Marmalade

A Music Creation Language

Savvas Petridis (sdp2137) Language Guru
Uzo Amuzie (ua2144) Tester
Cathy Jin (ckj2111) System Architect
Raphael Norwitz (rsn2117) Manager
Table of Contents:

Introduction .......................................................................................................................... 2

- Marmalade White Paper

Language Tutorial ............................................................................................................... 3

- Running the Compiler

Language Reference Manual .............................................................................................. 4

  - Lexical Conventions
  - Data Types
  - Lists
  - Operator Usage
  - Reserved Words
  - Other Expressions
  - Funktions

Project Plan ....................................................................................................................... 18

  - Process
  - Roles & Responsibilities

Architectural Design ......................................................................................................... 20

  - Block Diagram
  - Interface Between Components

Test Plan ............................................................................................................................ 22

  - Sample Marmalade Programs
  - Translator Test Suite

Lessons Learned ............................................................................................................... 23

  - Description
  - Advice for Future Teams

Appendix ............................................................................................................................ 27

  - Instrument List
  - Code Listing
Introduction:

--

Marmalade is a very readable musical programming language. Similar languages and libraries use clunky object-oriented syntax and require language-specific knowledge, thereby preventing users from composing right away. We wanted to create a tool that would minimize the distance from one's conception of a musical idea to actually writing and playing it. So we tossed aside classes and anything else that seemed unnecessary and left the bare bones: integers, notes, time signatures, instruments, functions, lists, and lists of lists.

Why Marmalade?

This list structure, along with intuitive operators, simple function syntax, clear control flow, and a spartan standard library, gives the composer the freedom to write what they want without compromise. Marmalade breaks a musical piece into four discrete building blocks: notes, measures, phrases, and songs. The user must define and combine notes to form measures, then combine these measures into phrases to be played simultaneously. This bottom-up approach encourages the user to think about his composition as discrete pieces to be arranged and rearranged.

Marmalade can suit any user from those who only seek to use Marmalade's core features and create songs, to those who'd like to create as well as experiment with their pieces by defining complicated functions to transform them. Perhaps the most enticing feature of Marmalade is its low learning curve. One can easily define a series of measures, turn them into phrases and combine them into a song. While doing so, one can define and redefine the time signature at the measure level, instruments at the phrase level, and tempo at the song level to tweak the song to his or her particular liking. And in a few minutes a song has been created, played, and outputted as a midi file!
Language Tutorial:

Running the compiler and executing a program:

Steps to run the compiler:

1. $ make
2. $./make_java.sh
3. $./marmac name.marm executable_name
4. $./executable_name

It's that easy!

marmac is a bash script which calls an executable named ‘marmalade’ created by our compiler (marmalade.ml). It takes in two arguments: a marmalade file and the name of the executable to be created.

→ Sample programs are available in the “marmalade_sample_programs” directory
Language Reference Manual:

**Lexical Conventions:**

**Comments:**

Comments are ignored by the compiler and have no effect on the behavior of programs. There are is only one style of comments in Marmalade: multi-line.

Multi-line comments are initiated with a slash and star character `/*` and terminated with a star and slash character `*/`. The compiler ignores all content between the indicators. This type of comment does not nest.

```
/* this is a comment */
```

**Whitespace:**

Whitespace consists of any sequence of blank and tab characters. Whitespace is used to separate tokens and format programs. All whitespace is ignored by the marmalade compiler. As a result, indentations are not significant in Marmalade.

**Tokens:**

In Marmalade, a token is a string of one or more characters consisting of letters, digits, or underscores. Marmalade has 3 kinds of tokens:

1. Identifiers
2. Keywords
3. Operators

**Identifier Tokens:**

An identifier consists of a sequence of letters and digits. An identifier must start with a letter. A new valid identifier cannot be the same as reserved keywords or pitch literals (see Keywords and Literals). An identifier has no strict limit on length and can be composed of the following characters:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
```
Keywords:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>funk</td>
<td>Function declaration</td>
</tr>
<tr>
<td>return</td>
<td>Function return keyword</td>
</tr>
<tr>
<td>if</td>
<td>Conditional Expression</td>
</tr>
<tr>
<td>else</td>
<td>Conditional Expression</td>
</tr>
<tr>
<td>while</td>
<td>Conditional Expression</td>
</tr>
<tr>
<td>and</td>
<td>Boolean AND</td>
</tr>
<tr>
<td>or</td>
<td>Boolean OR</td>
</tr>
<tr>
<td>int</td>
<td>Integer data type</td>
</tr>
<tr>
<td>int_list</td>
<td>Integer list data type</td>
</tr>
<tr>
<td>string</td>
<td>String data type</td>
</tr>
<tr>
<td>str_list</td>
<td>String list data type</td>
</tr>
<tr>
<td>measure</td>
<td>Measure data type</td>
</tr>
<tr>
<td>phrase</td>
<td>Phrase data type</td>
</tr>
<tr>
<td>song</td>
<td>Song data type</td>
</tr>
<tr>
<td>timesig</td>
<td>Time signature data type</td>
</tr>
<tr>
<td>play</td>
<td>Standard library function to play data types</td>
</tr>
<tr>
<td>print</td>
<td>Standard library function to print data types</td>
</tr>
<tr>
<td>output_midi</td>
<td>Standard library function to write data type to a .midi file</td>
</tr>
</tbody>
</table>

Operator tokens:

```
+    -    *    /    =    $  
[]   &    |    ==    !=    !<  
<=   >    >=   .    :  
```

These will be detailed in the ‘Operator Usage’ section.
**Data Types:**

---

**Integer:**

An integer declaration consists of the ‘int’ declaration, a variable name, an assignment operator and an optional sign and any number of operators. Any arithmetic operators can be applied to integers as in any other programming language.

Example Declarations:

- `int i = 5;`
- `int i = -5;`

**String:**

A string declaration consists of the ‘string’ declaration, a variable name, an assignment operator, followed by a string defined in quotes. The string can contain any of the following characters:

- `letter -> [‘a’- ‘z’ ‘A’- ‘Z’]`
- `digit -> [‘0’- ‘9’]`
- `symbols -> ‘_’ | ‘ ‘ | ‘.’ | ‘’ | ‘?’ | ‘$’ | ‘:’ | ‘;’ | ‘?’ | ‘!’ | ‘|’ | ‘/’ | ‘\’ | ‘{$’ | ‘}’ | ‘&’ | ‘#’ | ‘@’ | ‘?’ | ‘<’ | ‘>’ | ‘+’ | ‘-’ | ‘’`

Example Declarations:

- `string s = “string”;
- string s = “Hello World!”;`

**Note:**

A note consists of the ‘note’ declaration, a positive integer, a period and a character indicating the note value. The positive integer represents the specific note to be played and the type indicates the note value. For example:

`44 → C4 (the 4th C on a piano)`

The character after the period represents the note value, or length that the note is played.

- `*s → sixteenth note`
- `*e → eighth note`
- `*q → quarter note`
- `*h → half note`
- `*w → whole note`
Example Declarations:

- note n_0 = 44.s
- note n_1 = 44.e;
- note n_2 = 44.q;
- note n_3 = 54.h;
- note n_4 = 68.w;

Time-Signature:

A time-signature consists of the ‘timesig’ declaration and two integers indicating how a set of notes should be played. The time signature should be separated by a colon to indicate the numerator and denominator.

Example Declaration:

- timesig t_sig_0 = $(4:4);
- timesig t_sig_1 = $(6:8);

Instrument:

An instrument is a set of capital letters which indicate what set of sounds a given set of notes will map to.

See the appendix for the full list of instruments that can be used in Marmalade.

Tempo:

The tempo of a song is defined as the speed that a passage of music should be played. In the case of this language, tempo must be a positive integer that can only be applied to a song. Examples can be found in the Song section.
Lists:

In Marmalade, lists are the only container for storing sets of data. Lists are of variable length. The lists below are list types defined in this language. See ‘Operator Usage’ for more specifics.

Integer List:

An integer list declaration consists of the ‘int_list’ declaration, a variable name, an assignment operator, followed by a list of integers or variable names representing integers.

Example Declarations:

- int_list i = [42, 47, 54, 19, 22];
- int_list i = [0];
- int_list i = [];

String List:

A string list declaration consists of the ‘str_list’ declaration, a variable name, an assignment operator, followed by a list of strings or variable names containing strings.

Example Declarations:

- str_list s = [“marmalade”, “language”, “plt”];
- str_list s = [“Hello World!”];
- str_list s = [];

Measure:

Lists of notes in Marmalade are treated as measures. Given a list of notes:

    measure m_0 = {} [44.q, 64.h, 89.q];

If the user would like to add a time signature, he defines it between the parentheses following the $ symbol. If nothing is typed between these parentheses the time-signature is automatically inferred to be 4/4. Thus ‘$/’ represents the default time signature of a measure.

Here is an example with a time signature (6/8) included by the user:

    measure m_1 = $6:8) [42.e, 36.q, 50.h];

If notes are predefined, a measure can also be declared as such:

    note n1 = 44.q;
note n2 = 55.q;
measure m_1 = $(3:4) [n1, n2];

**Phrases:**

A Phrase is a list of measures to be played in succession from first to the last element. Each phrase is associated with an instrument. For instance:

```
ph_0 = $$() [m_0, m_1]; /* m_0 and m_1 are measures defined above */
```

The `$$()` indicates that the user opted to not specify an instrument for the phrase, and so the default instrument, piano, will be used. (Note: the default symbol of the phrase has one more `$` than the default symbol of the measure).

The user can also input his own instrument:

```
ph_1 = $(GUITAR) [m_0, m_1];
```

which would make `ph_1` the same set of notes with the same time signature as `ph_0` played by a guitar instead of a piano.

If the measures are not predefined, a phrase can also be declared as below:

```
ph_0 = $$() [$(6:8)[44.q, 64.h, 89.q], $(3:4) [36.q, 50.h]];
```

As mentioned in the introduction, the programmer should note that a phrase does not need to represent all the notes being played by a given instrument at a given time. Rather it is more analogous to the left hand of a piano, which can play at the same time as the right hand.

**Songs:**

A song is a list of phrases to be played concurrently. An example of a song could be:

```
song_0 = $$$$() [ph_0, ph_1];
```

This song will play phrases `ph_0` and `ph_1` simultaneously. `$$$$()` represents the default tempo (beats per minute) a song is set to, which is 60 bpm. (Note: the default symbol here has one more `$` than the phrase default symbol, and two more `$` than the measure default symbol).

Of course the user can set one himself as well:

```
song_1 = $(120) [ph_0, ph_1];
```

As demonstrated above, phrases can also be declared directly to create a song. For example:
song_1 = $(120) [ [ $(PIANO) [ $(6:8)[64.h, 89.q], $(3:4) [36.q, 50.h]], $$() [ $(6:8)[44.q, 64.h, 89.q], $(3:4) [36.q, 50.h] ] ]);  

**Operator Usage:**

<table>
<thead>
<tr>
<th>Arithmetic Operators</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>x + 3</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>x - 2</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>5 * x</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>12 / x</td>
</tr>
</tbody>
</table>

Arithmetic operators are listed in increasing precedence, addition and subtraction having the least and multiplication and division having the most. Also, addition and subtraction can be applied to notes. For example:

45.q + 5                      /* this expression has the value: 50.q */

<table>
<thead>
<tr>
<th>Assignment Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Assigns value from left hand side to right hand side</td>
<td>int a = 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>List Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Access Element</td>
<td>list&amp;1</td>
</tr>
</tbody>
</table>

List Operator Example:

```plaintext
m_1 = $$() [35.q, 35.h];
n_1 = m_1&1; /* n_1 == 35.q */
```

**Precedence of Operations:**

ASSIGNMENT, LAST ELEMENT INSERTION (lowest)
### Logical Operator

<table>
<thead>
<tr>
<th>Logical Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>AND</td>
<td>A and B</td>
</tr>
<tr>
<td>or</td>
<td>OR</td>
<td>A or B</td>
</tr>
<tr>
<td>==</td>
<td>EQUALS</td>
<td>A == B</td>
</tr>
<tr>
<td>!=</td>
<td>NOT EQUAL</td>
<td>A != B</td>
</tr>
</tbody>
</table>

A or B
/* If A or B is not 0, the entire expression is evaluated to be 1 */

A and B
/* If A and B are both not 0, then the expression is 1, otherwise 0 */

A == B
/* If A has the same value as B, then A == B is 1 */

A != B
/* If A does not have the same value as B, then A != B is 1 */

### Definition Operator

<table>
<thead>
<tr>
<th>Definition Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>Define time signature, instrument, and tempo. Also indicates a function call returns something.</td>
<td>Instrument: $(GUITAR) Time Signature: $(4:4) Tempo: $(120)</td>
</tr>
<tr>
<td>[]</td>
<td>Define list</td>
<td>[35.q, 46.h, 42.q]</td>
</tr>
<tr>
<td>()</td>
<td>Define funk list (application shown later in funktons)</td>
<td>(play(), print(), play())</td>
</tr>
</tbody>
</table>
**Reserved Words:**

```
```

**Functions:**

`funk`

The keyword ‘funk’ is used to indicate the beginning of a function. For more detailed usage of functions, see the ‘Funktions’ section.

`return`

The keyword ‘return’ is used only in functions to signify a return value from the function. This keyword can only return one object, since a function can only return one object. A function must have a return value, or else an error will be thrown. For examples of use, see the ‘Funktions’ section.

**Control Flow:**

`if else`

‘if’ must be followed by an expression that evaluates to a boolean in parentheses and the body following must be contained in braces, such as:

```
if ( /* boolean expression here */ ) {
    /* body */
}
```

An ‘if’ block can stand alone, but ‘else’ must be accompanied by at least an ‘if’.

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

`while`

The keyword ‘while’ is implemented similarly to how ‘if’. The expression following ‘while’ must evaluate to a boolean expression. For example:

```
while ( i < 5 ) {
    /* body */
}
```
**Standard Library:**

**play**

The 'play' keyword will allow a user to play any set of notes once the code is run. This includes a single note or any list object containing a series of notes. 'play' only takes in one argument. Notice that the measure it's playing is in a list and how play is surrounded by parentheses; function application will be discussed in the functions section.

```plaintext
m_2 = $() [25.q, 26.q];
(play()) [m_2]; /* play m_2 */
```

**print**

The 'print' keyword will be able to print a string literal or any defined data type in the language. It will print out to the standard output in a way that is readable to the user, and only takes in one argument, which is the argument directly after the keyword. The usage of this keyword is similar to that of the 'play' keyword.

```plaintext
(print()) [“hello world!”]; /* print ‘hello world!’ */
```

**write**

The 'write' keyword performs almost the same as 'play' but will output a midi file, called 'out.mid'.

```plaintext
m_2 = [25.q, 26.q];
(write()) [m_2]; /* writes m_2 to a .midi file */
```

**length_measure**

The 'length_measure' keyword represents a function that can only be applied to measures and will return the length of the measure's note list.

```plaintext
int m_len = $length_measure(m); /* the $ signifies that the function is returning a value */
/* m has type measure */
```

**length_phrase**

The 'length_phrase' keyword represents a function that can only be applied to phrases and will return the length of the phrase's measure list.

```plaintext
int p_len = $length_phrase(ph); /* ph has type phrase */
```

**length_song**
The ‘length_song’ keyword represents a function that can only be applied to songs and will return the length of the song’s phrase list.

```c
int s_len = $length_measure(song); /* song has type song */
```

**length_int_list**

The ‘length_int_list’ keyword represents a function that can only be applied to int_lists and will return the length of the int_list.

```c
int il_len = $length_measure(intli); /* inli has type int_list */
```

**length_string_list**

The ‘length_string_list’ keyword represents a function that can only be applied to string_lists and will return the length of the string_list.

```c
int strl_len = $length_measure(strl); /* strl has type string_list */
```

**evaluate_note**

The ‘evaluate_note’ keyword represents a function that can only be applied to a note. It creates a new copy of the note the function was applied to.

**evaluate_measure**

The ‘evaluate_measure’ keyword represents a function that can only be applied to a measure. It creates a new copy of the measure the function was applied to.

**evaluate_phrase**

The ‘evaluate_phrase’ keyword represents a function that can only be applied to a phrase. It creates a new copy of the phrase the function was applied to.

**evaluate_song**

The ‘evaluate_song’ keyword represents a function that can only be applied to a song. It creates a new copy of the song the function was applied to.
**Other Expressions:**

All expressions are made up of a sequence of variables, operators, & string literals.

**Variables**

All variables are of type string literal or one of the defined data types in the language. Variables must begin with a letter and can contain any combination of letters, digits, or the underscore ‘_’.

**Scope:**

The scope of all variables is contained to the the area limited between the outermost level of braces in which a variable is defined. For example, in the transpose function the scope of i is from lines 1-6 and the scope of m_1 is from lines 2-5. Local variables cannot be defined in a function however, so ‘i’ and ‘m_1’ exist outside the transpose function.

```plaintext
1    funk transpose(int i, measure m_1) {
2        m_1 = [40.h];
3        while (i < 5) {
4            /* body */
5        }
6    }
```

If there is a program with no outer braces, then the scope of the variable exists within the entire program. Scope will be fleshed out in more detail in the Funktions section.

**Boolean Expressions:**

Boolean expressions are defined as any expression that returns true or false. There is no boolean data type in our language. There must be a boolean expression using an operator like ‘and’, ‘or’, and ‘<’ between the parentheses of an if statement or while loop. So the while loop displayed below does not compile:

```plaintext
int i = 0;
while (i) {
    /* body */
}
```
Funktions:

**Very Important:** Local variables cannot be declared within the body of a function, if block, or while block.

This is the reason why function transpose_measure_w has so many arguments; the variables it takes in are all the variables being used within the function. For this reason, there is function ‘transpose_measure’ which takes in the two crucial arguments, then calls ‘transpose_measure_w’ with all of the necessary arguments.

/* This function transposes a measure by some value steps */

funk measure measure transpose_measure_w(measure m, int steps, int counter, int j, note k, measure l)
{
    j = $length_measure(m); /* standard library function */
    counter = 0;
    l = $evaluate_measure(m); /* standard library function */

    while(counter < j)
    {
        k = l&counter; /* access l element at position counter */
        l&counter = k + n; /* put new value at position counter in l */
        counter = counter + 1;
    }

    return l;
}

funk measure measure transpose_measure(measure m, int steps)
{
    return $transpose_measure_w(m, steps, 0, 0, 44.q, $() [55.h]);
}

/* keyword **funk** indicates following code block is a function */
/* the function is scoped with curly brackets */

All functions need to have an implicit parameter, which can either be a measure, phrase, or song. The first ‘measure’ following ‘funk’ indicates that the only implicit parameter this function takes is a measure. The second ‘measure’ indicates the return type, which in this case is a measure. The string after the return type is the name of the function: ‘transpose_measure_w’. The list in the parentheses after the name of the function are all the variables used within the function, as none can be declared within.
Applying multiple functions:

/* m_1 is a phrase */

(play(), play(), play()) [m_1, $\text{transpose\_measure}(m_1, 3),
$\text{transpose\_measure}(m_1,5)];

Functions that do not return anything can be placed in a list and applied to a list of
arguments. In the example above, the first ‘play’ is applied to m_1, the second play is
applied to the return value of $\text{transpose\_measure}(m_1, 3)$, and the third play is applied to
the return value of $\text{transpose\_measure}(m_1,5)$.

(print(), play()) [“hello”, m_1];

/* this prints ‘hello’ and plays measure m_1 */
Project Plan:

Specification

We used our weekly meetings in the first half of the semester to shell out our Language Proposal and Language Reference Manual. We often encountered issues adding features exactly as defined in our original LRM (i.e. static vs dynamic type system), so we had to update it as we went along.

Development

The development Marmalade's compiler began with implementing the features defined in the original Proposal and LRM in Marmalade's parser and scanner. The project was source-controlled through a git repository, and only the main branch was used, so as to prevent unnecessary merges conflicts. After creating a scanner, parser, and abstract syntax tree, we then created an initial java generator for Hello World. Afterwards we implemented a symbol table as well as an SAST to for semantic analysis. Then, we made a more robust java generator, using the verified objects from the SAST. Once the complete front-to-back progress was compiling marmalade tests and into Java executables, we added features one-by-one through the architecture, wrote tests for that feature, and ran our test suite to verify the functionality of the feature.

Testing

The tester script (run_tests.sh) was inspired by the MicroC version and was added around the Hello World stage of the project, so that tests could be added in. Every time a new feature was implemented, new tests were added to verify that the feature was working.

Programming Style Guide

Generally, we conformed to these general style conventions:

- Indentation → 4 spaces (or 8 for small branches)
- Characters/Line → Max 100

Roles and Responsibilities

Cathy Jin → System Architect
Savas Petridis → Language Guru
Uzo Amuzie → Tester
Raphael Norwitz → Manager
**Development Environment**

- Text Editor → Sublime Text 2/3
- Development Machines → Mac OS X, ArchLinux Virtual Machine
- Compiler Environment → OCaml 4.02.03
- Automatic Build (OCaml) → Make
- Testing Environment → Shell Scripts
- Version Control → Github/git

**Project Log**

Our git commit log can be found here:
→ [https://github.com/savvaspetridis/marmalade/commits/master](https://github.com/savvaspetridis/marmalade/commits/master)

Our team meeting log can be found here:
→ [https://docs.google.com/document/d/12bon_RbHgiMtegHVUVAjE5TC7hQV3rASFgyrxrVgQ3Y](https://docs.google.com/document/d/12bon_RbHgiMtegHVUVAjE5TC7hQV3rASFgyrxrVgQ3Y)
**Architectural Design:**

**Scanner**

The Scanner is passed an input marmalade (.marm) source file and converts the file to a tokenized output. It discards all whitespace and comments and raises an error if any invalid character sequences are encountered (e.g. invalid identifier or escape sequence).

**Parser**

The Parser is passed the tokenized stream from the scanner. It matches the tokens to a grammar defining the marmalade language. (This is the language structure defined in AST.) Syntax errors in the marmalade code will be identified during parsing, resulting in a raised exception.

**Abstract Syntax Tree (AST)**

The AST defines the rules and structure for the marmalade language in a Context Free Grammar (CFG). This includes all primitive types and things like variables, blocks, and funktions. This is the intermediate phase of a marmalade program, after being parsed but before being semantically checked.

**Symbol Table**

Using an approach adapted from corgi, we used the block ids set by the parser and translated these block ids into scope ids. The symbol table is a string map of declared variables and funktions. The symbol table is also used to enforce unique funktion and variables names within each scope (i.e. declaring “int swag” twice in the same scope) and to verify that each variable and funktion is visible within the current scope.
**Semantically-checked Abstract Syntax Tree (SAST)**

A SAST is generated at this stage with the data types from the AST but with additional information of the type attached. Through the construction of the SAST, the additional typing information allows us to check for type compatibility. Additionally, we check to make sure that funktion calls and return types of the funktions match the funktion declarations that we parsed. Any type mismatches or semantic errors will be reported during this step.

**Java Generation**

After doing research into music libraries developed for several different languages, we settled on using jMusic since it had some of the functionality most similar to what we were looking to implement in Marmalade. Although we were able to use certain functionalities in jMusic, a custom library (implemented as marmalade.jar in the project) was created to help create common functions for different objects.

**Marmac**

Marmac is an all-purpose shell script utility that (1) streamlines marmalade environment setup, (2) compiles marmalade source compilation to down to JAVA and binary .CLASS files, and (3) creates a new shell script -- given the same base name as the marmalade source -- that will run the JAVA executable upon its own execution. A vital piece of marmac is loading the location of the jMusic JAR dependency to your CLASSPATH variable in your local file system, enabling jMusic library calls to be made in Generated Java Code.
Testing Plan:
--

Script & Regression Testing

Our test script and testing infrastructure was adapted from MicroC. We developed a shell script that automated testing from a directory named "tests" that contained both unit tests written in marmalade source code, as well as target ".out" files with the expected standard output from each respective marmalade test. Source files were ran from the "tests" directory and their output (.java, .class, .out files) was sourced to a "testdir" directory that was timestamped. We also enabled automatic regression testing by adding an automated archival feature where each time the test script was run, all "testdir" directories from previous runs were automatically sourced to a "previous_tests" directory, with the most recently time-stamped "test_output" directory remained in the project's TLD. This allowed for our file system to remain clean and for previous test results to be easily referenced.

Issues Faced

Slightly inconsistent jMusic standard output: Our test suite adopted a MicroC like process where we utilize the “diff” utility to compare the output generated when marm source is compiled and run versus expected output pre-populated in a separate .out file.

```
[ua2144@uzo marmalade]$ ./test_note_play
jMusic Play: Playing score One note score using JavaSound General MIDI soundbank.
jMusic Play: Waiting for the end of One note score.
[ua2144@uzo marmalade]$ [ua2144@uzo marmalade]$
[ua2144@uzo marmalade]$ ./test_note_play
jMusic Play: Playing score One note score using JavaSound General MIDI soundbank.
jMusic Play: Waiting for the end of One note score.
jMusic MidiISynth: Stopped JavaSound MIDI playback
```

The “test_note_play” test has a corresponding “test_note_play.out” file that contains the stdout in green. The orange output is the typical/expected output, but once every now and then, the stdout in yellow is generated, which makes the test “fail”. This has popped up on occasion for tests with the format “test_<musicStructure>_play***”

Testing MIDI Creation: It was a challenge to add tests for the “write” library function call directly to the structure of test suite because of the .MIDI files generated by write. Upon Instructor/TA suggestion, time was spent looking for an alternate to the “diff” tool for .MIDI files but no useful solution was found. Instead, .MIDI files generated were compared to samples with “diff” straight from the command line (byte code was compared). They were also physically played and compared back-to-back as an informal sanity check.

Deprecated Tests: We made major adjustment to our language features in the last few weeks. We moved the test suite to “tests_deprecated “and populated a new “tests” folder with relevant unit tests.
Lessons Learned:
--

Uzo Amuzie (Tester)

All in all, I was glad to hear Professor Edwards say, "oh, that's cool," as anticlimactic as that sounds. This was a long semester that reaped a lot of lessons learned. For one, our group would've benefitted greatly from maintaining our early regularity with team meetings. As the semester unravelled, there was a bit of reluctance to keep our regular meeting time for fear of not accomplishing much in meetings, but we recently learned that the more time we spend together, the better. Even if it is to go over the concepts from lecture or the project plan/structure together and ensure that everyone was on the same page. Open and honest communication was a huge factor as well.

Personally, I wish I would have spoken up more and asked questions (to the TA team, professor, and my teammates) as soon as I was confused or didn't understand a concept. Often times, I had an idea of what I thought I needed to do, only to find out there were corners that I didn't cover. Developing good relationships early on with teammates will help to mitigate the feeling of apprehension of asking of help or clarification.

I'd advise heavily considering 3 things before taking PLT: (1) your potential group, (2) your potential semesterly workload, and (3) your learning/working/communication style. I noticed that most people had their groups set up on/before the first day of class, so strongly consider -- even plan ahead for -- taking this class with a cohort of friends you've gone through some CS classes with, if possible. Think twice about taking 3 (or even 2) heavy programming classes in the same semester. And lastly, know thyself.

Cathy Jin (System Architect)

Some of the biggest challenges we faced were organizing a semester-long group project efficiently, both in terms of working as a group and working on the code. Although we started off at the beginning of the semester meeting regularly, we weren't able to continue this pattern. Even though we thought meeting as a group took up more of everyone's time and sometime seemed like a fruitless use of time, I think meeting up in person would have put more responsibility on people and forced us to talk about and work on the project more consistently.

Something else that was a challenge was planning our work around the language better. We spent a lot of time thinking about interesting features to implement and how they would function within our language, but by the end we realized we should have focused on some of the more basic features first before trying to get other ones to work.
Even though I think we were able to turn in a reasonably cool project, having better time management skills as a team and better communication and accountability for work would have given us the chance to implement a few more features.

Savvas Petridis (Language Guru)

In the end, PLT was a positive experience. It was an incredible opportunity to build a significant piece of software from the bottom up in a semester. The process of deciding what kind of language we wanted to make and what tools we were going to use to actually implement the language was very insightful. I learned to maintain my reservations and not jump into programming without a clear idea of what I wanted to do. We started off with high expectations and an idea for a music creation language that was more clever than the one we actually implemented. But, this is to be expected. In the end we scrapped a few of the more complicated ideas and built a particularly solid, easy-to-read music language.

There were challenges of course, the primary one being team management. We would meet and perhaps only two to three of us would actually be thinking about the project at once. We needed to define more concrete roles and really hold people accountable for their work and define a strict set of deadlines. Because of this, we never really had a good testing system during the entire semester, which of course hurts our project.
A big piece of advice for other teams is to really be careful with who you take. Be sure to choose reliable workers. Finally, my last piece of advice is to get the core details of your language hammered out first before attempting to add any fancy functions. We wasted a great deal of time on parts of our language that made it ‘cool’. We should have really nailed down a robust testing system and solidified our language.

Raphael Norwitz (Manager)

Overall this was a terrific experience. Building a substantial piece of software from the ground up is not something you’ll get to do all that often and though it’s really daunting and potentially horrible if you push it off till the last minute, you’ll really get out what you put in. Expect to have mixed feelings throughout the process, especially before you start generating code. Were I to have written this a few days ago, before I started coding in Marmalade, I may have voiced different sentiments.

Given that our goal was to create something that would allow people to quickly and efficiently write readable code that played music, I’d absolutely call Marmalade a success. The way we break music down into discrete interchangeable parts removes the inherent clunky-ness associated with writing music in a typical object oriented language and the way we evaluate functions via lists allows the user to perform many more operations in a tiny fraction of the code, which when you're doing something as involved as writing music is a blessing. We did have to scrap two of our biggest features at the last minute, but in retrospect they weren't critical to the core functionality of the language. To that end, were I
to do it again I would start off focusing more on the actual structure of the language, rather
than what Edwards calls ‘syntactic sugar’. Our decisions to make Marmalade almost script
like, so a user can just start coding and in two or three lines produce a useable program,
ended up being a really cool, but if we had a different purpose in mind this may not have
worked out so nicely.

In terms of general advice, there’s a lot of finicky stuff you’ll have to do but if you’re
resourceful and aren’t scared to ask help from Edwards and the TA’s (provided you’re not
wasting their time) you’ll be fine. Though OCaml looks completely intractable to begin with,
if you go to Edwards with specific questions, he’ll really clear things up. One thing we as a
group could have done better is get an understanding of the bigger picture before diving in
and writing a parser. Had I spent a good 10-15 hours at the beginning of the semester
looking at other projects and making sure I at least had a sense of what each of the files
were doing, it would have saved us the chore of rewriting the parser. On the other hand, I
think the way we looked closely at three different languages, Corgi, Sheets and Fry (all
from Fall 2014), each of which had a few features we wanted to emulate but which were
otherwise very different, worked really well. The case studies gave us a wide view of
different implementations which helped us implement some of our more unorthodox
features.

Maybe the most valuable thing I got out of this class was experiencing firsthand the
difficulties that come with trying to manage people in the context of a sizable project. An
accommodating, hands-off leader may work when you have an all around solid team, but in
the grand scheme of PLT those are rare. Don’t assume your teammates’ code works just
because they say it works, or they show you it working on their machine or in a specific
case. Look at their code and if you see something fishy, be vocal about it and don’t be
scared to threaten and go to Edwards if there’s repeated misbehavior. Though Marmalade
feels like such an excellent project now, I recognize that there’s a high likelihood there are
fixable bugs because our test suite isn’t comprehensive. This would not have been the case
had I been more forceful and proactive.

As a manager, I also learned how judicious one needs to be in delegating work. Especially if
you’ve worked something out in your head, make sure you understand exactly how it’ll
churn out code on the other end before dumping it on a teammate. Much of one of my
teammates time was spent building a complicated function to parse expressions which was
a million times more complicated than it needed to be, and made code generation on the
front end totally impossible. Though mistakes happen, this one could easily have been
avoided and hopefully this mistake will finally teach me to plan rigorously before I code.
All complaints aside though, I’m extremely happy with the end result and I’ve gained so
much experience and programmatic maturity that I’d unquestionably do it again,
irrespective of the team.
Appendix
Appendix:

**Instrument List:**

The following instruments can be applied in Marmalade:

<table>
<thead>
<tr>
<th>Instrument</th>
<th>Alternate 1</th>
<th>Alternate 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC_GUITAR</td>
<td>FRENCH_HORN</td>
<td>POLYSYNTH</td>
</tr>
<tr>
<td>ACCORDION</td>
<td>GLOCK</td>
<td>RECORDER</td>
</tr>
<tr>
<td>AGOGO</td>
<td>GUITAR</td>
<td>REED_ORGAN</td>
</tr>
<tr>
<td>ALTO</td>
<td>HARMONICA</td>
<td>SAXOPHONE</td>
</tr>
<tr>
<td>ALTO_SAX</td>
<td>HARP</td>
<td>SITAR</td>
</tr>
<tr>
<td>BAGPIPE</td>
<td>HARPSICHORD</td>
<td>STEELDRUM</td>
</tr>
<tr>
<td>BANJO</td>
<td>HONKYTONK</td>
<td>STRINGS</td>
</tr>
<tr>
<td>BARITONE_SAX</td>
<td>HONK</td>
<td>TOM_TOMS</td>
</tr>
<tr>
<td>BASSOON</td>
<td>HORN</td>
<td>TROMBONE</td>
</tr>
<tr>
<td>BELLS</td>
<td>JAZZ_GUITAR</td>
<td>TRUMPET</td>
</tr>
<tr>
<td>BRASS</td>
<td>JAZZ_ORGAN</td>
<td>TUBA</td>
</tr>
<tr>
<td>CELLO</td>
<td>KALIMBA</td>
<td>VIBRAPHONE</td>
</tr>
<tr>
<td>CHOIR</td>
<td>MARIMBA</td>
<td>VIOLA</td>
</tr>
<tr>
<td>CHURCH_ORGAN</td>
<td>MUSIC_BOX</td>
<td>VIOLIN</td>
</tr>
<tr>
<td>CLARINET</td>
<td>OBOE</td>
<td>VOICE</td>
</tr>
<tr>
<td>CYMBAL</td>
<td>OOH</td>
<td>WHISTLE</td>
</tr>
<tr>
<td>DOUBLE_BASS</td>
<td>ORGAN</td>
<td>WOODBLOCKS</td>
</tr>
<tr>
<td>DRUM</td>
<td>PAN_FLUTE</td>
<td>XYLOPHONE</td>
</tr>
<tr>
<td>ECHO</td>
<td>PHONE</td>
<td></td>
</tr>
<tr>
<td>ELECTRIC_BASS</td>
<td>PIANO</td>
<td></td>
</tr>
<tr>
<td>ELECTRIC_GUITAR</td>
<td>PICCOLO</td>
<td></td>
</tr>
<tr>
<td>FIDDLE</td>
<td>PIPE_ORGAN</td>
<td></td>
</tr>
<tr>
<td>FLUTE</td>
<td>PIPES</td>
<td></td>
</tr>
</tbody>
</table>
Marmac and marmalade.ml:

```
#!/bin/bash

# run the marmalade compiler
./marmalade $2 < $1

# compile the java source
javac -classpath ./javaclasses/jMusic1.6.4.jar:./javaclasses/marmalade.jar: $2.java

# create a Bash script which runs the java program and set the
# privileges so it's accessible
STR=$'#!/bin/bash

njava -classpath ./javaclasses/jMusic1.6.4.jar:./javaclasses/marmalade.jar:

'echo "$STR" $2

> ./$2

chmod 755 ./$2
```

Listing 1: marmac

```
(* Compiler for Marmalade *)

open Printf

let _ =

let lexbuf = Lexing.from_channel stdin in

let program = Parser.program Scanner.token lexbuf in

let env = Table.create_table program in

let sast_pgm = Sast.confirm_semantics program env in

let compiled_program = (*Compile.to_java program Sys.argv.(1)*)

Javagen.gen_pgm sast_pgm Sys.argv.(1) in

fprintf file "%s" compiled_program;
```

Listing 2: marmalade.ml

Scanner:

```
{ open Parser }

let digit = ['0'-'9']

let letter = ['a'-'z' 'A'-'Z']

rule token = parse

[ '"' | '\t' | '\r' | '\n' ] { token lexbuf } (* Whitespace *)

[ '/' | ' ' ] { comment lexbuf }

[ '(' | LPAREN ]

[ ')' | RPAREN ]

[ '{' | LBRACE ]

[ '}' | RBRACE ]

[ '[' | LBRACK ]

[ ']' | RBRACK ]

[ ';' | SEMI ]

[ '+' | PLUS ]

[ '-' | MINUS ]

[ '*' | TIMES ]

[ '/' | DIVIDE ]

[ '=' | ASSIGN ]

```
Parser:

def scope_id = ref 1

def inc_block_id (u:unit) =
    let x = scope_id.contents in
    scope_id := x + 1; x

Listing 3: scanner.mll
program :<nothing> /
|program fdecl [{ stmts = $1 : stmts ; funcs = $2 : : $1 : funcs }
|program stmt [{ stmts = $2 : : $1 . stmts ; funcs = $1 . funcs }
|/* Function Declaration */
fdecl:
|fdecl ID LPAREN arguments RPAREN stmt_list RBRA
|{ ret_type = List.hd (List.rev $1) ;
|f_type = List.tl (List.rev $1) ;
|fname = $2 ;
|args = $4 ;
|body = { locals = $4 ; statements = List.rev $7 ; block_id =
|inc_block_id () }
|}
| /* Argument list creator for Function Declarations */
|arguments:
|/* nothing */ { [ ] }
|arg_list { List.rev $1 }
|/* Argument list */
arg_list:
  fnmod { [ $1 ] }
  | arg_list COMMA fnmod { $3 :: $1 }
  fnmod:
  | INT ID { ($2, false, Int)}
  | STRING ID { ($2, false, String)}
  | NOTE ID { ($2, false, Note)}
  | SONG ID { ($2, true, Song)}
  | MEASURE ID { ($2, true, Measurepoo)}
  | INTLIST ID { ($2, true, Intlist)}
  | STRL ID { ($2, true, Stringlist)}
  | PHRASE ID { ($2, true, Phrase)}
  | TIMESIG ID { ($2, false, Timesig)}
  | INSTR ID { ($2, false, Instr)}
  | TEMPO ID { ($2, false, Tempo)}
*/ a stmt can be an expression, variable modification, a conditional stmt, or return */
stmt:
  expr SEMI { Expr($1) }
  | fnmod SEMI { VarDecl($1) }
  | conditional_stmt { $1 }
  | RETURN expr SEMI { Return($2) }
*/ all type declarations */
type_dec:
  INT { Int}
  | NOTE { Note}
  | MEASURE { Measurepoo}
  | PHRASE { Phrase}
  | SONG { Song}
  | STRING { String}
  | LIST { List}
  | INTLIST { Intlist}
  | STRL { Stringlist}
  | TIMESIG { Timesig}
  | INSTR { Instr}
  | TEMPO { Tempo}
conditional_stmt:
  IF LPAREN expr RPAREN block %prec NOELSE { If($3, $5, {local = [], statements = []; block_id = inc_block_id ()}) }
  | IF LPAREN expr RPAREN block ELSE block { If($3, $5, $7) }
  | WHILE LPAREN expr RPAREN block { While($3, $5) }
block:
  LBRACE stmt_list RBRACE { {local = []; statements = List.rev $2; block_id = inc_block_id ()} }
stmt_list:
  /* nothing */ { [] }
  | stmt_list stmt { $2 :: $1 }
vmod:
  | type_dec ID ASSIGN expr { Assign($1, $2, $4)}
  | ID ASSIGN expr { Update($1, $3)}
  | list_index ASSIGN expr { Index_Update($1, $3)}
expr:
  app_gen {\$1}
  | list_index {\$1}
  | arith {\$1}
  | add_on_expr {\$1}

list_index:
  | ID INDEX INT_LIT { Index($1, IntLit($3)) }
  | ID INDEX ID { Index($1, Id($3))}

add_on_expr:
  | DOLLAR LPAREN reg_list { Measure($4, TimeSig(4, 4)) }
  | DOLLAR DOLLAR LPAREN reg_list { Phrase ($5, Instr("PIANO")) }
  | DOLLAR DOLLAR DOLLAR LPAREN reg_list { Song($6, Tempo(60)) }
  | DOLLAR LPAREN INT_LIT COLON INT_LIT LPAREN reg_list { Measure( $7, TimeSig($3, $5)) }
  | DOLLAR LPAREN ID LPAREN reg_list { Phrase($5, Instr($3)) }
  | DOLLAR LPAREN INT_LIT LPAREN reg_list { Song($5, Tempo($3)) }

/* beginning of chain of expressions, ordered by precedence */

arith:
  lor { \$1 }

primary_expr:
  | ID { Id($1) }
  | literal { \$1 }
  | LPAREN expr RPAREN { \$2 }

bound_list:
  | ( )
  | bound_list BOUND DASH BOUND { (Ranges($2, $4) :: $1) }

literal:
  | INT_LIT { IntLit($1) }
  | note { $1 }
  | STRING_LIT { String_Lit($1) }
  | DOLLAR function_invocation { \$2 }

/* multiplication */

mul_expr:
  | primary_expr /*lit*/ { \$1 }
  | mul_expr TIMES primary_expr { Binop($1, Times, $3) }
  | mul_expr DIVIDE primary_expr { Binop($1, Divide, $3) }

/* addition */

add_expr:
  | primary_expr PLUS mul_expr { Binop($1, Plus, $3) }
  | primary_expr MINUS mul_expr { Binop($1, Minus, $3) }

/* <=, >= */

r_expr:
  | r_expr LT r_expr { Binop($1, Less, $3) }
  | r_expr LEQ r_expr { Binop($1, Leq, $3) }
186 | r_expr GT r_expr { Binop($1, Greater, $3) }
187 | r_expr GEQ r_expr { Binop($1, Geq, $3) }
188
189 /* equal and not equal */
190
191 eq_exp:
192 | r_expr { $1 }
193 | eq_exp EQ eq_exp { Binop($1, Equal, $3) }
194 | eq_exp NEQ eq_exp { Binop($1, Neq, $3) }
195
196 /* logical And */
197 l_AND:
198 | eq_exp { $1 }
199 | l_AND AND l_AND { Binop($1, And, $3) }
200
201 /* logical Or */
202 l_OR:
203 | l_AND { $1 }
204 | l_AND OR l_OR { Binop($1, Or, $3) }
205
206 /* app_gen creates lists, as well as lists of functions for application */
207
208 app_gen:
209 | funk reg_list { FuncList($1, $2) }
210 | reg_list { BasicList($1) }
211
212 /* parenthesis contain the list of functions to be applied */
213
214 funk:
215 | LPAREN f_arithmetics RPAREN {$2}
216
217 /* make list of function_invocations */
218
219 f_arithmetics:
220 | f_arithmetics COMMA function_invocation { $3 :: $1 }
221 | function_invocation { [$1] }
222
223 /* ID and arguments of function in list */
224
225 function_invocation:
226 | ID LPAREN funk_args RPAREN { FunkCall($1, List.rev $3) }
227 | ID LPAREN RPAREN { FunkCall($1, []) }
228
229 /* make list of function arguments */
230
231 funk_args:
232 | funk_args COMMA arithmeticID_arg { $3 :: $1 }
233 | arithmeticID_arg { [$1] }
234
235 /* arguments can be many expressions: another list, addition expr, logical expr, etc */
236
237 arithmeticID_arg:
238 | list_index { $1 }
239 | app_gen { $1 }
240 | arith { $1 }
241 | add_on_expr { $1 }
242 | function_invocation { $1 }
Abstract Syntax Tree:
While of expr * block
| Return of expr
| Fdecl of fdecl
| Null_Type
| None

(* each block has a list of variables, statements, and an id *)

and block = {
  locals: var list;
  statements: stmt list;
  block_id: int
}

(* function declaration *)

and fdecl = {
  fname: string;
  ret_type: declare_type;
  f_type: declare_type list;
  args: var list;
  body: block;
}

type scope_var_decl = string * bool * declare_type * int

type scope_func_decl = string * declare_type * declare_type list *
declare_type list * int

type decl =
| FuncDecl of scope_func_decl
| VarDecl of scope_var_decl

type program = {stmts: stmt list; funcs: fdecl list}

Listing 5: ast.ml

Symbol Evaluation Table:

(*
 * table.ml of marmalade
 * Creates table for checking SAST
 *)

open Ast

module StrMap = Map.Make(String)

let table_env (table,_) = table
let scope_env (_,scope) = scope
let type_of_funct_args (_,_,p_type) = p_type
let over_scope = Array.make 1000 0

let rec map func lst env =
  match lst with
  | [] -> env
  | head :: tail ->
    let new_env = func head env in
    map func tail new_env
let name_scope_str (name:string) env =
  name ^ "_" ^ (string_of_int (scope_env env))

let rec get_scope name env =
  if StrMap.mem (name_scope_str name env) (fst env) then (snd env)
  else if (snd env) = 0 then raise(Failure("Error: Symbol " ^ name ^ " not declared. " ^ string_of_int (snd env)))
  else get_scope name (fst env, over_scope.(snd env))

let rec get_decl name env =
  let key = name_scope_str name env in
  if StrMap.mem key (fst env) then StrMap.find key (fst env)
  else if (snd env) = 0 then raise(Failure("Error: Symbol " ^ name ^ " not declared in current scope" ^ string_of_int (snd env) ^ "."))
  else get_decl name ((fst env), over_scope.(snd env))

let insert_symb (name:string) (decl:decl) env =
  let key = name_scope_str name env in
  if StrMap.mem key (table_env env) then raise(Failure("Error: Symbol " ^ name ^ " declared twice
  in same scope."))
  else ((StrMap.add key decl (table_env env)), (scope_env env))

let insert_var var env =
  let (name, p_type) = var in
  let is_implicit_array =
    (match p_type with
      (Int | Note | String | TimeSig | Instr | Tempo) -> false
      | _ -> true) in
  insert_symb name (VarDecl(name, is_implicit_array, p_type, (scope_env env))) env

let insert_astvar var env =
  let (name, arr_b, typ) = var in
  insert_var (name, typ) env
  (* insert stmt — matches first, then inserts *)

let rec insert stmt stmt env =
  (match stmt with
    Expr(exp) -> env
    | If(e, bl_1, bl_2) -> let env_1 = insert_code_block bl_1 Wild env in
      insert_code_block bl_2 Wild env_1
    | While(e, bl) -> insert_code_block bl Wild env
    | Fdecl(fdec) -> insert_funk fdec env
    | VarDecl(chan) -> (match chan with
      Assign(typ, id, blah) -> insert_var (id, typ) env
      | Update(str, expr) -> env
      | Index_Update(_, _) -> env
      | _ -> env )
  (* insert contents of a block of code *)
  and insert_code_block block returntp env =
    let (table, scope) = env in
    let id = block_id in
    let env = map insert_astvar block.locals (table, id) in
    let env = map insert_stmt block.statements env in
    over_scope.(id) <- scope;
(table_env env), scope)
and insert_funk func env =
  let (table, scope) = env in
  let arg_names = List.map type_of_func_args func.args in
  let env = insert_symb func.fname (FuncDecl(func.fname, func.
    ret_type, func.f_type, arg_names, scope)) env in
  insert_code_block func.body (func.ret_type) ((table_env env),
    scope)

(* initialize start_env *)

let start_env =
  let table = StrMap.add "print_0" (FuncDecl("print", Null_Type, [
    Int; Note;
    String; Song; Phrase; Measurepoo; TimeSig; Instr; Tempo; List ;
    Intlist; Stringlist; Wild], [], 0)) StrMap.empty in
  let table = StrMap.add "evaluate_note_0" (FuncDecl("evaluate_note",
    Note, [Int; Note; String; Song; Phrase; Measurepoo; TimeSig; Instr;
      Tempo; List ; Intlist; Stringlist; Wild], [Note], 0)) table in
  let table = StrMap.add "evaluate_measure" (FuncDecl("evaluate_measure",
    Measurepoo, [Int; Note; String; Song; Phrase; Measurepoo; TimeSig;
      Instr; Tempo; List ; Intlist; Stringlist; Wild], [Note], 0)) table in
  let table = StrMap.add "evaluate_phrase_0" (FuncDecl("evaluate_phrase",
    Phrase, [Int; Note; String; Song; Phrase; Measurepoo; TimeSig; Instr;
      Tempo; List ; Intlist; Stringlist; Wild], [Phrase], 0)) table in
  let table = StrMap.add "evaluate_song_0" (FuncDecl("evaluate_song",
    Song, [Int; Note; String; Song; Phrase; Measurepoo; TimeSig; Instr;
      Tempo; List ; Intlist; Stringlist; Wild], [Phrase], 0)) table in
  let table = StrMap.add "length_measure_0" (FuncDecl("length_measure",
    Int, [Measurepoo; Note; Phrase; Song; Intlist; Stringlist],
    Measurepoo), [Note], 0)) table in
  let table = StrMap.add "length_phrase_0" (FuncDecl("length_phrase",
    Int, [Measurepoo; Note; Phrase; Song; Intlist; Stringlist],
    Phrase), 0)) table in
  let table = StrMap.add "length_song_0" (FuncDecl("length_song",
    Int, [Measurepoo; Note; Phrase; Song; Intlist; Stringlist],
    Song), 0)) table in
  let table = StrMap.add "length_int_list_0" (FuncDecl("length_int_list",
    Int, [Measurepoo; Note; Phrase; Song; Intlist; Stringlist],
    Intlist), [Intlist], 0)) table in
  let table = StrMap.add "length_string_list_0" (FuncDecl("length_string_list",
    Int, [Measurepoo; Note; Phrase; Song; Intlist; Stringlist],
    Stringlist), [Stringlist], 0)) table in
  let table = StrMap.add "play_0" (FuncDecl("play", Null_Type, [
    Note; String; Song; Phrase; Measurepoo; Wild], [], 0)) table in
  let table = StrMap.add "write_0" (FuncDecl("write", Null_Type, [
    Note; String; Song; Phrase; Measurepoo], [], 0)) table in
  let table = StrMap.add "main_0" (FuncDecl("main", Null_Type, [],
    []), 0)) table in

(* main function in this file — initiates table, inserts
statements and funks *)

let create_table p =
  let env = start_env in
  let env = map insert_stmt (List.rev p.stmts) env in
  let env = map insert_funk (List.rev p.funcs) env in
  let () = Printf.printf "// Symbol Table Created" in
  env

Listing 6: table.ml

Semantic Analysis:

/* Semantic analysis for Marmalade */

open Ast

let fst_of_three (t, _, _) = t
let snd_of_three (_, t, _) = t
let thrd_of_three (_, _, t) = t

(* verified expressions *)

type s_expr =
  | S_1st Lit of int * declare_type
  | S_Id of string * declare_type
  | S_String Lit of string * declare_type
  | S_Note of int * char * declare_type
  | S_Measure of s_expr list * s_expr * declare_type (* S_Note list , S_TimeSig, declare_type *)
  | S_Phrase of s_expr list * s_expr * declare_type (* S_Measure list , S_Instr, declare_type *)
  | S_Song of s_expr list * s_expr * declare_type (* S_Phrase list , *)
  | S_TimeSig of int * int * declare_type (* ex: ((4:4), TimeSig) *)
  | S_Instr of string * declare_type (* ex: (BASS, Instr) *)
  | S_Tempo of int * declare_type (* ex: (120, Tempo) *)
  | S_Binop of s_expr * op * s_expr * declare_type
  | S_Call of string * s_expr * s_expr list * declare_type list * declare_type
  | S_Index of string * s_expr * declare_type
  | S_Arr of s_expr list * declare_type
  | S_DbArr of s_expr * s_expr
  | S_CallList of s_expr list
  | S_Noexpr (* Default – No value *)

(* verified statements *)

type s_stmt =
  | S_CodeBlock of s_block
  | S_expr of s_expr
  | S_Assign of string * s_expr * s_expr * declare_type
  | S_Return of s_expr
  | S_If of s_expr * s_stmt * s_stmt (* stmts of type D_CodeBlock *)
S_Fo r s_stmt * s_stmt * s_stmt * s_block (* stmts of type D_Assign | D_Noexpr * D_Expr of type bool * D_Assign | D_Noexpr *)

S_While of s_expr * s_block
S_Append_Assign of declare_type * string * s_expr list
S_Index_Update of string * s_expr * s_expr * declare_type

and s_block = {
  s.locals : scope_var_decl list;
  s.statements : s_stmt list;
  s.block_id : int;
}

(* verified function declaration *)

type s_func = {
  s.fname : string;
  s.ret_type : declare_type; (* Changed from types for comparison error in confirm_stmt *)
  s.f_type : declare_type list;
  s.formals : scope_var_decl list;
  s.f_block : s_block;
}

type s_program = {
  s.g_vars : scope_var_decl list;
  s.p_funcs : s_func list;
}

let rec get_range l (a:char) b =
  let lower = Char.code a in
  let upper = Char.code b in
  if lower = upper then
    a :: l
  else
    get_range (a :: l) (Char.chr (lower+1)) b

let get_dt fdc = match fdc with
  | Func_Decl(_, dt, it, _, den) -> (dt, it, den)
  | Var_Decl(_, _, dt, den) -> (dt, [dt], den)

(* returns string of the primitive type *)

let string_of_prim_type = function
  | Int -> "int"
  | String -> "string"
  | Note -> "note"
  | Measurepoo -> "measure"
  | Phrase -> "phrase"
  | Song -> "song"
  | TimeSig -> "timesig"
  | Instr -> "instr"
  | Tempo -> "tempo"
  | Intlist -> "int_list"
  | Stringlist -> "str_list"
  | Null_Type -> "null"

(* returns type of expr *)

let rec type_of_expr here = match here with
let rec map_to_list_env func lst env =  
match lst with  
[] -> []  
| head :: tail ->  
let r = func head env in  
r :: map_to_list_env func tail env  

let rec traverse_main func lst =  
match lst with  
[] -> []  
| head :: tail ->  
let r = func head in  
r :: traverse_main func tail  

let drop_funk li =  
match li with  
Expr(v) -> Expr(v)  
| VarDecl(v) -> VarDecl(v)  
| If(exp_1, blk, exp_2) -> If(exp_1, blk, exp_2)  
| While(exp, blk) -> While(exp, blk)  
| _ -> Null_Type  

let confirm_var var env =  
let decl = Table.get_decl (fst_of_three var) env in  
match decl with  
FuncDecl(f) -> raise(Failure("Error: symbol is not a variable"))  
| VarDecl(v) -> let (vname, vararray, vtype, id) = v in  
(vname, vararray, vtype, id)
let confirm_func_decl name env =
let decl = Table.get_decl name env in
match decl with
  Func_Decl(f) -> name
  | _ -> raise(Failure("Error: id " ^ name ^ " not a function"))

let confirm_id_get_type id env =
let decl = Table.get_decl id env in
match decl with
  Var_Decl(v) -> let (_, _, t, _) = v in t
  | _ -> raise(Failure("Error: id " ^ id ^ " not a variable."))

(* get variables *)

let get_vars li =
  match li with
  VarDecl (v) ->
    (match v with
     Assign (dt, iden, v) ->
      (match dt with
       Int -> (iden, false, dt)
       | Note -> (iden, false, dt)
       | Measurepoo -> (iden, false, dt)
       | String -> (iden, false, dt)
       | TimeSig -> (iden, false, dt)
       | Instr -> (iden, false, dt)
       | Tempo -> (iden, false, dt)
       | _ -> (iden, true, dt))
     | Update(iden, v) -> ("", false, Wild))
     | Index_Update(expr_1, expr_2) -> ("", false, Wild))
     | _ -> ("", false, Wild))

  (* confirm correct format of a binary operation *)

let confirm_binop l r op =
  let tl = type_of_expr l in
  let tr = type_of_expr r in
  match op with
  Plus | Minus | Times | Divide -> (match (tl, tr) with
  Int, Int -> Int
  | Note, Int -> Note
  | _, _ -> raise(Failure("Error: Cannot apply + * / op to types " ^ string_of_prim_type tl ^ " + " ^ string_of_prim_type tr)))
  | Equal | Neq -> if tl = tr then Int else (match(tl, tr) with
  Int, Int -> Int
  | Note, Int -> Int
  | _, _ -> raise(Failure("Error: Cannot apply == != op to types " ^ string_of_prim_type tl ^ " + " ^ string_of_prim_type tr)))
  | Less | Greater | Leq | Geq -> (match (tl, tr) with
  Int, Int -> Int
  | Note, Int -> Int
  | Note, Note -> Int
  | _, _ -> raise(Failure("Error: Cannot apply < > <= >= op to types " ^ string_of_prim_type tl ^ " + " ^ string_of_prim_type tr)))
  | And | Or -> (match (tl, tr) with
  Int, Int -> Int
  | _, _ -> raise(Failure("Error: Cannot apply && || op to types " ^ string_of_prim_type tl ^ " + " ^ string_of_prim_type tr)))

  (* map function to list *)
let rec map1 lst func env boo =  
  match lst with  
  [ ] ->  
  | head :: tail ->  
    let ret = func head env boo in  
    ret :: map1 tail func env boo

(* map function to 2d list *)

let rec map2 lst func env boo =  
  match lst with  
  [ ] ->  
  | head :: tail ->  
    let ret = map1 head func env boo in  
    ret :: map2 tail func env boo

(* map function to 3d list *)

let rec map3 lst func env boo =  
  match lst with  
  [ ] ->  
  | head :: tail ->  
    let ret = map2 head func env boo in  
    ret :: map3 tail func env boo

(* convert AST expressions into SAST expressions *)

let rec confirm_expr ex env boo =  
  match ex with  
  IntLit(i) -> S_IntLit(i, Int)  
  | Id(st) -> S_Id(st, confirm_id_get_type st env)  
  | StringLit(st) -> S_StringLit(st, String)  
  | Note(ct, nt) -> S_Note(ct, nt, Note)  
  | Measure(nt_list, time) -> let new_time = confirm_expr time env  
    true in  
    let s_note_list = map1 nt_list confirm_expr env  
    true in  
    S_Measure(s_note_list, new_time, Measurepoo)  
  | Phrase(m_l, inst) -> let verified_list = map1 m_l confirm_expr  
    env boo in  
    S_Phrase( verified_list, confirm_expr inst env boo, Phrase)  
  | Song(s_l, tempo) -> S_Song(map1 s_l confirm_expr env boo,  
    confirm_expr tempo env boo, Song)  
  | TimeSig(num, den) -> S_TimeSig(num, den, TimeSig)  
  | Instr(st) -> S_Instr(st, Instr)  
  | Tempo(i) -> S_Tempo(i, Tempo)  
  | Index(str, i) ->  
    let st = get_id_type str env in  
    let rl_int = (match i with IntLit(v) -> S_IntLit(v, Int)  
    | Id(nme) -> S_Id(nme, Int)) in  
    S_Index(str, rl_int, st)  
  | Binop(lft, op, rgt) ->  
    let l = confirm_expr lft env false in  
    let r = confirm_expr rgt env false in  
    let tp = confirm_binop l r op in  
    let lt = type_of_expr l in  
    let rt = type_of_expr r in  
    if lt = rt then S_Binop(l, op, r, tp)  
    else (match (lt, rt) with  
      Note, Int -> S_Binop(l, op, r, Note)  
    | _ -> S_Binop(l, op, r, Note)  
  | _ -> S_Binop(l, op, r, Note)
let mapval fu (arg: expr) = (* for array to be created *)
  let (nme, ag) =
  (match fu with
   FunkCall(i, e) -> (i, e)
   | _ -> raise(Failure("Error: Specified string in FuncList
       is not a valid function.")))
  let fn_decl = Table.get_decl nme env in
  let typ = (match arg with
    IntLit(i) -> Int
    | Note(_, _) -> Note
    | StringLit(_, _) -> String
    | Id(st) -> let v_decl = Table.get_decl st env in
    | _ -> raise(Failure("A function cannot be called
       on this type.")))
  id env in
  let ty_funk, _ = get_dt f_decl in ty_funk
  let t_obj, _ = get_dt v_decl in t_obj
  | _ -> raise(Failure("Error: Illegal function call
       on argument ")
  on argument "))
  if List.mem typ i it then
    (match dt with
     Null_Type ->
     IntLit(i) -> Int
     | Note(_, _) -> Note
     | StringLit(_, _) -> String
     | Id(st) -> let v_decl = Table.get_decl st env in
     | _ -> raise(Failure("Error: Illegal operation on illegal pair
       of types ")
     string_of_prim_type lt " and "
     string_of_prim_type rt))
  )
  let verify_type_and_vars tok =
    let nwvar = check_ex_list tok env in
    let nwtp = check_call_and_type nme nwvar env in
    nwvar in
    let verify_mod_expr tok = confirm_expr tok env false in
    let ags = verify_type_and_vars ag in
    let i_arg = verify_mod_expr arg in
    if List.mem typ it then
      (match dt with
       Null_Type ->
       IntLit(i) -> Int
       | Note(_, _) -> Note
       | StringLit(_, _) -> String
       | Id(st) -> let v_decl = Table.get_decl st env in
      )
let (t_obj, _, _) = get_dt var_dec in
t_obj

| Default -> Wild
| FunkCall(nme, arg_vals) -> Wild
| | Index(id, place) -> let var_dec = Table.get_decl
id env in

let (t_obj, _, _) = get_dt var_dec in t_obj
| Measure(_, _) -> Measurepoo
| Phrase(_, _) -> Phrase
| Song(_, _) -> Song
| _. -> raise(Failure("A function cannot be called on this type."))

in

let verify_type_and-vars tok =
let nwvar = check_ex_list tok env in
let nwtp = check_call_and_type nme nwvar env in
let verify_mod_expr tok = confirm_expr tok env false in
let ags = verify_type_and_vars arg in
let i_arg = verify_mod_expr arg in
if List.mem typ i then

(m -> S_Call(nme, i_arg, ags, i, dt)
| _. -> S_Noexpr)
else raise(Failure("Error: Illegal function call " ^ nme ^ " on an argument.

in

let l_calls = List.map2 mapval li fl in
let r_calls = List.map2 mapcall (List.rev li) fl in
let (i, ty) = check_arr fl env in
let ret = (match boo with
true -> S_DbArr(S_Call_lst(r_calls), S_Arr(l_calls, ty))
| false -> S_DbArr(S_Call_lst(r_calls), S_Noexpr)
)

in ret
| FunkCall(i, lis) ->
let arg_var = check_ex_list lis env in
let rt_typ = check_call_and_type i arg_var env in
let decl_f = Table.get_decl i env in
let (implicit_parm_type, explicit_param_types, arg_types) =
get_dt decl_f in
S_Call(i, (confirm_expr Default env false), arg_var,
explicit_param_types, rt_typ)
| Default -> S_Noexpr

and check_arr arr env =
match arr with
[|] -> ([], Null_Type) (* Empty *)
| head :: tail ->
let verified_head = confirm_expr head env false in
let head_type = type_of_expr verified_head in
let rec verify_list_and_type l t e = match l with
[|] -> ([], t)
| hd :: tl ->
let ve = confirm_expr hd e false in
let te = type_of_expr ve in
(ve :: (fst (verify_list_and_type tl te e)), t) in
(verified_head :: (fst (verify_list_and_type tail head_type env
)), head_type)

and check_ex_list (lst: expr list) env =
match lst with

(*) confirm correct function calls *)

and check_call_and_type name vars env =
let decl = Table.get_decl name env in (* function name in symbol table *)
let fdecl = match decl with
  FuncDecl(f) -> f (* check if it is a function *)
  _ -> raise (Failure ("Error: " ^ name ^ " is not a function.") in

if name = "print" then Int (* note returns wrong type *)
else if name = "write" then Wild (* note returns wrong type *)
else if name = "play" then Wild (* note returns wrong type *)
else if name = "evaluate" then Wild
else
  let (_,rtype,_,params,_) = fdecl in
  if (List.length params) = (List.length vars) then
    let arg_types = List.map type_of_expr vars in
    if params = arg_types then rtype
    else raise (Failure("Error: Argument types in " ^ name ^ " call do not match formal parameters."))
    else raise (Failure("Error: Function " ^ name ^ " takes " ^ string_of_int (List.length params) ^ " arguments, called with " ^ string_of_int (List.length vars)))

(* get the type of an id of a variable *)

and get_id_type den env =
let mark = Table.get_decl den env in
let var = match mark with
  VarDecl(sk) -> sk
  _ -> raise (Failure("Error: " ^ den ^ " is not a variable."))
in
let (.,_,tp,_) = var in
tp

(* convert AST statements into SAST statements *)

let rec confirm_stmt stmt ret_type env =
  (match stmt with
    Return(e) ->
      let verified_expr = confirm_expr e env false in
      S_Return (verified_expr)
    Expr(e) ->
      let verified_expr = confirm_expr e env false in
      S.Expr (verified_expr)
    VarDecl(mo) -> (match mo with
      Assign (typ, id, e) -> (* Verify that id is compatible type to e *)
      let ve = confirm_expr e env true in
      let eid_type = type_of_expr ve in
      if typ = eid_type
      then S.Assign (id, ve, typ)
      else raise (Failure("Error: Return type does not match " ^ string_of_prim_type typ ^ " string_of_prim_type eid_type " ^ "")
    | Update(st, ex) ->

let head :: tail -> confirm_stmt head env false :: check_ex_list tail env
let vid_type = get_id_type st env in
let de = confirm_expr ex env true in
let de_tp = type_of_expr de in
if de_tp = vid_type then S.Assign ( st , de , de_tp )
else raise ( Failure ( "Attempting to assign variable name " ^ st ^ " to value of type " ^ string_of_prim_type de_tp ^ " when " ^ st ^ " is already defined as a variable of type " ^ string_of_prim_type vid_type ^ ") )
| Index_Update ( expr_1 , expr_2 ) -> let type_1 = ( match expr_1 with
  Index ( str , exp ) -> let typ_known = Table.get_decl str env in
  let ( plz , typ , den ) = get_dt typ_known in plz
  | _ -> raise ( Failure ( "Error in matching index type") ) in
let iden = ( match expr_1 with
  Index ( str , exp ) -> str ) in
let idx = ( match expr_1 with
  Index ( str , exp ) -> exp ) in
let v_exp1 = confirm_expr idx env false in
let v_exp2 = confirm_expr expr_2 env false in
S.Index_Update ( iden , v_exp1 , v_exp2 , type_1 )
| If ( e , b1 , b2 ) -> let verified_expr = confirm_expr e env false in
if ( type_of_expr verified_expr ) = Int then
let vb1 = confirm_block b1 ret_type ( fst env , b1.block_id ) in
let vb2 = confirm_block b2 ret_type ( fst env , b2.block_id ) in
S.If ( verified_expr , S.CodeBlock ( vb1 ) , S.CodeBlock ( vb2 ) )
else raise ( Failure ( "Error: Condition in IF statement must be a boolean expression.") )
| While ( condition , block ) ->
let vc = confirm_expr condition env false in
let vt = type_of_expr vc in
if vt = Int then
let vb = confirm_block block ret_type ( fst env , block.block_id ) in
S.While ( vc , vb )
else raise ( Failure ( "Error: Condition in WHILE statement must be boolean expression.") )
| _ -> raise ( Failure ( "Error: Can’t map to statement.") )
(* iterates through a list of statements and confirms them *)
and confirm_stmt_list stmt_list ret_type env =
match stmt_list with
[ ] -> []
| head :: tail -> ( confirm_stmt head ret_type env ) :: ( confirm_stmt_list tail ret_type env )
(* function to confirm a block --- confirms each variable and statement *)
and confirm_block block ret_type env =
let verified_vars = map_to_list_env confirm_var block.locals ( fst env , block.block_id ) in
let verified_stmts = confirm_stmt_list block.statements ret_type env in
{ s_locals = verified_vars ; s_statements = verified_stmts ;
  s_block_id = block.block_id } 
(* goes through each fun, verifies block, arguments, and finally the declaration *)
let confirm_func func env =
let verified_block = confirm_block func.body func.ret_type (fst env, func.body.block_id) in
let verified_args = map_to_list_env confirm_var func.args (fst env, func.body.block_id) in
let verified_func_decl = confirm_func_decl func.fname env in
{ s.f_type = func.f_type; s.fname = verified_func_decl;
  s.ret_type = func.ret_type; s.formals = verified_args; s.fblock = verified_block }

(* SAST begins here – first function called: confirm_semantics *)

let confirm_semantics program env =
let main_stmts = traverse_main drop_funk (program.stmts) in
let main_vars = traverse_main get_vars main_stmts in
let g_var_val = List.filter (fun x -> x <> ("", false, Wild)) main_vars in
let verified_gvar_list = map_to_list_env confirm_var g_var_val env in
let main_func = confirm_func ({fname = "main"; ret_type = Null_Type; f_type = []; args = []; body = {locals = []; statements = List.rev main_stmts; block_id = 0}}) env in
let verified_func_list = main_func :: map_to_list_env confirm_func program.funcs env in
let () = prerr_endline "// Passed semantic checking \n" in
{ s.pfuncs = List.rev verified_func_list; s.gvars = List.rev verified_gvar_list}

Listing 7: sast.ml

Java Generator:

(* Java generator for Marmalade *)
open Ast
open Sast

(* rewrite AST types as the actual java types in the file. *)
let write_type = function
  | Int -> "j.int"
  | String -> "j.string"
  | Note -> "j.note"
  | Measurepoo -> "j.measure"
  | Phrase -> "j.phrase"
  | Song -> "j.song"
  | TimeSig -> "TimeSig"
  | Instr -> "int"
  | Tempo -> "int"
  | Intlist -> "j.intlist"
  | Stringlist -> "j.stringlist"
  | _ -> raise (Failure "Error: Type string of PD_Tuple or Null_Type being generated")

(* rewrite operations to their actual expressions in java. *)
let write_op Primitive op el e2 =
  match op with
    | Plus -> "new j.int(j.int.add(" + " e1 + ", " + e2 + "))"
let write_rhythm dr = match dr with 's' -> "0.125" (* sixteenth note maps to 0.125 *) | 'e' -> "0.25" (* eighth note maps to 0.25 *) | 'q' -> "0.5" | 'h' -> "1.0" | 'w' -> "2.0" (* get type of expression *) let rec get_typeof_dexpr = function S_Int_Lit(intLit, t) -> | 0 | S_String_Lit(strLit, t) -> | S_Id (str, t) -> | S_Arr(dexpr_list, t) -> | S_Binop(dexpr1, op, dexpr2, t) -> | S_Noexpr -> | S_Call (str, _, dexpr_list, _, t) -> (* write actual java compare expression *) let write_op_compared e1 op e2 = match op with Equal -> "(" + e1 + ").equals(" + e2 + ")" | Less -> "(" + e1 + ").compareTo(" + e2 + ")" + " < 0" | Leq -> "(" + e1 + ").compareTo(" + e2 + ")" + " <= 0" | Greater -> "(" + e1 + ").compareTo(" + e2 + ")" + " > 0" | Geq -> "(" + e1 + ").compareTo(" + e2 + ")" + " >= 0" | Neq -> "(" + e1 + ").compareTo(" + e2 + ")" + " != 0" | _ -> raise (Failure("Error: Not a comparator operation.")) (* convert marmalade's sax expressions into java expressions *) let rec write_expr = function S_Int_Lit(intLit, t) -> "(new j_int(" + string_of_int intLit + "))" | S_String_Lit(strLit, t) -> "(new j_string(" + strLit + "))" | S_Id (str, yt) -> str | S_Arr(dexpr_list, t) -> write_array_expr dexpr_list t | S_Binop(dexpr1, op, dexpr2, t) -> write_binop_expr dexpr1 op dexpr2 t | S_Db_Arr(call, mark) -> (match mark with | S_Arr(l_one, l_two) -> write_expr call | S_Noexpr -> write_expr call)
```java
| S_Measure(s_note_list, s_time, typ) -> "new j_measure(new
| j_note[" " (String.concat " " (List.map write_expr
| s_note_list)) "]}, new TimeSig(" ~ write_expr s_time " "))"
| S_Phrase(s_measure_list, s_instr, typ) -> "new j_phrase(new
| j_measure[" " (String.concat " " (List.map write_expr
| s_measure_list)) "]), " ~ write_expr s_instr " "))"
| S_Song(s_phrase_list, s_tempo, typ) -> "new j_song(new
| j_phrase[" " (String.concat " " (List.map write_expr
| s_phrase_list)) "]), " ~ write_expr s_tempo " "))"
| S_Noexpr -> ""
| S_Note(i, ch, tp) -> "new j_note(" ~ string_of_int i " " " ~
| write_rhythm ch " "")"
| S_TimeSig(i, i_2, tp) -> string_of_int i " " " ~
| string_of_int i_2
| S_Instr(str, tp) -> str
| S_Tempo(i, tp) -> string_of_int i
| S_Index(str, i, tp) -> str " ~ get(" ~ write_expr i " "")"
| S_Call(str, exp, dexpr_list, t_ret, t_send) -> (match str with
| "print" -> "System.out.println(" ~ write_expr exp
| " ~ "):\n" |
| "play" -> write_expr exp " ~ .play():\n" |
| "write" -> Write.midi(" ~ write_expr exp ~ .getObj
| (), \"out.midi\":\n" |
| "evaluate_measure" -> "new j_measure(" ~ String
| .concat " (List.map write_expr dexpr_list) " " ~
| "evaluate_phrase" -> "new j_phrase(" ~
| String .concat " (List.map write_expr dexpr_list) " ~
| "evaluate_song" -> "new j_song(" ~
| String .concat " (List.map write_expr dexpr_list) " ~
| "length_measure" -> "new j_int(" ~ String
| .concat " (List.map write_expr dexpr_list) " ~ .length())"
| "length_phrase" -> "new j_int(" ~ String
| .concat " (List.map write_expr dexpr_list) " ~ .length())"
| "length_song" -> "new j_int(" ~ String
| .concat " (List.map write_expr dexpr_list) " ~ .length())"
| "length_int_list" -> "new j_int(" ~
| String .concat " (List.map write_expr dexpr_list) " ~ .len()
| " length_string_list " -> "new j_int(" ~
| String .concat " (List.map write_expr dexpr_list) " ~ .length())"
| _ -> ( match exp with
| S_Noexpr -> str " ~ " ~ String .concat " ~ " (List
| .map write_expr dexpr_list)) " ~ " |
| _ -> write_expr exp " ~ ~ str " ~ " ~ String
| .concat " ~ " (List.map write_expr dexpr_list) " ~")://\n"
| S_Call_list(s) -> String.concat " ~ (List.map write_expr s)
| _ -> raise(Failure("Error: Not a valid expression.
| 
* this function matches to each kind of s_stmt. calling the
function write_expr to write each of them in Java. *)
```

S_Db_Arr(a1, a2) -> write_expr (S_Db_Arr(a1, a2))
write_assign name a2 t true ";\n" | false -> (match dexpr with
S_Db_Arr(a1, a2) -> write_expr (S_Db_Arr(a1, a2))
write_assign name a2 t false ";\n"
| S_Return(dexpr) -> "return " write_expr dexpr ";\n"
| S_If(dexpr, dstmt1, dstmt2) -> "if(" write_expr dexpr " )
write_stmt dstmt1 vg "else" write_stmt dstmt2 vg
| S_While(dexpr, dblock) -> "while(" write_expr dexpr " )
write_block dblock vg (* check true *)
| S_Index_Update(nme, expr1, expr2, typ) -> (match typ with
| JMusic_syntax for setting a note, measure, and part (which is the same as a phrase in marmalade) *)
| Measurepoo -> nme " . set_Note(" write_expr expr2 ") write_expr expr1 ");\n"
| Phrase -> nme " . set_Measure(" write_expr expr1 ");\n"
| Song -> nme " . set_Part(" write_expr expr1 ");\n"
| _ -> raise(Failure("is not a valid statement")))
and write_stmt_true d = write_stmt d true
and write_stmt_false d = write_stmt d false
(* function that matches the expression on each side of the binop, then writes it *)
and write_binop_expr expr1 op expr2 t =
  let e1 = write_expr expr1 and e2 = write_expr expr2 in
  let write_binop_expr_help e1 op e2 =
    match t with
    Int -> (match op with
      Plus | Minus | Times | Divide | Equal | Neq | Less | Leq | Greater | Geq | And | Or) ->
        write_op_primitive op e1 e2
      | String -> (match op with
        Plus -> "new j_string (j_string . add(" e1 ",
          " e2 "))")
        | (Equal | Less | Leq | Greater | Geq) ->
          write_op_comparise e1 op e2
          in
        raise(Failure(write_op_primitive op e1 e2 " is not a supported operation for String_Type"))
      | Note -> (match op with (Plus | Minus | Divide | Times) -> "new j_note (" write_op_primitive op e1 e2 ",", " e1 ".getLength())
          | _ -> raise(Failure("Error: Cannot add to note."))
          | _ -> raise(Failure("Error: " write_op_primitive op e1 e2 "," is not a supported operation for " write_type t ")
          in
        write_binop_expr_help e1 op e2
(* writes an array expression *)
and write_array_expr dexpr_list t =
  match t with
Int → "new j_intlist (new j_int []) {" ^ String.concat ","
  (* if Int, then write an int list *)
  (List.map write_expr dexpr_list) ^ "})"
| String → "new j_stringlist (new j_string []) {" ^ String.
  concat "," (* if String, then write a string list *)
  (List.map write_expr dexpr_list) ^ "})"
| _ → "new " ^ write_type t ^ " [" ^ " {" ^ String.
  concat "," (List.map write_expr dexpr_list) ^ " "})"

(* helper function to apply java toString function *)
and toString_str dexpr =
  let t = get_typeof_dexpr dexpr in
  match t with
  | Int → write_expr dexpr
  | String → write_expr dexpr
  | _ → "(" ^ write_expr dexpr ^ ")".toString()
and write_scope_var_decl_func svd =
  let (n , b , t , _ ) = svd in
  write_type t ^ " " ^ n
and write_scope_var_decl svd =
  write_scope_var_decl_func svd ^ ";\n"
and write_global_scope_var_decl gsvd =
  "static " ^ write_scope_var_decl_func gsvd ^ ";\n"

(* write assign expression in java *)
and write_assign name dexpr t vg =
  match vg with
  | true -> (match t with
    String | Instr | Tempo | Intlist | Stringlist -> name ^ " = "
    ^ write_expr dexpr
  | Int | Note | TimeSig | Measurepoo | Phrase | Song -> name ^ " = " ^ "(" ^ write_expr dexpr ^ ")")
  | _ → raise (Failure("" ^ write_type t ^ " is not a
    valid assign_type.")))
  | false -> (match t with
    String | Instr | Tempo | Intlist | Stringlist -> write_type t
    ^ " " ^ name ^ " = " ^ write_expr dexpr
  | _ → raise (Failure("" ^ write_type t ^ " is not a
    valid assign_type.")))
and write_block dblock vg =
  match vg with
  | true -> "{\n" ^ String.concat "\n" (List.map write_stmt_true
dblock.s_statements ) ^ "\n"
  | false -> "{\n" ^ String.concat "\n" (List.map write_scope_var_decl dblock.s_locals) ^ String.concat "\n" (List.map write_stmt_false dblock.s_statements ) ^ "\n"

(* include necessary java lines --> main *)
let write_func_wrapper x str =
  String.concat "\n"
  (let write_func dfunc =
    match (dfunc.s_fname , str) with
    ("main", String) -> "public static void main(String[] args)"
```
| (, _) -> | (String.concat \n (let match_type ftype =
  match ftype with
  str -> "static " write_type dfunc.s_ret_type ^ " " " dfunc.s_formals ^ " )" ^ write_block dfunc.s_fblock true
  | _ -> "" in
  List.map match_type dfunc.s_f_type) in
  List.map write_func x)

(* Below is necessary java placed into the file *)

let gen_pgm pgm name =
  "import java.util.Arrays;\n"
  "import java.util.ArrayList;\n"
  "import jm.JMC;\n"
  "import jm.music.data.*;\n"
  "import jm.util.*;\n"
  "import marmalade.*;\n"
  "import jm.midi.event.TimeSig;\n"

  "public class " ^ name ^ " implements JMC{\n  "String.concat "
  List.map write_global_scope_var_decl pgm.s_gvars ) ^
  (write_func_wrapper pgm.s_pfuncs String) ^
  "\n"

  "public static class j_int extends m_Int {\n  "public j_int(int n) {\n  "super(n);\n"
  "public j_int(j_int n) {\n  "super(n);\n"
  "(write_func_wrapper pgm.s_pfuncs Int) ^
  "\n"

  "public static class j_intlist extends m_Int.List {\n  "public j_intlist(j_int [] j) {\n  "super(j);\n"
  "public j_int get(int i) {\n  "return new j_int(getList()[i]);\n"
  "(write_func_wrapper pgm.s_pfuncs IntList) ^
  "\n"

  "public static class j_string extends m_String {\n  "public j_string(j_string x) {\n  "super(x);\n"
  "public j_string(String x) {\n  "super(x);\n"
  "(write_func_wrapper pgm.s_pfuncs String) ^
  "\n"

  "public static class j_stringlist extends m_String.List {\n  "public j_stringlist(j_string [] j) {\n  "super(j);\n"
  "public j_string get(int i) {\n  "return new j_string(getList()[i]);\n"
  "(write_func_wrapper pgm.s_pfuncs StringList) ^
  "\n"
```
public static class j_note extends m_Note {
    public j_note(Note n) 
        super(n);
    public j_note(int pitch, double length) 
        super(pitch, length);
    (write_func_wrapper pgm.s_pfuncs Note) 
}

public static class j_measure extends Measure {
    public j_measure(j_note[] m, TimeSig n) 
        super(m, n);
    public j_measure(Phrase p) 
        super(p);
    public j_measure(j_measure l) 
        super(l.getObj());
    public j_note get(int i) 
        Note n = getObj().getNote(i); 
        j_note m = new j_note(n);
    public void setNote(j_note i, j_note n) 
        this.p.setNote(i.getObj(), n);
    public void setNote(j_note i, int k) 
        this.p.setNote(i.getObj(), k);
    (write_func_wrapper pgm.s_pfuncs Measurepoo) 
}

public static class j_phrase extends 
    m_Phase 
{
    public j_phrase(Part p) 
        super(p);
    public j_phrase(j_measure[] m, int n) 
        super(m, n);
    public j_phrase(j_phrase[] m, j_measure n) 
        super(m, n);
    public j_phrase(j_phrase l) 
        super(l.getObj());
    public j_measure get(int i) 
        Phrase p = getObj().getPhrase(i);
    public void setMeasure(j_int idx, j_measure n_measure) 
        this.setMeasure(idx.get(), (Measure) n_measure); 
    (write_func_wrapper pgm.s_pfuncs Measurepoo) 
}


(Listing 8: javagen.ml)

Script to Test Java:

```bash
# make java.sh
cd javaclasses
javac -cp ./jMusic1.6.4.jar: marmalade/m_Int.List.java
javac -cp ./jMusic1.6.4.jar: marmalade/m_String.java
javac -cp ./jMusic1.6.4.jar: marmalade/m_String.List.java
javac -cp ./jMusic1.6.4.jar: marmalade/m_Note.java
javac -cp ./jMusic1.6.4.jar: marmalade/m_Measure.java
javac -cp ./jMusic1.6.4.jar: marmalade/m_Phrase.java
javac -cp ./jMusic1.6.4.jar: marmalade/Song.java
javac -cp ./jMusic1.6.4.jar: marmalade/SongTempo.java
javac -cp ./jMusic1.6.4.jar: marmalade/SongPhrase.java
javac -cp ./jMusic1.6.4.jar: marmalade/Tester.java
jar cvf marmalade.jar marmalade/*/class
java -cp ./jMusic1.6.4.jar: marmalade/Tester
cd ..
```

(Listing 9: script to make java)

Script to Run Test Suite:

```bash
#!/bin/bash
# run_tests.sh
# Based on MicroC Regression Test Suite Script (microc/testall.sh)
```
# 0 stdin
# 1 stdout
# 2 stderr

MARMALADE="/marmac"
    #marmac depends on "marmalade" compiler

# Set time limit for all operations
ulimit -t 30

globallog=tests.log
rm -f $globallog
error=0
globalerror=0
keep=0

Usage() {
    echo "Usage: run_tests.sh [options] [..marm files]"
    echo "-k Keep intermediate files"
    echo "-h Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
        echo "FAILED"
        error=1
        fi
        echo "$1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile. Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    diff -b "$1" "$2" "$3" 1>&2
    cat "$1" >$2
diff -b "$1" "$2" "$3" 2>&1 || {
    SignalError "$1 differs"
    echo "$1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* 1>&2
    SignalError "$1 failed on $*$
    return 1
}

Check() {
    # $1 name of basename file (i.e. test_arith_add1)
    # $2 name of testdir (i.e. testdir_2015-11-24_061339)
```
error=0
basename='echo $1 | sed 's/\.*///
 s/\./\marm\///'
reffile='echo $1 | sed 's/\.*marm$///'
basedir="'echo $1 | sed 's/\[/\[\]/\]/\marm$///'

echo -n "$basename..."

echo "# # # # # # Testing $basename" 1>&2
generatedfiles=""

# GENERATE .java .class <outfile>.t.out .t.diff FILES

generatedfiles="$2/$basename.t.out"
Run "$MARMALADE" "$1" "$basename"
mv "$basename" "$2/$basename.t.out"
mv "$basename.java" "$2/$basename.java"
mv "$basename.class" "$2/$basename.class"
mv *.class "$2/"

Compare $2/$basename.t.out $reffile.out $2/$basename.t.
diff
echo
generatedfiles="$generatedfiles $2/$basename.t.out $2/$reffile.out $2/$basename.t.
diff"

echo

generatedfiles="$generatedfiles $2/$basename.t.out $2/$reffile.out $2/$basename.java $2/$basename.class"

# Report the status and clean up the generated files
if [ $error -eq 0 ] ; then
if [ $keep -eq 0 ] ; then
# rm -f $generatedfiles
    echo ""
fi
echo "OK"
echo "# # # # # # SUCCESS" 1>&2
else
echo "# # # # # # FAILED" 1>&2
globalerror=$error
fi

#BEGINNING OF SCRIPT
#BEGINNING OF SCRIPT
#BEGINNING OF SCRIPT
while getopts kdpsh c ; do
case $c in
k) # Keep intermediate files
    keep=1
    ;;
h) # Help
```
Usage

; ;

esac
done

shift ‘expr $OPTIND - 1’

if [ $# -ge 1 ]
then
  files=$(0)
else
  files="tests/fail_* tests/test_*"
  # files="tests/fail_* .marm tests/test_* .marm"
fi

# AUTO ARCHIVE TEST FILES
if [ ! -d "testing_archive" ]; then
  mv testdir_* testing_archive/
else
  mkdir "testing_archive"
fi

# CREATE NEW TEST DIR FOR INTERMEDIATE FILES
date='date +%Y%m%d%H%M%S'
testdir="testdir_${date}"
mkdir "$testdir"

for file in $files
do
case $file in
  *.out)
    Check $file $testdir 2>> $globallog
    ;;
  *.test*)
    Check $file $testdir 2>> $globallog
    ;;
  *.fail*)
    CheckFail $file $testdir 2>> $globallog
    ;;
  *)
    echo "unknown file type $file"
    globalerror=1
    ;;
esac
done

exit $globalerror

Listing 10: script to run test suite
Makefile:

```makefile
# Makefile for marmalade compiler

OBJS = ast.cmo table.cmo sast.cmo parser.cmo scanner.cmo javagen.cmo marmalade.cmo

TESTS = \
YACC = ocamlyacc

marmalade : $(OBJS)
  ocamlc -o marmalade $(OBJS)

scanner.ml : scanner.mll
  ocamllex scanner.mll

parser.ml parser.mli : parser.mly
  $(YACC) parser.mly

%.cmo : %.ml
  ocamlc -c $<

%.cmi : %.mli
  ocamlc -c $<

.PHONY : clean
clean :
  rm -f marmalade parser.ml parser.mli scanner.ml \
   *.cmo *.cmi *.out *.diff

ast.cmo:
  ast.cmo

ast.cmx:
  ast.cmx

sast.cmo: ast.cmo
  sast.cmx: ast.cmx

javagen.cmo: ast.cmo
  javagen.cmx: bytecode.cmx ast.cmx

marmalade.cmo: scanner.cmo parser.cmi compile.cmo
  marmalade.cmx: scanner.cmx parser.cmx compile.cmx

parser.cmo: ast.cmo parser.cmi
  parser.cmx: ast.cmx parser.cmi

scanner.cmo: parser.cmi
  scanner.cmx: parser.cmx

parser.cmi: ast.cmo
```

Listing 1: Makefile for marmalade

Programs:

```c
/* gcd and Fibonacci algorithm */

/* fibonacci number algorithm */

/* recursive algorithm for calculating nth fibonacci number */

funk int int fib(int n, int val_1, int val_2)
```
{ 
   if (n <= 2 or n <= 0){
      return 1;
   }
   else{
      val_1 = $fib(n-1, 0, 0);
      val_2 = $fib(n-2, 0, 0);
      n = val_1 + val_2;
      return n;
   }
} /* gcd algorithm */

funk int int gcd(int a, int b){
   while (a != b){
      if (a > b) {
         a = a - b;
      } else {
         b = b - a;
      }
   }
   return a;
}

/* prints the gcd and factorial */

(print(), print()) [gcd(30, 90), fib(10, 0, 0)];

Listing 2: a function implementing and testing gcd and fibonacci algorithms

/* 99 bottles of beer in marmalade */

int offset = 0;
int current_bottle = 99;
int next_bottle = 98;

while (offset < 98) {
   current_bottle = current_bottle - 1;
   next_bottle = next_bottle - 1;
   (print(), print()) [ current_bottle, " bottles of beer on the wall "];
   (print(), print()) [ current_bottle, " bottles of beer. Take one down, pass it around "];
   (print(), print()) [ next_bottle, " bottles of beer on the wall."
   offset = offset + 1;
Listing 3: script that prints 99 bottles of beer

```plaintext
/* 99 bottles of beer */

measure t_1 = $(6:8) [67.e, 67.e, 67.e, 62.e, 62.e];
measure t_2 = $(6:8) [67.e, 67.e, 67.e, 67.h];
measure t_3 = $(6:8) [69.e, 69.e, 69.e, 64.e, 64.e];
measure t_4 = $(6:8) [69.h, 0.e, 0.e, 67.e];
measure t_5 = $(6:8) [65.e, 65.e, 65.e, 62.e, 62.e];
measure t_6 = $(6:8) [65.e, 65.e, 65.e, 65.e, 65.e];
measure t_7 = $(6:8) [62.e, 62.e, 62.e, 62.e, 64.e, 65.e];
measure t_8 = $(6:8) [67.e, 67.e, 67.e, 67.h];

phrase ph1 = $(HARP) [t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8];
phrase ph2 = $(HARP) [t_2, t_3, t_4, t_5, t_6, t_7, t_8];
song s1 = $(60) [ph1];

(int offset = 0);

while (offset < 30)
{
    offset = offset + 15;
    (print(), play(), print(), play()) ["Original song:", s1, ", " Transposed song", $transpose_song(s1, offset)];
}

while (offset < 60)
{
    offset = offset + 15;
    (print(), play(), print(), play()) ["Piano:", $(PIANO) [t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8], "Harp:", ph2];
}

funk song song transpose_song_w(song s, int n, int counter, int j, phrase k, song g)
{
    j = $length_song(s);
    counter = 0;
    g = $evaluate_song(s);
    while (counter < j)
    {
        k = $kcounter;
        g&counter = $transpose_phrase(k, n);
        counter = counter + 1;
    }
    return g;
}
funk song song transpose_song(song s, int n)
```

60
Listing 4: script that implements transpose methods for all musical objects and uses it to play 99 bottles of beer transposed in different ways

```plaintext
/* Reptilia.marm */

measure a_1 = $(4:4) [47.e, 47.e, 47.e, 47.e, 47.e, 47.e, 47.e, 47.e];
measure a_2 = $(4:4) [47.e, 47.e, 47.e, 47.e, 47.e, 47.e, 47.e, 47.e];
measure a_3 = $(4:4) [52.e, 52.e, 52.e, 52.e, 52.e, 52.e, 52.e, 52.e];
measure a_4 = $(4:4) [52.e, 52.e, 52.e, 52.e, 52.e, 52.e, 52.e, 52.e];
```

```plaintext
{ return $transpose_song_w(s, n, 0, 0, $$()[$(44.q)], $$()[$(44.q)]); }
}

tunk phrase phrase transpose_phrase_w(phrase p, int n, int counter, int j, measure k, phrase h){
    j = $length_phrase(p);
    counter = 0;
    h = $evaluate_phrase(p);
    while (counter < j)
    {
        k = p&counter;
        h&counter = $transpose_measure(k, n);
        counter = counter + 1;
    }
    return h;
}

tunk phrase transpose_phrase(phrase p, int n)
{
    return $transpose_phrase_w(p, n, 0, 0, $$()[$(44.q)], $$()[$(44.h)]);
}

tunk measure transpose_measure_w(measure m, int n, int counter, int j, note k, measure l)
{
    j = $length_measure(m);
    counter = 0;
    l = $evaluate_measure(m);
    while (counter < j)
    {
        k = l&counter;
        l&counter = k + n;
        counter = counter + 1;
    }
    return l;
}

tunk measure transpose_measure(measure m, int n)
{
    return $transpose_measure_w(m, n, 0, 0, 44.q, $$()[$(55.h)]);
}
```
Listing 5: script which plays Reptilia by the Strokes

```bash
/* Script which plays a remix of Clocks by Coldplay */

measure r_1 = $(4:4) [0.e, 0.e, 0.e, 0.e, 0.e, 0.e, 0.e];
measure b_1 = $(4:4) [50.e, 50.e, 50.e, 50.e, 50.e, 50.e, 50.e];
measure b_2 = $(4:4) [50.e, 50.e, 50.e, 50.e, 50.e, 50.e, 50.e];
measure b_3 = $(4:4) [55.e, 55.e, 55.e, 55.e, 55.e, 55.e, 55.e];
measure b_4 = $(4:4) [55.e, 55.e, 55.e, 55.e, 55.e, 55.e, 55.e];
measure r_2 = $(4:4) [0.e, 0.e, 0.e, 0.e, 0.e, 0.e, 0.e];
measure c_1 = $(4:4) [62.e, 62.e, 62.e, 59.e, 0.e, 57.e, 0.e];
measure c_2 = $(4:4) [62.e, 62.e, 62.e, 59.e, 0.e, 57.e, 0.e];
measure c_3 = $(4:4) [56.e, 56.e, 0.e, 56.e, 59.e, 0.e, 62.e];
measure c_4 = $(4:4) [56.e, 56.e, 0.e, 56.e, 59.e, 0.e, 62.e];
measure r_3 = $(4:4) [0.e, 0.e, 0.e, 0.e, 0.e, 0.e, 0.e];
phrase ph_01 = $(BASS) [a_1, a_2, a_3, a_4, r_1, r_1, r_1, r_1, a_1, a_2, a_3, a_4, r_1, r_1, r_1, r_1];
phrase ph_02 = $(BASS) [r_2, r_2, r_2, b_1, b_2, b_3, b_4, r_2, r_2, r_2, b_1, b_2, b_3, b_4];
phrase ph_03 = $(PIANO) [c_1, c_2, c_3, c_4, r_3, r_3, r_3, r_3];
phrase ph_04 = $(PIANO) [r_3, r_3, r_3, r_3, c_1, c_2, c_3, c_4, r_3, r_3, r_3, c_1, c_2, c_3, c_4, r_3, r_3, r_3, c_1, c_2, c_3, c_4];
song reptilia = $(80) [ph_01, ph_10, ph_02, ph_11, ph_22];
( play() ) [ reptilia ];
```

Listing 6: script which plays a remix of Clocks by Coldplay
measure s.2 = $ (4:4) [62.w];
measure s.3 = $ (4:4) [60.w];
measure w.1 = $ (4:4) [60.s, 60.s, 60.s, 60.s, 60.s, 60.s, 60.s, 60.s, 60.s, 60.s, 60.s, 60.s, 60.s];
measure rest.1 = $ (4:4) [0.w];

phrase ph.1.0 = $(PIANO) [c_1, c_2, c_3, c_4, rest_1, rest_1, rest_1, rest_1, c_1, c_2, c_3, c_4, rest_1, rest_1, rest_1];
phrase ph.2.0 = $(BASS) [b_1, b_2, b_1, b_2, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1];
phrase ph.3.0 = $(TIMPANI) [t_1, t_2, t_2, t_3, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1];
phrase ph.4.0 = $(TENOR_SAX) [s_1, s_2, s_2, s_3, rest_1, rest_1, rest_1, rest_1, s_1, s_2, s_2, s_3, rest_1, rest_1, rest_1];
phrase ph.5.0 = $(PIPES) [w_1, w_1, w_1, w_1, rest_1, rest_1, rest_1, rest_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1];

phrase ph.1.1 = $(PIANO) [rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, c_1, c_2, c_3, c_4, rest_1, rest_1, rest_1, rest_1];
phrase ph.2.1 = $(BASS) [rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, b_1, b_1, b_1, b_1, b_2, b_2, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1];
phrase ph.3.1 = $(TIMPANI) [rest_1, rest_1, rest_1, rest_1, t_1, t_2, t_2, t_2, t_2, t_2, t_2, t_2, t_3, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1, rest_1];
phrase ph.5.1 = $(PIPES) [rest_1, rest_1, rest_1, rest_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1, w_1];

song clocks = $(80) [ph.1.0, ph.2.0, ph.3.0, ph.4.0, ph.5.0, ph.1.1, ph.2.1, ph.3.1, ph.5.1];
( play( ) ) [ clocks ];

Listing 6: script which plays Clocks