pitch with illegal properties will result in an error. `Chord`s can easily be expressed using built-in chord generation function calls on `Note` literals. In addition, `Number` literals also exist (integral numbers limited to signed 32-bit range). MIDILC does not have floating-point literals. Note that literals look like the following: `Ab7`, `C4s`, `G5h`

Pitches and `Number` literals have the correspondence shown in figure 3.1.

3.2 Meaning of Identifiers

Identifiers in MIDILC have the following attributes: scope, name space, linkage, and storage duration, as detailed in Section 4.1. Since static scope is handled automatically, there are no storage class specifiers in MIDILC.

3.2.1 Disambiguating Names

Scope

The scope of of an identifier is defined as the region of a program within which it is visible, and begins when it is declared. In MIDILC, all identifiers are globally scoped, and are therefore visible to all blocks within a program unless hidden in another scope. This is due to the fact that the language automatically handles static identifiers.

Name Space

All the identifiers in MIDILC are categorized as ordinary identifiers. These include user-defined type names, object names, and function names.

Linkage of Identifiers

Identifiers in MIDILC may be linked across different files of the same program, but the identifier name must be unique in all files. Furthermore, the compiler will generate a compile time error about the identifier if there is a conflict.

Storage Duration

Storage duration denotes the lifetime of an object. All objects in MIDILC are static, and have static storage duration. The initialization of these objects occurs only once, prior to any reference.

3.2.2 Object Types

The MIDILC language supports two types of objects: numbers and musical notations. Objects are dynamically typed. Typing of an identifier is determined on assignment.

Number type

The only supported numerical type is `Number`, which has a size of 32 bits and ranges from -2^{31} to $2^{31} - 1$. This is also the underlying type for all fields within the musical types.

Musical types

`Note`, `Chord`, and `Sequence` are all of the musical types supported by MIDILC. `Note` literals are made up of strings consisting of integers and characters in sequences that match the following regular expression: `[A-G R][b #]?[0-9]?[w h q e s]?` As these types are not stored directly internally, their sizes are not exact. As a general rule, for non-empty objects, `Number` ¡ `Note` ¡ `Chord` ¡ `Sequence` in terms of their relative sizes.

Note type `Note` type has the following attributes: pitch and duration. Pitch refers to the frequency of the note (Fig 3.1), and duration is specified as a type of note: whole, half, quarter, eighth, or sixteenth. `Note` literals with the pitch indicated as R instead of A-G are rests (numerically represented as -1).

Chord type `Chord` type has the following attributes: duration and length. Duration is a `Number` type that specifies a type of note: whole, half, quarter, eighth, or sixteenth. All `Note` literals within the same `Chord` must have the same duration. This property can be specified as number of sixteenths. Length of the `Chord` refers to the number of `Note` literals in the `Chord`. `Chord` literals can be constructed with the following syntax: `new_chord(Note n1,  Note n2, Note n3);`

Sequence type `Sequence` type has the following attributes: current, beginning, and length. Each is of type `Number`. Beginning denotes the starting point of the `Sequence`, while current denotes the current time (from beginning) where a new note may be inserted. The length of the `Sequence` refers to the number of `Note` literals or `Chord` objects in the `Sequence`. Sequences can be constructed with the following command: `new_sequence()`.

Derived types

`Note`, `Chord` and `Sequence` objects can be derived. `Note` can be derived from a `Number` (specifying the pitch of the `Note`, with duration of a quarter note). A `Chord` be derived from a collection of `Note` objects. A `Sequence` can be derived from a collection of `Note` or `Chord` objects.

Void type

The `Void` type specifies an empty set of return values. It never refers to an object.

3.2.3 Objects and lvalues

An object is a manipulable region of storage. An lvalue is an expression referring to an object, for example, an identifier. Assignment causes the region of memory specified by the lvalue to be replaced or modified according to the value on the right side of the assignment. For instance, if `a = b` and `c = b`, if `b` is changed, `a` and `b` will remain unchanged.

3.3 Operator Conversions

Due to the nature of the primitive types, very few conversions are supported in MIDILC. It is possible to cast from `Number` to `Note`, from `Note` to `Chord`, from `Note` to `Sequence`, and from `Chord` to `Sequence`. Casts cannot be done in the opposite direction.

assignment-expression	`note = Ab7`
operation-expression	`Ab7 .+ 4`

Figure 3.2: Examples of expressions

3.3.1 Conversions of Number and Note

`Number` objects can be converted into `Note` objects as a note with the pitch represented as an integer in MIDI notation. `Note` objects cannot be converted to `Number` objects. The new object has a default duration of quarter note.

3.3.2 Conversions of Note and Chord

`Note` objects can be converted into `Chord` objects as one-note chords. `Chord` objects cannot be converted into `Note` objects, as this is a narrowing conversion. The resulting `Chord` has the same duration as the `Note` used to construct it.

3.3.3 Conversions of Note and Sequence

`Note` objects can be converted into `Sequence` objects as a sequence that contains a single note. `Sequence` objects cannot be converted into `Note` objects, as this is a narrowing conversion, even if the `Sequence` contains only a single `Note`.

3.3.4 Conversions of Chord and Sequence

`Chord` objects can be converted into `Sequence` objects as a sequence that contains a single chord. `Sequence` objects cannot be converted into `Chord` objects, as this is a narrowing conversion, even if the `Sequence` contains only a single `Chord`.

3.4 Expressions and Operators

In MIDILC, expressions include one or more operators and a number of operands that follow certain associativity rules. Operators may change the value of an operand or leave it alone. Expressions (Fig 3.4) can be used for assignment or other operations. Associativity of these assignments can be overrridden by parentheses (Fig 3.4). Associativity of operators followed the table shown in figure 3.4.

3.4.1 Primary Expressions

Identifiers

An lvalue or function designator, discussed in part 2.

Constants

An object of constant value, discussed in part 2.

Expression	Result	Explanation
`C7 .+ 4`	`E7`	`Note` with E7 pitch
`3 + 2 * 4`	`11`	Regular assignment order (multiplication has tightest binding, then addition)
`(3 + 2) * 4`	`20`	Parentheses change order of operations
`note = C7;` `new_chord(note,` `note .+ 4,` `note .+ 7);`	`Chord` of `(C7, E7, G7)`	Addition operator has tightest binding, followed by the assignment operator

Figure 3.3: Associativity overridden by use of parentheses.

Tokens (From High to Low Priority)	Operators	Class	Associativity
Identifiers, constants, parenthesized expression	Primary expression	Primary	
`() [] .`	Function calls, subscripting, direct selection	Postfix	L-R
`id as Type`	Cast	Binary	L-R
`* /`	Times/Divide	Binary	L-R
`.+ .-`	DotPlus/DotMinus	Binary	L-R
`+ -`	Add/Minus	Binary	L-R
`== !=`	Equality comparisons	Binary	L-R
`< <= >= >`	Relational Comparisons	Binary	L-R
`&&`	Logical and	Binary	L-R
`||`	Logical or	Binary	L-R
`=`	Assignment	Binary	R-L
`,`	Comma	Binary	L-R

Figure 3.4: Order of operations for built in operators

Parenthesized Expressions

Parenthesized expressions allow a user to change the order of operations. They are executed before the operations and can be used as part of a larger expression (Fig 3.4).

3.4.2 Postfix

Postfix calls are made as follows:

Function call	`Chord c; c =  new_chord(Ab6, Ab7, C4);`
Subscripting	`Note n; n = c[0];`
Direct selection	`Number i; i = n.pitch;`

Function calls

The syntax of a function call is as follows:

$postfix\text{-}expression \rightarrow (argument\text{-}expression\text{-}list_{opt})$
$argument\text{-}expression\text{-}list \rightarrow argument\text{-}expression$
$\rightarrow argument\text{-}expression\text{-}list, argument\text{-}expression$

An argument expression list may either be a single argument or a list of arguments. All functions are allowed to be recursive. Each function must be declared before it is called. With that in mind, certain casts are made by the runtime compiler to match arguments. A `Number` may be cast to a `Note`, `Chord`, or `Sequence`, for example. A function may only take the a parameter of type Void. For functions like this, a function call may include no parameters.

Subscripting

Certain objects may be acted upon by the subscripting operation. For example, a `Chord` object may be acted upon by a subscript to select a particular note in the chord. Similarly, a `Sequence` object may be acted upon to select a `Chord` at any particular moment in time. For a `Chord` object, the index of the subscript reflects the order that a `Note` was added. For a `Sequence` object, the index subscript indicates the order that Chords were inserted in. The subscripting operator allows both retrieval and mutation of elements in those objects that support it. There is no implicit casting for subscription.

Direct Selection

Used to change pitch and duration in objects of type `Note`, `Chord`, or `Sequence`. Pitch and duration are treated as objects of type `Number` with the pitch affected (either positively or negatively) by the successor operand. For example, `C7.pitch = C7.pitch + 1` will result in C#7. Similarly for duration: `C7.duration = C7.duration + 1` will result in C7 with a duration of a 1/16th note greater. Direct selection can be done for the following parameters on the following objects: `Note`: pitch, duration `Chord`: duration, length `Sequence`: current, length

3.4.3 Unary Operations

Casting

Syntax of casting is as follows:

cast-expression → *unary-expression*
→ (*cast-expression* as *type-name*)

Casting allows a user to explicitly change the Type of an object, according to the order established in Musical Types. Implicitly casting will take place during a function call or in the use of a binary operator between two objects of different type. If, however, we wanted to craft two notes, and then append one to another in a chord, we would need to do the following: `s = (((note1 as Chord) as Sequence) + note2)` This would allow us to use the + operator of Sequences instead of the + operator of Notes.

3.4.4 Binary Operations

Mult/Divide

DotAdd/DotMinus

Add/Subtract

Used to add or subtract two `Number` objects. When applied to objects of type `Note`, `Chord`, or `Sequence`, results in a `Sequence` object with given elements concatenated. If two or more objects of different type are concatenated, the element of highest cast determines the cast. That is, a `Note` added to a `Sequence` would return a new `Sequence` with the given note appended as a degenerate `Chord` to the end.

Syntax is as follows:

add-expression → *cast-expression*
→ *add-expression* + *cast-expression*
→ *add-expression* − *cast-expression*

Relational comparisons

Yields a `Number` result (1 if true, 0 if false). Allows for comparison between objects (casting is done in one direction).

relational-expression → *add-expression*
→ *relational-expression* < *add-expression*
→ *relational-expression* > *add-expression*
→ *relational-expression* <= *add-expression*
→ *relational-expression* >= *add-expression*

Equality comparisons

Compares two values for equality. MIDILC uses the number 0 to denote false and all values other than 0 to denote truth. Equality follows the following rules: Two `Number` objects are equal if they evaluate to the same value Two `Note` objects are equal if they have the same pitch and duration Two `Chord` objects are equal if they have the same notes and the same duration Two `Sequence` objects are equal if they have the same chords in the same order

equality-expression → *relational-expression*
→ *equality-expression* == *relational-expression*
→ *equality-expression* != *relational-expression*

Logical and

Performs a logical "and" on two expressions. Returns 0 if the left expression evaluates to 0. Otherwise, evaluates right expression. If true, returns 1; if false, 0. Syntax:

logical-AND-expression → *logical-OR-expression*
→ *logical-AND-expression* `&&` *logical-OR-expression*

This is done with lazy evaluation.

Logical or

Performs a logical "or" on two expressions. Returns 1 if ever the left expression evaluates to 1. Otherwise, evaluates right expression. If true, 1; if false, 0. Syntax:

logical-OR-expression → *logical-AND-expression*
→ *logical-OR-expression* `||` *logical-AND-expression*

This is again an example of MIDILCs power to perform lazy evaluation.

Assignment

Right associative. The expression on the right is evaluated and then used to set the lvalue. The rvalue must have the same type as the lvalue; no casting is implicitly done.

Comma

Separates elements in a list (such as parameters in a function or `Note` literals in a `Chord`). Example of `Chord` constructor: `Chord myChord; myChord = new_chord(C4, E4);`

3.5 Declarations

Declarations specify the interpretation given to a set of identifiers.

direct-declarator → *type-specifier declarator*
init-declarator → *type-specifier declarator* = *initializer*

Only a single declarator can be declared at once. Declarators must be preceded by the type of the identifier. At most one declaration of the identifier can appear in the same scope and name space.

3.5.1 Storage class specifiers

Static scope is handled automatically because functions have access to any identifiers not declared in their scope. No storage class specifiers are available.

3.5.2 Type specifiers

Type specifiers listed below. Syntax as follows:

type-specifier	`Void`
	`Number`
	`Note`
	`Chord`
	`Sequence`

3.5.3 Type qualifiers

Types cannot be declared mutable or immutable by the programmer. All types are mutable

3.5.4 Function Declarators

There are no function prototypes (all function declarations are definitions). The syntax for function declarators is shown below:

direct-declarator → (*identifier-list*$_{opt}$) *body*
identifier-list → *identifier-list*, *direct-declarator*

For example, *T D* (*identifier-list*$_{opt}$) creates a function with identifier D and return type T with the specified parameters. An identifier list declares the types of and identifiers for the formal parameters of a function.

Function declarators do not support variable additional arguments.

If the type of any parameter declared in the identifier list is other than that which would be derived using the default argument promotions, an error is posted. Otherwise, a warning is posted and the function prototype remains in scope.

When a function is invoked for which a function is defined, no attempt is made to convert each actual parameter to the type of the corresponding formal parameter specified in the function prototype. Instead an error is thrown.

The following is an example of a function definition: `Chord transposeChord(Chord oldChord, Note newKey) { ... }` This declares a function `transposeChord()` which returns a `Chord` and has two parameters: a `Chord` and a `Note`.

3.5.5 Initialization

A declaration of a type can specify an initial value for the identifier after being declared. The initializer is preceded by = and consists of an expression.

initializer → *assignment-expression*

Variables that are not explicitly initialized may cause a null pointer exception during compilation. When an initializer applies to a literal, it consists of a single expression, perhaps in parentheses. The initial value of the object is taken from the expression. Type conversion is only attempted with an explicit cast.

Examples of initialization

```
Note root;
```

```
Chord notes;
Sequence gProgression;
/*Initializes root with a note literal.*/
root = C3q;

/*Initializes notes with a chord literal*/
notes = new_chord( root, root .+ 4, root .+ 7 );

/*Initializes gProgression with the result of the function call.*/
gProgression = oneFourFiveProg( G7q );
```

3.6 Statements

A statment is a complete instruction to the midi compiler. Except as indicated, statements are executed in sequence. Statements have the following form:

statement → *expression-statement*
→ *selection-statement*
→ *iteration-statement*
→ *jump-statement*

3.6.1 Expression statement

Most statements are expression statements, which have the following form:

expression-statement → *expression*;

Usually expression statements are expressions evaluated for their side effects such as assignments or function calls.

3.6.2 Compound statement or block

A compound statement (or block) groups a set of statements into a syntactic unit. The set can have its own declarations and initializers, and as the following form:

compound-statement → {*declaration-list statement-list*$_{opt}$}
declaration-list → *declaration*
→ *declaration-list declaration*
statement-list → *statement*
→ *statement-list statement*

Declarations within compound statements have block scope. If any of the identifiers in the declaration list were previously declared, the outer declaration is hidden for the duration of the block, after which it resumes its force. Function declarations can only be defined at the outermost scope.

3.6.3 Selection statements

Selection statements include the if and else statements and have the following form:

selection-statement $\rightarrow if$ (*expression*) *statement*
$\rightarrow if$ (*expression*) *statement else statement*

Selection statements choose one of a set of statements to execute, based on the evaluation of the expression. The expression is referred to as the controlling expression.

if statement

The controlling expression of an `if` statement must have `Number` type. For both forms of the `if` statement, the first statement is executed if the controlling expression evaluates to nonzero. For the second form, the second statement is executed if the controlling expression evaluates to zero. An else clause that follows multiple sequential else-less if statements is associated with the most recent `if` statement in the same block (that is, not in an enclosed block).

3.6.4 Iteration statements

Iteration statements execute the attached statement (called the body) repeatedly until the controlling expression evaluates to zero. In the for statement, the second expression is the controlling expression. The format is as follows:

iteration-statement $\rightarrow$ *while*(*expression*) *statement*
$\rightarrow for$ (*expression*; *expression* ; *expression*) *statement*

The controlling expression must have `Number` type.

while statement

The controlling expression of a `while` statement is evaluated before each execution of the body.

for statement

The for statement has the form specified above. The first expression specifies the initialization for the loop. The second expression is the controlling expression, which is evaluated before each iteration. The third expression often specifies incrementation. It is evaluated after each iteration. It is equivalent to the following:

expression-1 $\rightarrow$ *while* (*expression*-2) {*statement expression*-3}

One exception exists, however. If a `continue` statement is encountered, *expression*-3 of the `for` statement is executed prior to the next iteration.

3.6.5 Jump statements:

jump-statement $\rightarrow$`continue`
$\rightarrow$`break`
$\rightarrow$`return` $expression_{opt}$;

continue statement

The `continue` statement can appear only in the body of an iteration statement. It causes control to pass to the loop-continuation portion of the smallest enclosing `while` or `for` statement; that is, to the end of the loop.

break statement

The `break` statement can appear only in the body of an iteration statement or code attached to a switch statement. It transfers control to the statement immediately following the smallest enclosing iteration, terminating its execution.

return statement

A function returns to its caller by means of the `return` statement. The value of the expression is returned to the caller as the value of the function call expression. The `return` statement cannot have an expression if the type of the current function is `Void`. If the end of a function is reached before the execution of an explicit `return`, an implicit `return` (with no expression) is executed. If the value of the function call expression is used when none is returned, the behavior is undefined.

3.7 Built-In Functions

`Void play(Sequence s)`	Instructs compiler to write a `Sequence` to the MIDI file.
`Void set_tempo(Number n)`	Sets the tempo of the file to `Number`.
`Void set_instrument("instrument")`	Sets the instrument for the song
`Sequence new_sequence()`	Initializes an empty `Sequence`.
`Chord new_chord(Note n1, Note n2, ...)`	Initializes a `Chord` object.

Chapter 4

Project Plan

Chapter 5

Architecture & Design

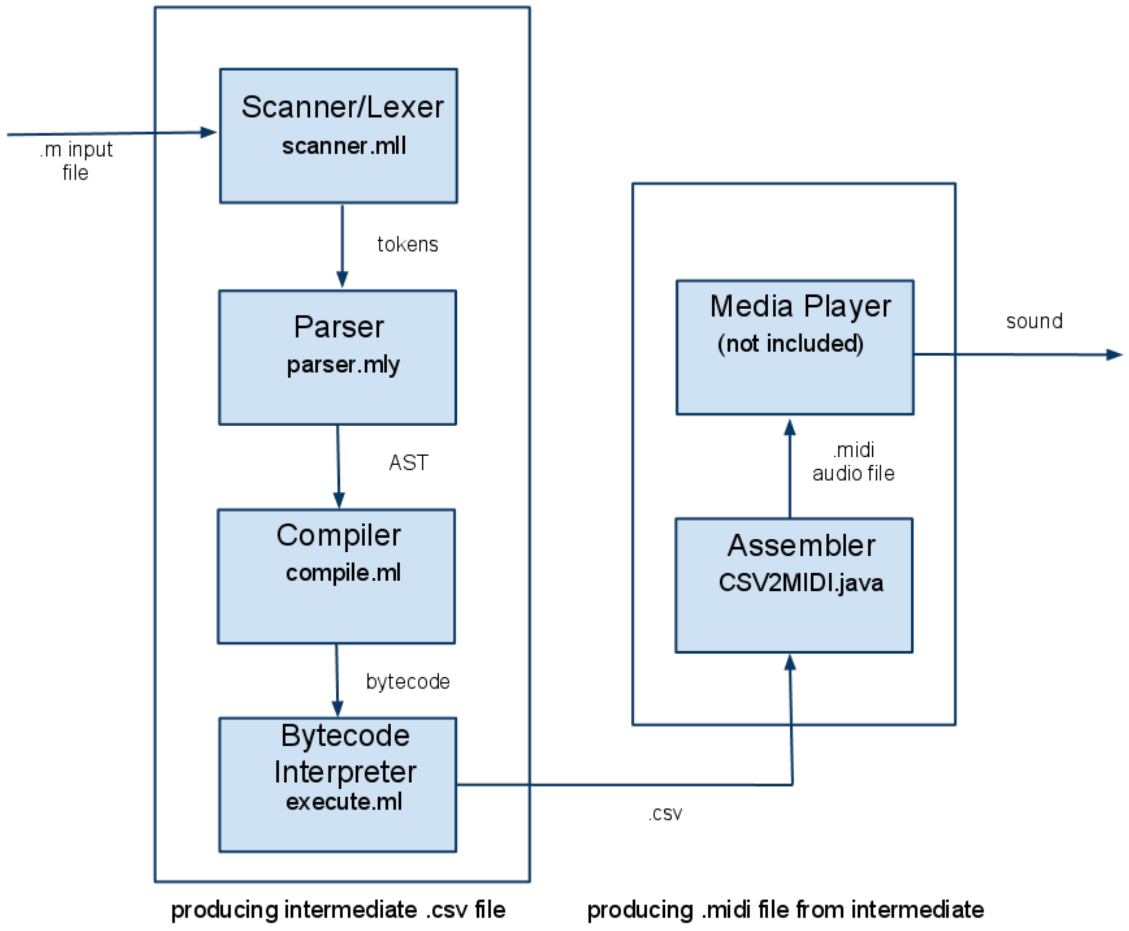


Figure 5.1: A simplified diagram displaying the conversion of source code to .midi music.

Chapter 6

Test Plan

Chapter 7

Lessons Learned

Chapter 8

Appendix