CMajor
A Music Production Language
Contents

1 Introduction 4
  1.1 Background .............................................. 4

2 Language Tutorial 6
  2.1 Installation & Compilation ................................. 6
  2.2 Compose with CMajor ..................................... 7

3 Language Reference Manual 9

4 Project Plan 10
  4.1 Project Process ........................................... 10
    4.1.1 Planning ............................................ 10
    4.1.2 Specification ...................................... 10
    4.1.3 Development ....................................... 10
    4.1.4 Testing ............................................ 10
  4.2 Style Guide .............................................. 10
  4.3 Project Timeline ......................................... 11
  4.4 Roles and Responsibilities ............................... 11
  4.5 Development Environment ................................ 11
  4.6 Project Log .............................................. 12

5 Architectural Design 13
  5.1 Components .............................................. 13
    5.1.1 Scanner ............................................ 13
    5.1.2 Parser .............................................. 13
    5.1.3 Compiler & Analyzer ................................. 13
  5.2 Interfaces .............................................. 14

6 Test Plan 14
  6.1 Testing Phases ........................................... 14
    6.1.1 Unit Testing ........................................ 14
    6.1.2 Integration Testing ................................. 14
    6.1.3 System Testing ..................................... 14
  6.2 Examples ................................................ 14
  6.3 Test Suites .............................................. 15
    6.3.1 Motivation .......................................... 15
    6.3.2 Automation ......................................... 15

7 Lessons Learned 15
  7.1 Andrew O’Reilly ........................................... 15
  7.2 Stephanie Huang .......................................... 15
  7.3 Jonathan Sun ............................................ 16
  7.4 Laura Tang ............................................... 16

8 Appendix 16
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Source Code</td>
<td>16</td>
</tr>
<tr>
<td>8.2 Demos</td>
<td>42</td>
</tr>
</tbody>
</table>
1 Introduction

CMajor is a procedural, imperative language used to create musical compositions for playback on a MIDI device. It provides a set of types for abstracting time and frequency components of Western music, as well as a notation for referencing frequencies and pitches commonly employed. CMajor outputs the results of a composition to CSV bytecode, later to be interpreted by a Java program and rendered via MIDI playback. In addition to data types which correspond to the duration and pitch aspects of sound, it provides structured types which allow a programmer composer to organize pitches into sequences and to layer them into chords and phrases, giving them control over harmonic properties of musical composition as well as its melodic ones. Users of the language may also take advantage of familiar programming constructs such as loops and conditional statements, allowing them to easily repeat precomposed phrases, reuse previously composed structures, and conditionally alter the course of a composition based on number of repetitions or whatever conditions they choose to supply. CMajor possesses a C-style syntax, consisting of lists of expressions separated by semicolons, each of which return types that can be operated upon according to the rules of the language. Programmers may additionally write their own functions to modify pitches or return composed elements.

1.1 Background

Perhaps one of the most fascinating aspects of music is that its generation entails complex mathematical calculations, and that these calculations can be made by a performer and perceived by listeners regardless of their mathematical abilities. Further, the twelve tone western system further abstracts these calculations by classifying different frequencies as named “pitches” and uses a system of fractions to describe durations. In this system, frequencies, here called pitches, are given a letter and optionally a modifier to raise their frequency. Each key on a piano is made to strike and therefore vibrate, a different string, each tuned to one of these frequencies. The keys of a piano therefore provide a useful diagram for the arrangement of named pitches according to their frequency:
The frequencies named above increase from left to right. The pitch that corresponds to the key that would appear immediately to the right of the "B / Cb" key would be named "B# / C" along with the one that appears furthest to the left in the above diagram. This not-pictured pitch has a frequency equal to twice that of the pitch to the far left; the one after it has a frequency equal to twice that of the the one corresponding to the next pictured key (C# / Db), and so on. The perceived difference between any two pitches is called an interval, and in the case where the perceived difference is between a pitch and its corresponding one of doubled frequency, the interval is called an octave. Pitches an octave apart share a certain auditory quality and are easily identified, and so the pattern of keys on a piano repeats itself:

In CMajor, pitches are identified according to the naming scheme employed in the first picture (a pitch with two names may be referred to by either one) and by their octave number, with higher numbers referring to octaves with higher frequencies. CMajor further takes inspiration from Western music in its classification of pitch duration as a basis for rhythm. All durations are stored as a pair of integers, which represent the numerator and denominator of a rational number. The actual duration in milliseconds depends upon the number of beats per minute, and further upon the definition of a beat Western music tends to be flexible in this regard, but here a beat is defined as 1/4, or a quarter note, and the beats per minute is set to 120:
Above are a few durations that may be constructed, although any pair of integers may be used to do so. By using a combination of a pitch and a duration, a note may be constructed, and a sequence of notes may be used to create a song. Notes may also be played simultaneously to create harmonies and interlocking rhythms that add complexity to a composition. Two other important aspects of Western musical composition, timbre and volume, were not considered for this project and are left to a future implementation.

2 Language Tutorial

2.1 Installation & Compilation

To install CMajor, run make from the root directory after extracting the tar file. This will build and compile all the components needed for the CMajor compiler. Next, write your CMajor composition in a file ending in .cmaj. Example CMajor programs can be found in the tests/directory, as well as the demo/directory.
Compile your CMajor program by running: ./cmajor scale.cmaj
This will generate two output files: an out.csv file and a play.out file. out.csv is an intermediate "bytecode" file, comparable to .class files generated by the Java compiler. play.out is the executable file (a generated shell script), which can be executed with the following command in order to play the music composed:

./play.out
Alternatively, out.csv can be manually played by executing the following command using the included CSVPlayer:

java CSVPlayer out.csv

2.2 Compose with CMajor

Every CMajor file (.cmaj) is a self contained piece of music that contains the functions, sequences of statements, and control structures necessary to describe and play that piece of music. All musicians write music by composing. They listen to music by playing. This gives us our two most important functions in CMajor: compose() and play().

Every CMajor file must contain a compose() function, and if the piece is to be played, must call the built-in play() function. A simple CMajor program to play the single note middle C is shown below:

```cmaj
int compose() {
    // call play on a note literal
    play((C, (1,4));
}
```

After compiling, we will get an out.csv and play.out file. These files are only generated when play() is called within the compose() function.

CMajor uses C-like syntax with function return types in the function declarations, explicit typing, brackets enclosing blocks of code, and semicolon line endings.

In the example above, we use the two methods of commenting. Inline/single line comments using // and multilne comments enclosed by /* */. We also create a note literal, which is represented by a tuple of pitch and duration. Pitch literals are simple note letters preceded by $. Octave and sharps and flats can also be utilized (ex. $C#4 is a C# in octave 4, or the note a half step above middle C).

CMajor supports common control structures such as ifelse statements and for loops. You can also write your own functions. Example syntax is shown below:

```cmaj
// for loops
int i; // note that i is initialized outside the loop
for (i = 0; i < 4; i = i + 1) {
```
CMajor has some special types that make music writing easier. These include pitch and duration
types, as well as structural types like notes, chords, phrases, and scores.

One key feature of CMajor is the special music related operators that we have. In particular, we
feature the layer (\(^\)) operator, which allows for creation of notes, chords, phrases, and scores from
layers of types such as pitches and durations (to form a phrase), or multiple phrases to form a
score. We also have an array concatenation operator (++) and repeater operator for replication of
structural types such as notes or chords (**).

The Language Reference Manual is a self-contained document in the following pages.
C-Major Language Reference Manual

1. Expressions

An expression is a series of tokens that return a value. They consist of one or more literals and zero or more operators. Expressions are grouped according to their operators (if present) and evaluated according to operator precedence. One or more expressions may be combined at terminated with a semicolon (;) to form an expression statement, or separated by commas (,) to form a list for use in function calls. A list of expressions of variable size make up the body of blocks, which are delimited by braces ({ }). An array of expressions separated by the comma (,) character may be used to populate an array.

\[
\text{stmt_list} \rightarrow \text{stmt_list} \text{ stmt} | \epsilon \\
\text{stmt} \rightarrow \text{expr;}
\]

\[
\text{actualls_list} \rightarrow \text{expr} | \text{actualls_list, expr}
\]

Basic expressions consist of one or more identifiers (see Lexical Conventions) and zero or more operators. An identifier may be a literal or a variable.

\[
\text{expr} \rightarrow \text{expr op expr}
\]

Assignment expressions assign the value returned by an expression to an identifier. The type of value returned by the expression must match the type of the variable represented by the identifier.

\[
\text{expr} \rightarrow \text{id = expr}
\]

Function calls consist of an identifier followed by an open parenthesis, followed by an expression array. The return value of the expression is the return value of the function.

\[
\text{expr} \rightarrow \text{id(expr_array)}
\]
2. Data Types

2.1 Primitive Types

There are two primitive types in C-Major, int and pitch, upon which all other types in the language are built.

2.1.1 Int

Represents a whole number.

2.1.2 Pitch

Pitch represents a musical pitch, typically an integer that maps to an index on the piano keys (0-88). It is stored internally as an integer. The default pitch is 40 (C4).

2.2 Non-Primitive/Structural Types

2.2.1 Array

An array type has the format t[] where t is a type that specifies the type of all elements of the array. Thus, all elements of an array of type t[] must themselves have type t. Note that t itself may be an array type.

Arrays can be initialized as an array literal of type literals:

```plaintext
int[] array = [1,2,3,4,5];
```

2.2.2 Tuple

A tuple is a pair of elements within parenthesis separated by a comma. Each element can be a different type.

2.2.3 Duration
A duration is tuple of integers. The ratio of the first element to the second element represents the fraction of a whole note the associated pitch will play.

### 2.2.4 Note

A note is a tuple consisting of a pitch and a duration. The pitch must be in the left element.

( pitch, duration )

### 2.2.5 Chord

A chord is a tuple wherein the left element is an array of pitches, and the right element is a duration type element. All pitches in the array will be played for the duration specified by the second element.

( pitch[], duration )

### 2.2.6 Phrase

A phrase is an array of chords. This would represent a single line or voice of music in a piece. Every note will start and end individually; there are no overlaps. A second voice should be designated with a separate phrase. A variable of type phrase may be initialized to or otherwise assigned the value of an expression whose type is a chord array.

chord[]

### 2.2.7 Score

A Score is an array of phrases. Each element points to a single phrase which would represent the multiple voices of a single piece. A variable of type score may be initialized to or assigned the value of an expression whose type is an array of phrases.

phrase[]

### 3. Operators

#### 3.1 Assignment Operator =
As previously stated, the assignment operator is denoted by the equals sign =.

### 3.2 Comparison Operators

Comparison operators are used to test for equality or inequality between identifiers or literals. A expression consisting of a comparison operator and two other expressions return an integer type whose value is 1 where the assertion is true and 0 where it is false. All comparison operators test the value of their identifiers. The return type of each expression being operated on by comparison operators must be the same. The greater-than, greater-than-or-equal-to, less-than, or less-than-or-equal-to operators (> , >= , < , and <= , respectively) may be used with the following types:

```c
int
pitch
duration
```

The equality and inequality operators (== and !=, respectively) may additionally be used with the note type.

<table>
<thead>
<tr>
<th>Production rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr → expr == expr</td>
<td>Evaluates to 1 if the return values of the expressions in the production body are equivalent, and 0 otherwise.</td>
</tr>
<tr>
<td>expr → expr != expr</td>
<td>Evaluates to 1 if the return values of the expressions in the production body are not equivalent, and 0 otherwise.</td>
</tr>
<tr>
<td>expr → expr &gt; expr</td>
<td>Evaluates to 1 if the expression on the left is greater in return value than the return value of expression on the right, and 0 otherwise.</td>
</tr>
<tr>
<td>expr → expr &lt; expr</td>
<td>Evaluate to 1 if the expression on the right is greater in return value than the return value of expression on the right, and 0 otherwise.</td>
</tr>
<tr>
<td>expr → expr &gt;= expr</td>
<td>Evaluates to 1 if the expression on the left is greater in return value than the expression on the right, or if the return values of the expressions are equal, and 0 otherwise.</td>
</tr>
</tbody>
</table>
expr \rightarrow \text{expr} \leq \text{expr} \quad \text{Evaluates to 1 if the expression on the right is greater in return value than the expression on the left, or if the return values of the expressions are equal, and 0 otherwise.}

The inequality of integers is evaluated according to the standard ordering of integers from negative infinity to infinity. In evaluations of pitch types, their inequality is evaluated according to their frequency or the position of their corresponding keys on a piano—pitches that correspond to keys towards the right end of the piano are greater than pitches that correspond to keys on the left. The inequality of durations is evaluated according to a standard ordering of rational numbers from 0 to infinity.

### 3.3 Arithmetic Operators

Arithmetic operators are binary operators and consist of addition (+), subtraction (-), multiplication (*), and division (/). The return type of expressions involving arithmetic operators depends upon the return type of the expressions in the operation. Addition and subtraction are commutative.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Left expression type</th>
<th>Right expression type</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+</td>
<td>int</td>
<td>int</td>
<td>The sum of the two integers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pitch</td>
<td>int</td>
<td>A pitch raised the number of half steps indicated by the integer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dur</td>
<td>int</td>
<td>A duration. The integer is converted to a fractionally equivalent duration. The durations are then added according to fractional arithmetic. ((1,2) + 1 = (3,2))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dur</td>
<td>dur</td>
<td>The sum of the two durations according to fractional arithmetic,</td>
</tr>
<tr>
<td>Operation</td>
<td>Operator</td>
<td>Type 1</td>
<td>Type 2</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>--------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>int</td>
<td>int</td>
<td>The product of the two integers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dur</td>
<td>int</td>
<td>The product of the fractional value of the duration and the integer, reduced to its least possible denominator. $(1,4) \times 2 \text{ yields } (1,2)$.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dur</td>
<td>dur</td>
<td>The fractional product of the two durations. $(1,4) \times (1,2) \text{ yields } (1,8)$.</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>int</td>
<td>int</td>
<td>The difference between the left integer and the right integer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pitch</td>
<td>int</td>
<td>A pitch lowered by the number of half steps specified by the integer expression.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dur</td>
<td>int</td>
<td>A duration whose length is the result of the fractional subtraction of right integer converted to a fraction from the fractional value of the left duration expression. If the result is negative, the absolute value is returned. $(5,4) - 1 = (1,4)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pitch</td>
<td>pitch</td>
<td>An integer representing the difference between the two pitches, in scale positions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>chord</td>
<td>pitch</td>
<td>A chord with the right-expression pitch removed, if it was present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dur</td>
<td>dur</td>
<td>A duration whose length is equal to the fractional subtraction of the right duration from the left. $(1,2) - (1,4) = (1,4)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>note</td>
<td>dur</td>
<td>A note whose duration is equal to the subtraction of the right duration from the duration of the left note expression.</td>
</tr>
<tr>
<td>operation</td>
<td>operands</td>
<td>types</td>
<td>description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>-------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>chord / dur</td>
<td>chord, dur</td>
<td>int, int</td>
<td>A chord whose duration is equal to the subtraction of the right duration from the duration of the left note expression.</td>
<td></td>
</tr>
<tr>
<td>Division /</td>
<td>int, int</td>
<td>int</td>
<td>A duration whose numerator is equal to the left integer and whose denominator is equal to the right.</td>
<td></td>
</tr>
<tr>
<td>dur / int</td>
<td>dur, int</td>
<td>int</td>
<td>A duration whose fraction is equal to the fractional division of the fractional component of the left expression by the integer value of the right expression.</td>
<td></td>
</tr>
<tr>
<td>note / int</td>
<td>note, int</td>
<td>int</td>
<td>A note whose duration is equal to the division of the duration of the note in the left expression divided by the integer value of the right expression, as described above.</td>
<td></td>
</tr>
<tr>
<td>chord / int</td>
<td>chord, int</td>
<td>int</td>
<td>A chord whose duration is equal to the division of the duration of the chord in the left expression divided by the integer value of the right expression, as described above.</td>
<td></td>
</tr>
<tr>
<td>int / dur</td>
<td>int, dur</td>
<td>int</td>
<td>A duration whose fractional component is equal to the fractional division of the integer by the the fractional value of the duration.</td>
<td></td>
</tr>
<tr>
<td>dur / dur</td>
<td>dur, dur</td>
<td>int</td>
<td>Fractional division of durations.</td>
<td></td>
</tr>
<tr>
<td>note</td>
<td>dur</td>
<td>A note whose duration is equal to the fractional division of the left expression’s duration component by the right expression’s duration.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-----</td>
<td>----------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>chord</td>
<td>dur</td>
<td>A chord whose duration is equal to the fractional division of the left expression’s duration component by the right expression’s duration.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dur</td>
<td>note</td>
<td>A note whose duration is equal to the fractional division of the left duration by the duration component of the note in the right expression.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dur</td>
<td>chord</td>
<td>A chord whose duration is equal to the fractional division of the left duration by the duration component of the note in the right expression.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dur</td>
<td>chord</td>
<td>A chord whose duration is equal to the fractional division of the left duration by the duration component of the note in the right expression.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 3.4 Repeater Operator - `**`

Supplying an expression or any type followed by the repeater operator (`**`) and a subsequent integer yields an array of size equal to the given integer with each element containing the return value of the expression:

\[
\text{expr} \rightarrow \text{expr} \; ** \; \text{int}
\]

### 3.5 Concatenation Operators (`+`, `++)`

When used exclusively with notes, chords, and phrases, the + symbol is used as a concatenation operator. The use of the + operator with any combination of notes, chords, and phrases returns a phrase type.

\[
\text{expr} \rightarrow \text{expr} + \text{expr}
\]

The left expression is appended to the beginning of the right within the resulting phrase. All notes and chords are then intended to be read and/or played from left to right.

The ++ concatenation operator is used for array concatenation and always returns an array of the base type of its operands. One or both operands may be an array whose base type matches the base type of the other. The result is an array wherein the right expression is appended to the end of the left.

### 3.6 Layer Operator (^)

The layer operator is used to create musical structures wherein pitches are played simultaneously. It is a binary operator and its behavior is only defined for the pitch, note, chord, phrase, and score types.

\[
\text{expr} \rightarrow \text{expr} \, ^\text{expr}
\]

A pitch may be layered with a duration to form a note. An array of n pitches may be layered with an array of n durations to return an array of n notes, wherein the \(i^{th}\) note of the resulting array consists of the pitch at index \(i\) in the pitch array and the duration at index \(i\) in the duration array. Pitches may also be layered with chords, and in this instance a chord is returned with the pitch added. In all other cases a score is returned. When rendered, the arguments are synchronized by their beginning; if one argument has a longer total duration than the other, it continues playing after the shorter argument has completed. The layer operator is commutative.

### 3.7 Operator Associativity and Precedence
The layer operator is applied first, followed by the arithmetic operators - in the standard order of *, /, -, +. Boolean operators are applied next, followed by the repeater operator, the array concatenation operator, and finally the assignment operator.

4. Lexical Conventions

4.1 Comments

Comment syntax is similar to Java. Single line comments are preceded by //. Multiline comments are enclosed with /* and */. For example:

```plaintext
// Single line comment

/*
 * Multiline
 * comment
 * here
 */
```

4.2 Identifiers

An identifier names functions and variables and consists of a sequence of alphanumeric characters and underscores (_) in the set ['a'-'z' 'A'-'Z' '_' '0'-'9']. Identifiers are case-sensitive and must begin within a character within the set ['_' 'a'-'z' 'A'-'Z'].

4.3 Keywords

The following keywords are reserved:

<table>
<thead>
<tr>
<th>chord</th>
<th>dur</th>
<th>else</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>for</td>
<td>if</td>
</tr>
<tr>
<td>int</td>
<td>note</td>
<td>null</td>
</tr>
<tr>
<td>phrase</td>
<td>pitch</td>
<td>play</td>
</tr>
<tr>
<td>print</td>
<td>return</td>
<td>score</td>
</tr>
<tr>
<td>true</td>
<td>void</td>
<td></td>
</tr>
</tbody>
</table>
4.4 Constants/Literals

Integer literals

Integer literals are of type int and are of the form ['0'-9']

Pitch Literals

Pitch literals are of type pitch and are of the form '$' ['A'- 'G'] ['# ' 'b']? ['0'-9']?
The capital letter corresponds to the note name, '#' and 'b' denote sharp or flat, and the
integer denotes which octave the note is in. If '#' or 'b' is omitted, a natural pitch is
assumed. If an octave integer is omitted, octave 4 is assumed, or the octave of the set
key (see more on setting keys later on). For example, $C4 denotes C in octave 4, or
middle C.

A rest literal is a specific pitch literal that represents a rest. (No pitch.) It is represented
as $R

Duration Literals

A duration literal is of type dur and is a 2-tuple of integers that correspond to note
durations used in music. It is of the form '(' ['1'-9'], ['1'-9]+ ')'.
For example, a quarter note can be represented as the duration literal (1,4).

Note Literals

A note literal is of type note and is a 2-tuple of pitch and duration of the form '(' ('$'
['A'- 'G'] ['# ' 'b']? ['0'-9']? | "$R")'; '(' ['1'-9'], ['1'-9]+ ')')'

Chord Literals

A chord literal is of type chord and is a 2-tuple of an array of pitches and duration. It is of
the form '(' '[' ('$' ['A'- 'G'] ['# ' 'b']? ['0'-9']?)* | "$R")'; '(' ['1'-9'], ['1'-9]+ ')')'

4.5 Separators

Separators separate tokens and expressions. White space is a separator. Other separators are
tokens themselves:

( ) { } [ ] ; , . < >

4.6 White Space

White space consists of the space character, tab character, and newline character. White space
is used to separate tokens and is ignored other than when used to separate tokens. White
space is not required between operators and operands or other separators. Any amount of white space can be used where one space is required.

5. Statements

5.1 Expression Statements

Any expression can become a statement by terminating it with a semicolon.

5.2 Declaration and Initialization Statements

Giving a type name keyword followed by an identifier terminated with a semicolon yields a statement that allocates memory for a variable of the given type. Optionally, the assignment operator may be supplied followed by an expression prior to the semicolon in order to initialize the variable to a value. The value to which the variable is initialized is the return value of the expression to the right of the assignment operator. As with the assignment expression, the type of the variable and the type of the value to which it is initialized must match.

5.3 if/else

An if / else statement has the following structure:

```plaintext
if (expr) {
    stmt_list
}
else if (expr) {
    stmt_list
}
else {
    stmt_list
}
```
The expression in parentheses must evaluate to true or false. If true, then the if block is executed. Otherwise, the statement is tested. The else block is executed when no conditional expression evaluates to true.

5.4 for

A for statement (for loop) has the following structure:

```c
for (asn; expr1; expr2) {
    stmt_list
}
```

First, `asn` is evaluated. `asn` is traditionally an assignment expression. Next, `stmt_list` is evaluated if `expr1` evaluates to true. `expr2` is executed after `stmt_list`, and the condition in `expr1` is checked again. This repeats until `expr1` evaluates to false and the for statement is exited.

5.5 return expr;

The return statement evaluates `expr` and returns program control to the function that called it, and returns the evaluated value of `expr` into the higher level function. The type of `expr` must be the same as declared in the function definition.

6. Functions

6.1 Defining Functions

Function definitions have the form:

```
type declarator compound-statement
```

The `type` specifies the return type. A function can return any type. The declarator in a function declaration must specify explicitly that the declared identifier has a function type; that is, it must be of the form

```
direct-declarator ( expr_array )
```
The form and its parameters, together with their types, are declared in its parameter type list; the declaration-list following the function’s declarator must be absent. Each declarator in the parameter type list must contain an identifier.

A parameter-type-list is a list of expressions separated by commas. The parameters are understood to be declared just after beginning of the compound statement constituting the function’s body, and thus the same identifiers must not be redeclared there (although they may, like other identifiers, be redeclared in inner blocks). An example:

```c
int max(int a, int b) {
    if (a > b) return a;
    else return b;
}
```

Here int is the declaration specifier; max(int a, int b) is the function’s declarator, and { … } is the block giving the code for the function.

### 6.2 Calling Functions

A function call is an identifier followed by parentheses containing a possibly empty, comma-separated list of assignment expressions which constitute the arguments to the function, or an expression array. The term argument is used for an expression passed by a function call; the term parameter is used for an input object (or its identifier) received by a function definition, or described in a function declaration.

In preparing for the call to a function, a copy is made of each argument; all argument-passing is strictly by value. A function may change the values of its parameter objects, which are copies of the argument expressions, but these changes cannot affect the values of the arguments. The types of parameters are explicit and are part of the type of the function - this is the function prototype. The arguments are converted, as if by assignment, to the types of the corresponding parameters of the function’s prototype. The number of arguments must be the same as the number explicitly described parameters. Recursive calls to any function are permitted.

### 6.3 The play Function

The identifier play is reserved to let the compiler make MIDI calls in Java. Play takes either a score type expression or phrase type expression. It returns an integer: 0 on success, 1 for failure.
6.4 The *compose* Function

Every *C-Major* program must define the reserved identifier *compose*. The expression bound to *compose* is evaluated and its value is the value of the *C-Major* program itself. That is, when a *C-Major* program is compiled and run, the expression bound to *compose* is evaluated and the result is converted to a value of type score or int. If a definition for *compose* is not included, or the expression bound to it does not evaluated to *score*, a compile-time error will occur.
4 Project Plan

4.1 Project Process

4.1.1 Planning

We had a one hour meeting every Monday with all members in attendance. These meetings were led by our manager Andrew, and we discussed project milestones including what we would accomplish within the following week and made updates to the CMajor language design. Further into the project, we averaged two to three meetings per week where we would discussed implementation that overlapped between members and any debugging that needed to be done.

4.1.2 Specification

For the Proposal and LRM, we outlined what was to be included during our weekly meetings, and assigned sections to different members. Led by Jonathan, who composed a larger chunk of the reports and made the final transposition into LaTeX, each member wrote their own sections and proofread the documents individually, making edits as needed.

4.1.3 Development

While developing our language, Andrew and Stephanie were the primary authors of the scanner, parser, and analyzer/code generator code. Usually coding was done individually, and then reviewed by peers after submitting pull requests to a master branch of our project on a GitHub repository. Language features such as arrays, control structures, operators, and types were divided up and implemented independently of each other, so that we had a working compiler early on and simply expanded it out by adding features. This made testing much easier as well.

4.1.4 Testing

Testing was accomplished using the test suite written by Laura (described later on in detail in this report). We made sure to test during the development process, especially when implementing new features. Before and after merging every pull request that implemented a new feature, we would run the test suite to make sure any conflicts or bugs were resolved. We wrote sample test cases specific to the features being implemented while implementing those features.

4.2 Style Guide

All code was implemented using Unix line endings and spaces for indentation. Function bodies and other nested blocks of code were indented with two spaces. Lines were broken and indented two spaces when lines were longer than 84 characters. Match statements were similarly implemented, with the -> operator on the same line as the match case and subsequent lengthy code on following lines.
4.3 Project Timeline

<table>
<thead>
<tr>
<th>Date</th>
<th>Milestone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sep. 30, 2015</td>
<td>Proposal Due</td>
</tr>
<tr>
<td>Oct. 23, 2015</td>
<td>LRM Outlined</td>
</tr>
<tr>
<td>Oct. 25, 2015</td>
<td>LRM Drafted</td>
</tr>
<tr>
<td>Oct. 26, 2015</td>
<td>LRM Proofread</td>
</tr>
<tr>
<td>Oct. 26, 2015</td>
<td>LRM Due</td>
</tr>
<tr>
<td>Oct. 31, 2015</td>
<td>Scanner and Parser</td>
</tr>
<tr>
<td>Nov. 12, 2015</td>
<td>Semantic Analyzer</td>
</tr>
<tr>
<td>Nov. 16, 2015</td>
<td>Hello World Due</td>
</tr>
<tr>
<td>Dec. 16, 2015</td>
<td>Scanner and Parser Completed</td>
</tr>
<tr>
<td>Dec. 17, 2015</td>
<td>Semantic Analyzer Completed</td>
</tr>
<tr>
<td>Dec. 18, 2015</td>
<td>Testing Completed</td>
</tr>
<tr>
<td>Dec. 18, 2015</td>
<td>Presentation Due</td>
</tr>
<tr>
<td>Dec. 21, 2015</td>
<td>Code Cleanup</td>
</tr>
<tr>
<td>Dec. 22, 2015</td>
<td>Final Report Due</td>
</tr>
</tbody>
</table>

Commit Graph:

4.4 Roles and Responsibilities

Andrew O’Reilly  Manager
Stephanie Huang  System Architect
Jonathan Sun    Language Guru
Laura Tang      Testing Suite

4.5 Development Environment

CMajor has been tested and built in both OS and Windows 8 (running cygwin and an Ubuntu virtual machine) environments. Git was used for version control. Text editors used include vim, Sublime Text, and Notepad++. Most of the CMajor language (scanner, parser, compiler) was written in OCaml, utilizing features such as ocamlex. In order to generate sound written by
CMajor programs, we also used Java 7 with the javax.sound.midi library. Testing suites, Makefiles, and final output files utilize bash/shell scripting.

4.6 Project Log

(See following pages)
<table>
<thead>
<tr>
<th>Hash</th>
<th>Author</th>
<th>Date</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>4b4447c</td>
<td>Andrew</td>
<td>Tue Dec 22</td>
<td>Merge pull request #60 from phanieste/compiler</td>
</tr>
<tr>
<td>fb8d3f8</td>
<td>Stephanie</td>
<td>Tue Dec 22</td>
<td>writes executable play.out file to play out.csv</td>
</tr>
<tr>
<td>19a331d</td>
<td>Stephanie</td>
<td>Tue Dec 22</td>
<td>add authors to source code files</td>
</tr>
<tr>
<td>3812bda</td>
<td>Jonathan</td>
<td>Tue Dec 22</td>
<td>Compile Java with make. Move java files to root</td>
</tr>
<tr>
<td>3d16371</td>
<td>Andrew</td>
<td>Tue Dec 22</td>
<td>(Convert tabs to spaces</td>
</tr>
<tr>
<td>17e8d93</td>
<td>Andrew</td>
<td>Tue Dec 22</td>
<td>(Clean up comments</td>
</tr>
<tr>
<td>71921c1</td>
<td>Jonathan</td>
<td>Sat Dec 19</td>
<td>:gitignore *.toc</td>
</tr>
<tr>
<td>b31d5f0</td>
<td>Jonathan</td>
<td>Sat Dec 19</td>
<td>:Additional .gitignore</td>
</tr>
<tr>
<td>eba4e78</td>
<td>Jonathan</td>
<td>Sat Dec 19</td>
<td>:Initial final report latex template.</td>
</tr>
<tr>
<td>0da6088</td>
<td>Andrew</td>
<td>Fri Dec 18</td>
<td>:Lengthen shepard</td>
</tr>
<tr>
<td>81bd6fc</td>
<td>Stephanie</td>
<td>Fri Dec 18</td>
<td>:implement layer for score with phrase</td>
</tr>
<tr>
<td>7711edf</td>
<td>Andrew</td>
<td>Fri Dec 18</td>
<td>:Add demos</td>
</tr>
<tr>
<td>592c2f2</td>
<td>Stephanie</td>
<td>Fri Dec 18</td>
<td>:play works with single note</td>
</tr>
<tr>
<td>f2b3513</td>
<td>Andrew</td>
<td>Fri Dec 18</td>
<td>:Move up precedence of repeater operator</td>
</tr>
<tr>
<td>7c8df08</td>
<td>Stephanie</td>
<td>Fri Dec 18</td>
<td>:fix compile warnings with incomplete match case</td>
</tr>
<tr>
<td>ff588ae</td>
<td>Stephanie</td>
<td>Fri Dec 18</td>
<td>:fixes csv output and pitch ordering</td>
</tr>
<tr>
<td>623fbc4</td>
<td>Stephanie</td>
<td>Fri Dec 18</td>
<td>:add note+note=phrase implementation</td>
</tr>
<tr>
<td>d2c5e08</td>
<td>Laura</td>
<td>Fri Dec 18</td>
<td>:fix failure report test.sh</td>
</tr>
<tr>
<td>84ee8cb</td>
<td>Laura</td>
<td>Fri Dec 18</td>
<td>:update failure report</td>
</tr>
<tr>
<td>bcda10</td>
<td>Laura</td>
<td>Fri Dec 18</td>
<td>:tests folder renaming</td>
</tr>
<tr>
<td>6bc98d3</td>
<td>Stephanie</td>
<td>Fri Dec 18</td>
<td>:fix runtime errors in play</td>
</tr>
<tr>
<td>f509204</td>
<td>Laura</td>
<td>Fri Dec 18</td>
<td>:renamed tests</td>
</tr>
<tr>
<td>df8fb9f</td>
<td>Stephanie</td>
<td>Fri Dec 18</td>
<td>:implements multi-note play</td>
</tr>
<tr>
<td>36a67f1</td>
<td>Laura</td>
<td>Fri Dec 18</td>
<td>:test.sh updated with failure reports</td>
</tr>
<tr>
<td>3c34776</td>
<td>Laura</td>
<td>Fri Dec 18</td>
<td>:test.sh updated to compare to .outs</td>
</tr>
<tr>
<td>b34507c</td>
<td>Laura</td>
<td>Fri Dec 18</td>
<td>:added .outs</td>
</tr>
<tr>
<td>8f2be0d</td>
<td>Andrew</td>
<td>Thu Dec 17</td>
<td>:Add tests</td>
</tr>
<tr>
<td>2c4fde4</td>
<td>Andrew</td>
<td>Thu Dec 17</td>
<td>:Implement if/else</td>
</tr>
<tr>
<td>17fadce</td>
<td>Andrew</td>
<td>Thu Dec 17</td>
<td>:Add tests</td>
</tr>
<tr>
<td>15988fa</td>
<td>Andrew</td>
<td>Thu Dec 17</td>
<td>:Implement for loops</td>
</tr>
<tr>
<td>cf66fb8</td>
<td>Andrew</td>
<td>Thu Dec 17</td>
<td>:Add tests</td>
</tr>
<tr>
<td>79d88dc</td>
<td>Andrew</td>
<td>Thu Dec 17</td>
<td>:Fix block statement processing</td>
</tr>
<tr>
<td>3906ec</td>
<td>Stephanie</td>
<td>Thu Dec 17</td>
<td>:fixes issue #42</td>
</tr>
<tr>
<td>b1c1512</td>
<td>Andrew</td>
<td>Thu Dec 17</td>
<td>:Implement changes in execute.ml</td>
</tr>
<tr>
<td>9256fd4</td>
<td>Andrew</td>
<td>Thu Dec 17</td>
<td>:Fix field names in compile.ml</td>
</tr>
<tr>
<td>ba4c86b</td>
<td>Andrew</td>
<td>Thu Dec 17</td>
<td>:Fix typos in function definitions</td>
</tr>
<tr>
<td>837c8b1</td>
<td>Andrew</td>
<td>Thu Dec 17</td>
<td>:Add missing arguments</td>
</tr>
<tr>
<td>ac49964</td>
<td>Andrew</td>
<td>Thu Dec 17</td>
<td>:Implement find_var for environments and symtabs</td>
</tr>
<tr>
<td>9180c7d</td>
<td>Andrew</td>
<td>Thu Dec 17</td>
<td>:Implement update_var function</td>
</tr>
</tbody>
</table>
Laura Thu Dec 17 :modified test.sh testing script
Andrew Thu Dec 17 :More descriptive exception message
Andrew Thu Dec 17 :Compiles without warnings
Andrew Thu Dec 17 :Resolve last compile.ml warning
Andrew Thu Dec 17 :Resolve most match/unused case warnings in comp
Andrew Thu Dec 17 :Clear several match warnings
Andrew Thu Dec 17 :Change subtraction ops to return correct types
Andrew Thu Dec 17 :Suppress unmatched case warnings
Andrew Thu Dec 17 :Implement -
Andrew Wed Dec 16 :Add tests
Andrew Wed Dec 16 :Implement * (multiply)
Andrew Wed Dec 16 :Implement **
Andrew Wed Dec 16 :Merge branch 'phanieste-compiler-pitchfix' into
don
Andrew Wed Dec 16 :Additional fixes
Stephanie Wed Dec 16 :remove sign from pitch literal
Stephanie Wed Dec 16 :Implement + operator for notes
Stephanie Tue Dec 15 :Implement + operator for dur + int
Stephanie Tue Dec 15 :Implement + operator for pitch
Stephanie Tue Dec 15 :Fix merge conflicts between compiler and compil
Jonathan Tue Dec 15 :Cleanup & whitespace issues.
Andrew Tue Dec 15 :Add tests for >=
Andrew Tue Dec 15 :Implement <=
Andrew Tue Dec 15 :Implement <
Andrew Tue Dec 15 :Implement >=
Andrew Tue Dec 15 :Add tests for >
Andrew Tue Dec 15 :Update gcd function to handle 0
Andrew Tue Dec 15 :Fix typo
Andrew Tue Dec 15 :Modify gcd function to handle negative numbers
Andrew Tue Dec 15 :Implement > operator in syntactically correct f
Andrew Tue Dec 15 :Add dur_sub function for comparators
Andrew Tue Dec 15 :Add tests for != operator
Andrew Tue Dec 15 :Implement != operator
Andrew Tue Dec 15 :Implement == operator
Andrew Tue Dec 15 :Fix missing Vdecl case for arrays
Andrew Tue Dec 15 :Update tests
Andrew Tue Dec 15 :Fix compile errors
Andrew Mon Dec 14 :Fix line endings
a30746e Andrew Mon Dec 14 :Fix line endings

d08c422 Laura Mon Dec 14 :JONATHAN: Implement rests (silent notes). Remove line endings

7e8ee7f Jonathan Sat Dec 12 :Clean newlines

6a49a48 Jonathan Sat Dec 12 :CSVPlayer can play multiple lines independently

03aab2e Andrew Fri Dec 11 :Implement rests (silent notes). Remove java dependant code.

d9c8058 Andrew Fri Dec 11 :Convert line endings to Unix

be601d2 Andrew Fri Dec 11 :Add test and expected output for issue #32

dae7eb8 Andrew Thu Dec 3 :Try using separate parser rule to fix pitch issue. Make translate recursive to expose exec_fun function

b8b3942 Stephanie Tue Dec 8 :Implement ++ for concatenating two single elements

02a77de Stephanie Tue Dec 8 :Implement layer operator with pitch[]^dur[]

f4091af Stephanie Mon Dec 7 :Implement array concatenation operator (++)

bb438be Stephanie Sat Dec 5 :fix bug with assigning score and phrase types

b1391bd Stephanie Sat Dec 5 :Implement layer operator

f23005c Stephanie Fri Dec 4 :Implement array set operation

c1a49ad Andrew Fri Dec 4 :Fix #32 parsing issues with pitch literals

aac4850 Stephanie Fri Dec 4 :Implement array get operation

44f6f87 Andrew Thu Dec 3 :Try using separate parser rule to fix pitch issue

0c3fe22 Stephanie Wed Dec 2 :Implement phrases and scores

3e0e6c7 Stephanie Wed Dec 2 :Implement chords

072ac0e Stephanie Wed Dec 2 :Implement array type checking

98b61c2 Stephanie Wed Dec 2 :Declare arrays using typename[] syntax

f3b78c Stephanie Wed Dec 2 :Basic array literal creation and array type

6b57379 Stephanie Mon Nov 30 :Fix merge conflicts merging compiler into compilerW arrays

61f3545 Andrew Sun Nov 29 :Implement Call expression

a8bdcc6 Andrew Sun Nov 29 :Add Missing_function exception

d2d2091 Stephanie Sun Nov 29 :Add arrays to parser and ast

5ad5706 Stephanie Sun Nov 29 :Manually merge and fix compilation errors in parser

b95a157 Andrew Sun Nov 29 :Update exec_fun to return environment

cd460df Andrew Sun Nov 29 :Reorganize code

65b3803 Andrew Sun Nov 29 :Update toplevel test to test return

e388f18 Andrew Sun Nov 29 :Implement return statement

5e92323 Andrew Sun Nov 29 :Implement function to get string from s_type

359086b Andrew Sun Nov 29 :Update tests to use new syntax

b86de3c Andrew Sun Nov 29 :Add script to convert existing tests to new syntax

9b16265 Andrew Sun Nov 29 :Make translate recursive to expose exec_fun function

b8bf861 Andrew Sun Nov 29 :Update global environment data type

bb070d Andrew Sun Nov 29 :Remove option from globals type

8e4e288 Andrew Fri Nov 27 :Change global environment var map to allow empty

a24d2bb Andrew Fri Nov 27 :Add global_environment type
Andrew Fri Nov 27 : Fix compile syntax errors
Andrew Fri Nov 27 : Implement fxn call
Andrew Fri Nov 27 : Implement code to create global environment
Andrew Fri Nov 27 : Update AST program type to be two lists
Andrew Fri Nov 27 : Update parser/compiler to accept top level synt
Andrew Sun Nov 29 : Add division test
Andrew Thu Nov 26 : Fix basic test syntax
Andrew Thu Nov 26 : Fix missed variable rename
Andrew Thu Nov 26 : Update execute module to handle new types
Andrew Thu Nov 26 : Update main executable to handle new types
Andrew Thu Nov 26 : Add List.rev to block
Andrew Thu Nov 26 : Update compiler to use returned values instead
Andrew Thu Nov 26 : Fix syntax errors in test.cmaj
Andrew Tue Nov 24 : More spacing/readability/code style
Andrew Tue Nov 24 : More spacing/readability/code style
Andrew Tue Nov 24 : Spacing/readability/code style
Andrew Tue Nov 24 : Remove excess parentheses
Andrew Tue Nov 24 : Implement divide operator for chords
Andrew Tue Nov 24 : Implement divide Binop
Andrew Tue Nov 24 : Remove excess parens from dur_divide fxn
Andrew Tue Nov 24 : Add dur_divide function
Andrew Tue Nov 24 : Change Pitch to 3 ints instead of 3 Ints
Andrew Tue Nov 24 : Change Dur to be two ints instead of 2 Ints
Andrew Tue Nov 24 : Fix syntax errors
Andrew Tue Nov 24 : Add Binop to expr function
Laura Thu Nov 19 : Made test folder and tests
Andrew Wed Nov 18 : Finish play function
Stephanie Wed Nov 18 : Start writing play() function
Andrew Wed Nov 18 : Add missing line to compile
Andrew Tue Nov 17 : Add new stmt type
Stephanie Mon Nov 16 : Fix some variable declaration errors. * Use Li
Stephanie Mon Nov 16 : Switch env to hashtable
Stephanie Mon Nov 16 : Attempt to fix variable declaration struggles
Jonathan Mon Nov 16 : Fix ocamldep issues.
Stephanie Mon Nov 16 : Add execute.ml
Jonathan Mon Nov 16 : Generate additional Makefile lines with ocamlde
Stephanie Mon Nov 16 : Merge pull request #8 from phanieste/compiler
Stephanie Mon Nov 16 : Fixed syntactical and logical compilation error
Andrew Mon Nov 16 : Fix scanner ID/Typename conflict
Stephanie Mon Nov 16: Fix compilation errors in sast with c_type to s_type.

Stephanie Mon Nov 16: Start building out working hello world version.

Jonathan Sun Nov 15: Ignore java *.class files.

Jonathan Sun Nov 15: Rename 'readCSV' to 'play.' Add print statement.

Jonathan Sun Nov 15: Remove duplicate file.

Jonathan Sun Nov 15: Twinkle CSV example.

Stephanie Fri Nov 13: Fix merge conflicts in semantics.ml

Stephanie Fri Nov 13: Start compiler.

Jonathan Sun Nov 15: Ignore java *.class files.

Jonathan Sun Nov 15: Rename 'readCSV' to 'play.' Add print statement.

Jonathan Sun Nov 15: Twinkle CSV example.

Stephanie Fri Nov 13: Fix merge conflicts in semantics.ml

Stephanie Fri Nov 13: Start compiler.

Andrew Fri Nov 13: Resolve merge conflict in sast.

Andrew Fri Nov 13: Add Makefile.

Andrew Fri Nov 13: Update PITCH_SIGN token in scanner to mitigate.

Andrew Fri Nov 13: Add code to read cmajor code from stdin.

Andrew Fri Nov 13: Fix typo in scanner.

Andrew Fri Nov 13: Fix parser compile errors.

Andrew Fri Nov 13: Remove mergetool garbage.

Andrew Fri Nov 13: Fix syntax errors.

Andrew Fri Nov 13: Initial implementations of exec.

Andrew Fri Nov 13: Fixing AST for temporarily simplified program s.

Andrew Thu Nov 12: Removing mergetool garbage.

Andrew Thu Nov 12: Fixing conflicts with origin branch.

Andrew Thu Nov 12: Fix syntax errors in compiler.

Andrew Thu Nov 12: Fix additional syntax errors in SAST.

Andrew Thu Nov 12: Fix syntax errors.

Andrew Thu Nov 12: Fix undefined constructor in AST.

Andrew Thu Nov 12: Implement variable lookup and update routines.

Andrew Thu Nov 12: Add semantic types for checking.

Andrew Mon Nov 9: Add main compiler file.

Andrew Mon Nov 9: Change duration literal to tuple.

Andrew Mon Nov 9: Simplifying test prog further.

Andrew Mon Nov 9: Adding test program.

Andrew Mon Nov 9: Basic test grammar - no functions.

Andrew Thu Nov 12: Adding Steph's initial SAST.

Andrew Mon Nov 9: Add main compiler file.

Andrew Mon Nov 9: Change duration literal to tuple.

Andrew Mon Nov 9: Simplifying test prog further.

Andrew Mon Nov 9: Adding test program.

Andrew Mon Nov 9: Basic test grammar - no functions.

Stephanie Wed Nov 11: Began sast and semantic analysis (pitch validation).

Stephanie Mon Nov 9: Modify pitch literals in scanner and parser to
Stephanie  Sun Nov 6 11:28:07 2015  add precedence/associativity to fix shift/reduce

Stephanie  Sun Nov 6 10:49:00 2015  fix typos in scanner and parser

Andrew   Tue Nov 3 17:28:12 2015  Add PITCH_LIT ($) token

Andrew   Tue Nov 3 17:25:06 2015  Add production rules

Andrew   Tue Nov 3 17:16:15 2015  Add regexes for if

Andrew   Tue Nov 3 17:15:16 2015  Add tokens for if

Andrew   Tue Nov 3 17:12:28 2015  Add grouping operators described in previous cc

Andrew   Tue Nov 3 17:11:48 2015  Add return

Andrew   Tue Nov 3 17:00:09 2015  Fix comments in parser

Andrew   Mon Nov 2 23:34:30 2015  Add program start symbol

Andrew   Mon Nov 2 23:34:30 2015  Fix syntax errors

Andrew   Mon Nov 2 23:34:30 2015  Add program start symbol

Andrew   Sat Oct 31 00:11:34 2015  Fix syntax errors

Andrew   Sat Oct 31 17:21:50 2015  Add/fix comments

Andrew   Sat Oct 31 16:55:46 2015  Add vdecl type for variable declarations

Andrew   Sat Oct 31 16:26:34 2015  Fix regex quoting

Andrew   Sat Oct 31 16:00:09 2015  Add operators to AST

Andrew   Sat Oct 31 15:35:52 2015  Very unfinished AST

Andrew   Sat Oct 31 15:00:09 2015  Initial commit
5 Architectural Design

5.1 Components

5.1.1 Scanner

The scanner is implemented in scanner.ml and identifies language tokens using regular expressions.

5.1.2 Parser

Implemented in parser.mly. Creates the abstract syntax tree and passes it off as such to the Semantic Analyzer / Code Generator.

5.1.3 Compiler & Analyzer

Performs a dual task of semantic analysis and the storage of environment information in memory.
5.2 Interfaces

Within the compiler, all information is stored in OCaml data types and records, including a linkedlist symbol table which stores the contents of variables, a global environment type that stores global variables and a mapping of function definitions to names, and an environment type that stores a symbol table, a global environment, a return type for the current function, and its return value (if set by a return statement). Once the program has been processed, if the program calls the play() function, a CSV is output containing an intermediate form to be read by the CSVPlayer.

Andrew and Stephanie were primarily involved in the implementation of compiler components. Jonathan devised many of the language features and details. He created the CSVPlayer in java and detailed the format of the CSV files output by the compiler and read by the CSVPlayer.

6 Test Plan

6.1 Testing Phases

6.1.1 Unit Testing

Unit testing was done as language features were being completed during the coding phase of the project. Whenever a feature was added, multiple tests were run to ensure these basic blocks were parsed correctly.

6.1.2 Integration Testing

Once the unit testing was finished, the integration testing confirmed the correctness of semantic analysis and code generation.

6.1.3 System Testing

The entire endtoend testing of the language framework is the final testing phase. The CMajor compiler takes in an input program written in the language and produces an output file of the environment which is compared against the expected output of listed pitches and durations. An optional bytecode file and accompanying executable play.out file (if the play() function is present in the input program) is also generated and tested against the reference bytecode file. Finally, to test the bytecode, the play executable can be run. This utilizes the Java MIDI Player programs to produce the correct sounds, which we can listen to to ensure correct output. An output log is generated to list all the output of the test suites that are run, and a failure log is generated to list the error messages thrown for failed tests.

6.2 Examples

See following pages
int compose() {
    pitch[] pitches = $C ** 3 ++ $D ++ $E ++ $E ++ $E ++ $D ++ $E ++ $F ++ $G ++ $C ** 3 ++ $G ** 3 ++ $E ** 3 ++ $C ** 3 ++ $G ++ $F ++ $E ++ $D ++ $C;

dur dot8 = (3,16);
dur trip8 = (1,4) / 3;

dur[] durations = (1,4) ** 2 ++ dot8 ++ (1,16) ++ (1,4) ++ dot8 ++ (1,16) ++ (1,2) ++ trip8 ** 12 ++ dot8 ++ (1,16) ++ dot8 ++ (1,16) ++ (1,2);

phrase mainphrase = pitches ^ durations;

note rest = ($R, (1,1));

int i;
score song = newscore();

for(i = 0; i < 4; i = i + 1) {
    int j;
    phrase round = mainphrase;
    for(j = 0; j < i; j = j + 1) {
        round = rest + round;
    }
    song = song ^ round;
}

play(song);
}

chord newchord(dur d) {
    note n1 = ($R,d);
    return n1 ^ $R;
}

phrase newphrase() {
    chord c1 = newchord((0,1));
    chord c2 = c1;
    return c1 + c2;
}

score newscore() {
    phrase p1 = newphrase();
    return p1 ^ p1;
}

-x-

out of rowyourboat.cmaj:

dot8 = Dur(3,16)
durations = Array(Dur(1,4),Dur(1,4),Dur(3,16),Dur(1,16),Dur(1,4),Dur(3,16),Dur(1,16),Dur(3,16),Dur(1,16),Dur(1,2),Dur(1,12),Dur(1,12),Dur(1,12),Dur(1,12),Dur(1,12),Dur(1,12),Dur(1,12),Dur(1,12),Dur(1,12),Dur(1,12),Dur(1,12),Dur(1,12),Dur(3,16),Dur(1,16),Dur(3,16),Dur(1,16),Dur(1,2))
i = Int(4)
mainphrase = Phrase(Array(Chord(Array(Pitch(3,4)),Dur(1,4)),Chord(Array(Pitch(3,4)),Dur(1,4)),Chord(Array(Pitch(3,4)),Dur(3,16)),Chord(Array(Pitch(5,4)),Dur(1,16)),Chord(Array(Pitch(7,4)),Dur(1,4)),Chord(Arr
\text{trip8} = \text{Dur}(1,12)
int compose() {
  dur d = (1,8);
  pitch[] pitches = $C ** 8;
  pitch base = $C0;
  pitch max = $C8;
  int i;
  int n = 40;

  //Initialize an array of pitches
  for(i = 0; i < 8; i = i + 1) {
    pitches[i] = base + i * 12;
  }

  //Main loop
  phrase ph = newphrase();
  for(i = 0; i < n; i = i + 1) {
    int j;
    for(j = 0; j < 8; j = j + 1) {
      pitches[j] = pitches[j] + 1;
      if(pitches[j] == max)
        pitches[j] = base;
    }

    //Put them all on top of one another
    chord ch = newchord(d);
    for(j = 0; j < 8; j = j + 1) {
      ch = ch ^ pitches[j];
    }

    ph = ph + ch;
  }

  play(ph);
}

cord newchord(dur d) {
  note n1 = ($R,d);
  return n1 ^ $R;
}

phrase newphrase() {
  chord c1 = newchord((0,1));
  chord c2 = c1;
  return c1 + c2;
}
6.3 Test Suites

6.3.1 Motivation

Test cases were chosen to test individual features of the language (such as variable declaration, operator functionality, control structures, etc.) as independently of each other as possible. This makes it easier to debug our compiler and feature implementation. Test cases were also written to be as thorough as possible. For example, for operators, the operator is tested using all the different possible type operand combinations.

6.3.2 Automation

Testing is automated using the test.sh file and the test suite can be run by executing the command ./test.sh on the command line.

Laura was our Lead Tester and wrote the test suite, automated test script, and expected output files. Most of the individual test cases testing different components and features of the language were written by the person who implemented the features.

7 Lessons Learned

7.1 Andrew O’Reilly

If you have six tasks and three people, and you tell each of them to pick two and do them, very little will get done. The most invasive and dictatorial management styles are probably the most effective, to the extent that they do not upset everyone else. If your personality renders you incapable of this, or if your management style relies on sticks as well as carrots, you should not be the manager of this project, as the sticks available to you in this context will be limited (as will the carrots). Communication is of the utmost importance, and if there are team members who do not communicate effectively be sure to communicate this to them. If you set your own internal deadlines aside from those given by the professor, write them down or post them somewhere.

I have further learned that functional programming is all about answering questions and elucidation of meaning in a programming context. A functional compiler is constantly asking you what is returned by some set of code, what that result means, and whether it makes sense. Try to keep this in mind if you are to write functional code.

7.2 Stephanie Huang

Learning OCaml was a challenging, but rewarding experience. One of the key takeaways I got from the overall experience of writing in OCaml and also designing a programming language is the idea that you should try to do more with less. OCaml is all about doing a lot with a little bit of code, and similarly, our programming language is trying to do a lot with a few lines of code.
As for working as a team, and projects in general: communication is key. It’s important to always keep in touch, schedule regular meeting times and checkins, communicate your own ideas and what you’ve been working on. A lot of project work is also taking the initiative to do something, especially when working with peers when there isn’t as much of a topdown structure as there might be in some real world working environments. Try to set milestones and plan a timeline in advance, maybe even from day one.

7.3 Jonathan Sun

Keep things simple. Have a written long term game plan from the start. Don’t be married to any initial idea, learn to let go, allow yourself to be convinced of new ideas. Make attempts at documentation. Don’t touch working code. Always ask for help right away. Any embarrassment that holds you back will damage the group’s progress.

7.4 Laura Tang

For many students, including myself, this class is one of the first where you are required to complete a semester long group project with a team of other students. On top of that, the project requires you to get used to programming in OCaml, employing the functional programming paradigm rather than imperative programming that most are more familiar with. This lack of global view may cause analysis paralysis, so to resolve that, it is best to have an organized leader and to break down the tasks early after a thorough brainstorm with your members. Personally assigned tasks made of smaller chunks also are best, as they pre-commit members to have ownership of their own small goal.

I learned that it’s best to remain transparent: any changes that are made to the design should be communicated to all members, whether it is through the group chat, Github updates, a workboard, etc. Any work that you do on the project, work that may take longer than expected, and bugs you found should all be reported. In addition, I found that it’s important that each member keeps up to date on those changes, because the implementation of the language features will definitely be modified along the way.

8 Appendix

8.1 Source Code

```ml
(*
CMajor AST
by PLT Sandwich
Andrew OReilly, Stephanie Huang
*)

(* All operators. TODO: make sure this is complete *)
```
type operator = Add | Sub | Mul | Div | Layer | Arrcat | Rep
  | Eq | Neq | Gt | Gte | Lt | Lte

(* Variable declaration. See parser for initialization *)

(type vdecl = string * string)

(* Literal types *)
(type literal =
  Intlit of int (* Integers - 42 *)
  | Pitchlit of int * int (* Pitches - $Cb7 *)
  | Tuple of expr * expr (* Tuples - notes, durations *)

and expr =
  Binop of expr * operator * expr (* Binary operations *)
  | Noexpr (* Empty expression *)
  | Lit of literal (* Literals are expressions *)
  | Asn of literal * expr (* Variable assignment *)
  | Id of string (* Identifiers *)
  | Call of string * expr list (* Function call *)
  | Arr of expr list (* Arrays *)
  | Arrget of string * expr (* Array reference *)
  | Arrmod of string * expr * expr (* Array modification *)

(* Statements *)
(type stmt =
  Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | Vdecl of string * string
  | VdeclAsn of stmt * expr

(* Function declaration. Args list is now a vdecl list *)
(type func_decl = {
  ftype: string;
  fname: string;
  formals: vdecl list;
  locals : string list;
  body: stmt list;
})

(* type program = stmt *)
(type program = vdecl list * func_decl list)

cmajor.ml

(*
cmajor.ml
Main executable for C-Major
By PLT Sandwich
Andrew OReilly, Stephanie Huang
*)
open Ast
open Sast
open Compile
open Semantics
open Printf

let _ =
  let chan = if Array.length Sys.argv = 2 then
  try
    Some(open_in Sys.argv.(1))
  with Sys_error(s) ->
    print_endline ("Error: " ^ s); None
  else
    Some(stdin)
  in
  match chan with
  Some(channel) ->
    let lexbuf = Lexing.from_channel channel in
    let prog = Parser.program Scanner.token lexbuf in
    Execute.execute_prog prog
  | _ -> exit 1;

compile.ml

(* compile.ml
Main compiler for the CMajor programming language
by PLT Sandwich
Andrew O'Reilly, Stephanie Huang
*)

open Ast
open Sast
open Semantics
open Printf

exception Invalid_play of string
exception Illegal_operation of string
exception Type_error of string
exception Duplicate_name of string
exception Not_implemented of string

(* Symbol map *)
module NameMap = Map.Make(String)

(* Environment: symbol tables for functions, global, local vars *)
type global_environment = {
  variables : (c_type * s_type) NameMap.t;
  functions : func_decl NameMap.t
}

type symbol_table = {
  parent : symbol_table option;
  variables : (c_type * s_type) NameMap.t
}

type env = {
  glob_env : global_environment;
  scope : symbol_table;
  return_type : s_type;}
return_val : c_type;

type composition = {
    dur1 : int list;
    dur2 : int list;
    pitches : int array list
}

(* extract value from c_type * s_type tuple *)
let get_value (v, t) = match v with
    | None -> raise Not_found
    | _ -> v

(* execute play by writing pitches and durations to composition *)
let play s =
    (* next is a phrase *)
    let handle_phrase next l = match next with Phrase(chords) ->
        let comp = { dur1 = []; dur2 = []; pitches = [Array.make (Array.length chords) (-1)] }
          in let rec match_length llen alen l =
            if llen >= alen then l else
                match_length (llen + 1) alen ((Array.make (Array.length chords) (-1)) :: l)
          in
    (* handles the pitches *)
    let (na, new_pitches) = Array.fold_left (fun (i, comp_pitches) chord ->
        match chord with Chord(p,d) ->
            (match d with Dur(d1, d2) ->
                { dur1 = d1 :: comp.dur1;
                  dur2 = d2 :: comp.dur2;
                  pitches = new_pitches;
                } | _ -> raise (Type_mismatch ("Error in play")))
            | _ -> raise (Type_mismatch ("Error in play")))
            ) (0, comp.pitches) chords
        ) :: l
    in
    let s' = Array.fold_right handle_phrase phrases []
    (* handles the durations *)
    in
      match s' with Score(phrases) ->
        Array.fold_right handle_phrase phrases []
        | _ -> raise (Type_mismatch ("Error in play"))
    in

let csv_ints (listarg : int list) =
  List.fold_right (fun next str -> (string_of_int next) ^ "", ^ str) listarg "}
let rec find_symtab_var symtab name =
  try NameMap.find name symtab.variables
  with Not_found ->
    match symtab.parent with
    Some(parent) -> find_symtab_var parent name
    | _ -> raise Not_found

let find_var env name =
  try find_symtab_var env.scope name
  with Not_found ->
    try find_symtab_var {parent = None; variables = env.glob_env.variables} name
    with Not_found -> raise Not_found

let rec update_var symtab name newval (t : s_type) =
  try
    let (oldval, typ) = NameMap.find name symtab.variables in
    if t <> typ then raise (Type_mismatch(name))
    else
      if t = SArray then if get_arr_type newval <> get_arr_type oldval
        then raise (Type_mismatch(name))
                  (* else ()*)
      else
        let newparent = update_var psymtab name newval t in
        parent = Some(newparent);
        variables = symtab.variables
      |
      | _ -> raise Not_found
  with Not_found ->
    match symtab.parent with
    Some(psymtab) ->
      let newparent = update_var psymtab name newval t in
      |
    Some(psymtab) ->
      let newparent = update_var psymtab name newval t in
      |
    | _ -> raise Not_found

let update_symtab env name newval (t : s_type) =
  try
    let new_symtab = update_var env.scope name newval t in
    newval, {
      new_symtab = new_symtab;
      return_type = env.return_type;
      return_val = env.return_val;
      glob_env = env.glob_env;
      scope = new_symtab;
    }
  with Not_found -> (}
  try
    let new_glob_scope = update_var {parent = None; }
variables = env.glob_env.variables
| name newval t in
newval, {
  glob_env = {
    variables = new_glob_scope.variables;
    functions = env.glob_env.functions
  };
  scope = env.scope;
  return_type = env.return_type;
  return_val = env.return_val
}
with Not_found -> raise Not_found
| Match_failure(s, l, c) -> raise (Type_error(name ^ c_type_str(newval))

let rec translate block env = {
  (* translate all expressions to cmaj type *)
  let rec expr expenv = function
  Id(s) -> (try let (c, env1) = find_var expenv s in with Not_found -> raise (Missing_variable ("Error: "^"s"^" not defined!"))
    | Asn(s, e) -> (try let (c, env1) = (expr expenv e) in with Not_found -> raise (Missing_variable ("Error: "^"s"^" not defined!"))
      | Arr(e) -> (try let arr_type = c_to_s_type (fst (expr expenv (List.hd e))) in let arr_check elem = let c_elem = fst (expr expenv elem) in if (is_valid_elem c_elem arr_type) then c_elem else raise (Type_error("Error: unexpected type encountered")) in let arr_data = Array.of_list (List.map arr_check e) in Array(arr_type, arr_data), expenv
    )
  )

in arr_data.(index), env1
  with Invalid_argument x -> raise (Invalid_argument ("index out of bounds"))
  }
| Arrmod(s, i, e) -> (  
  try
    let (arr, env1) = expr expenv (Id(s)) in
    let index = match (fst (expr env1 i)) with
      Int(idx) -> idx
    | _ -> raise (Type_error ("Error: index value is not an integer"))
    in let arr_data = match arr with
      Array(typ, dat) -> dat
    | _ -> raise (Type_error ("Error: " ^s" is not an array"))
    in let newval = fst (expr env1 e) in
      if is_valid_elem newval (get_arr_type arr) then
        arr_data.(index) <- newval
      else raise (Type_error ("Error: unexpected type encountered"));
        newval, env1
      with Invalid_argument x -> raise (Invalid_argument ("index out of bounds"))
      }
| Lit(x) -> (  
  match x with
    Intlit(x) -> Int(x), expenv
  | Pitchlit(l,o) -> Pitch(l, o), expenv
  | Tuple(x, y) -> (  
    let (ex, env1) = expr expenv x in
    let (ey, env2) = expr env1 y in
      match ex, ey with
        Int(a), Int(b) -> Dur(a, b), env2
      | Pitch(l, o), Dur(a, b) -> Note(Pitch(l,o), Dur(a,b)), env2
      | Array(SPitch, d), Dur(a, b) -> Chord(d, Dur(a,b)), env2
      | _ -> raise (Type_error("Invalid tuple"))
    )
  )
| Call(name, elist) -> (  
  let argc, argv, callenv = List.fold_left
    (fun (c,v,envb) next ->  
      let lit, enva = expr envb next in
        c + 1, lit :: v, enva
    )
    (0, [], expenv) elist in
  match name with
    "play" -> (  
      let master_score = match s with
        Phrase(chords) as p -> Score([|p|])
      | Chord(p,d) as c -> Score([|Phrase([|c|])|])
      | Note(p,d) as c -> Score([|Phrase([|Chord([|p|],d)|])|])
      | _ -> raise (Invalid_play("Invalid call to play"))
    )
  in let comp_list = play master_score in
  let ofile = open_out "out.csv" in
  List.iter (fun comp ->  
    let (duristr, dur2str) =  
      (csv_ints comp.dur1, csv_ints comp.dur2)
    in List.iter (fun parray ->
let pitchstr = csv_ints (Array.to_list parray) in
fprint ofile "%s\n%s\n%s\n" pitchstr dur1str dur2str;
)
} comp.pitches
}
close_out ofile;
Int(1), expenv
| _ -> raise (Invalid_play("Invalid call to play"))
|
| _ ->
try
let fxn = NameMap.find name callenv.glob_env.functions in
let funenv = exec_fun callenv fxn (List.rev argv) in
funenv.return_val, {
glob_env = funenv.glob_env;
scope = callenv.scope;
return_type = callenv.return_type;
return_val = callenv.return_val
}
with Not_found -> raise (Missing_function("Unknown function " ^ name))
(*raise (Not_Implemented("Custom functions"))*)
|
| Binop (e1, op, e2) -> {
let (lit1, env1) = expr expenv e1 in
let (lit2, binenv) = expr env1 e2 in
match op with
Add -> (
match (lit1, lit2) with
| Int(x), Int(y) -> Int(x + y), binenv
| (Pitch(lo, o) as p), (Int(y) as i) -> raise_pitch i p, binenv
| Dur(x, y), Int(z) -> dur_add (Dur(z,1)) (Dur(x, y)), binenv
| Dur(x, y), Dur(z, w) -> dur_add (Dur(x, y)) (Dur(z, w)), binenv
(* as concatenation *)
| Note(p1,d1), Note(p2,d2) -> Phrase([| Chord([|p1|],d1); Chord([|p2|],d2) |]), binenv
| Note(p,d), (Chord(p2,d2) as c) -> Phrase([| Chord([|p|], d); c |]), binenv
| (Chord(p2,d2) as c), Note(p,d) -> Phrase([| c; Chord([|p|], d) |]), binenv
| Note(p,d), Phrase(c) -> Phrase(Array.append [| Chord([|p|],d) |]) c), binenv
| Phrase(c), Note(p,d) -> Phrase(Array.append c [| Chord([|p|],d)|]), binenv
| Chord(p1,d1) as c1, (Chord(p2,d2) as c2) -> Phrase([| c1; c2 |]), binenv
| Phrase(c1), (Chord(p,d) as ch) -> Phrase(Array.append c [| ch|]), binenv
| Chord(p,d) as ch, Phrase(c) -> Phrase(Array.append [|ch|] c), binenv
| Phrase(c1), Phrase(c2) -> Phrase(Array.append c1 c2), binenv
| _ as f, (_ as s) ->
raise (Illegal_operation((c_type_str f) " + " (c_type_str s))
)
| Sub ->
let lit = {
match lit1, lit2 with
| Int(x), Int(y) -> Int(x-y)
| Pitch(letr,oct), Int(x) ->
let rcomp = letr - x in
if rcomp < 0 then
let posval = rcomp - 1 in
let mds = posval mod 12 in
let submd = 12 - mds in
```ml
let oreduce = posval/12 in
let oret = oct - (oreduce+1) in
if oret < 0 then Pitch(0,0) else Pitch(submod, oret)
else Pitch(rcomp, oct)
do
| Dur(num,den) as d, Int(x) ->
  dur_sub_abs (d, Dur(x,1))
do
| Pitch(l1,o1) as p1, (Pitch(l2,o2) as p2) ->
  let i1 = pitch_to_int p1 in
  let i2 = pitch_to_int p2 in
  Int(i1 - i2)
do
| Chord(p_array,d), (Pitch(l,o) as p) ->
  let pitch_list = Array.fold_left
    (fun plist next ->
      if next = p then plist else next :: plist
    ) [] p_array
  in let pitches = Array.of_list pitch_list
  in Chord(pitches, d)
do
| Dur(_,_) as d1, (Dur(_,_) as d2) ->
  dur_sub_abs (d1,d2)
do
| Note(p, d), (Dur(_,_) as d2) ->
  Note(p, (dur_sub_abs (d, d2)))
do
| Chord(p,d1), (Dur(_,_) as d2) ->
  Chord(p, (dur_sub_abs (d, d2)))
do
| _ as f, (_ as s) ->
  raise (Illegal_operation((c_type_str f) ^ "-" ^ (c_type_str s)))
do
| Mul ->
  let lit = (match lit1, lit2 with
    | Int(x), Int(y) -> Int(x*y)
do
    | Dur(n,d), Int(x) ->
      let rn = n*x in
      let g = gcd rn d in
      Dur(rn/g,d/g)
do
    | Dur(n1,d1), Dur(n2,d2) ->
      let n = n1 * n2 in
      let d = d1 * d2 in
      let g = gcd n d in
      Dur(n/g,d/g)
do
    | _ as f, (_ as s) ->
      raise (Illegal_operation((c_type_str f) ^ "-" ^ (c_type_str s)))
do
  ) in lit, binenv
| Div -> {
  match {lit1, lit2 with
    | Int(x), Int(y) -> Dur(x, y), binenv
      (* int / int *)
do
    | Dur(n,d), Int(i) ->
      dur_divide (Dur(n,d), Int(i)), binenv
      (* dur / int *)
do
    | Note(p, d), Int(i) ->
      Note(p, dur_divide (d, Int(i))), binenv
      (* note / int *)
do
    | Chord(p, d), Int(i) ->
      Chord(p, dur_divide (d, Int(i))), binenv
      (* chord / int *)
do
    | Int(x), Dur(n,d) ->
      dur_divide (Int(x), Dur(n,d)), binenv
      (* int / dur *)
do
    | Dur(n1,d1), Dur(n2,d2) ->
      dur_divide (Dur(n1,d1), Dur(n2,d2)),
      (* dur / dur *)
do
  ) in lit, binenv
```

```
binenv (* dur / dur *)
| Note(p, dur), Dur(n,d) -> Note( p, dur_divide (dur, Dur(n,d)) ),

binenv (* note / dur *)
| Chord(p, chdur), Dur(n,d) -> Chord{ p, dur_divide (chdur, Dur(n,d)) },

binenv (* chord / dur *)
| Dur(n,d), Note(p, dur) -> Note(p, dur_divide (Dur(n,d), dur)),

binenv (* dur / note *)
| Dur(n,d), Chord(p, chdur) -> Chord( p, dur_divide (chdur, Dur(n,d)) ),

binenv (* dur / chord *)
| _ as f, () as s) -> raise (Illegal_operation((c_type_str f) */(*) (c_type_str s)))

379 |}
380 |
381 | Layer -> {
382 |   match (lit1, lit2) with
383 |   (Dur(n,d1) as d), (Pitch(l,o) as p) -> Note(p, d), binenv
384 |   (* TODO: array of pitches and array of durations (* Done? *)*)
385 |   Array(SPitch, p), Array(SDur, d) -> (386 |   if (Array.length p) != (Array.length d) then
387 |     raise (Illegal_operation("Error: pitch[] and dur[] must be same length")
388 |   )
389 |   (Pitch(l,o) as pitch), Note(p,d) ->
390 |   Chord([| p; pitch |], d), binenv
391 |   (Pitch(l,o) as pitch), Chord(p,d) ->
392 |   Chord(Array.append p [| pitch |], d), binenv
393 |   (Pitch(l,o) as pitch), (Chord(p_array,d2) as c) ->
394 |   Score([| Phrase([|Chord([|p1|],d1)|]); Phrase([|c|]) |]), binenv
395 |   (Chord(p_array,d2) as c), Note(p1,d1) ->
396 |   Score([| Phrase([|c|]); Phrase([|Chord([|p1|],d1)|]) |]), binenv
397 |   (Chord(p1,d1) as c1), (Chord(p2,d2) as c2) ->
398 |   Score([| Phrase([|c1|]); Phrase([|c2|]) |]), binenv
399 |   (Chord(p,d) as c), (Phrase(s) as ph) ->
400 |   Score([| Phrase([|c|]); ph |]), binenv
401 |   (Phrase(s) as ph), (Chord(p,d) as c) ->
402 |   Score([| ph; Phrase([|c|]) |]), binenv
403 |   (Phrase(c1) as p1), (Phrase(c2) as p2) ->
404 |   Score([| p1; p2 |]), binenv
405 |   (Phrase(c) as p), Score(ph) ->
406 |   Score(ph), (Phrase(c) as p) -> Score(Array.append ph [|p|]), binenv
407 |   _ as f, () as s) ->
408 |     raise (Illegal_operation((c_type_str f) */(*) (c_type_str s)))
409 |   )
410 |
411 |   Array(t1, d1), Array(t2, d2) ->
412 |     if t1 = t2 then Array(t1, Array.append d1 d2), binenv
413 |     else raise (Illegal_operation("Error: cannot concatenate arrays of two
types")
414 |     )
415 |   if c_to_s_type x - 0 then Array(t, Array.append d [|x|]), binenv
416 |     else raise (Illegal_operation("Error: type mismatch")
417 |   )
418 |
419 |   Arrcat -> {
420 |     match (lit1, lit2) with
421 |     Array(t1, d1), Array(t2, d2) ->
422 |       if t1 = t2 then Array(t1, Array.append d1 d2), binenv
423 |       else raise (Illegal_operation("Error: cannot concatenate arrays of two
types")
424 |       )
425 |     Array(t, d), x ->
426 |       if c_to_s_type x - t then Array(t, Array.append d [|x|]), binenv
427 |       else raise (Illegal_operation("Error: type mismatch")
428 |     )
429 |   )
Let's start by analyzing the code snippet provided. The code appears to be written in a functional programming language, possibly OCaml, given the syntax and usage of types and functions.

### Code Snippet Breakdown

1. **Function Definitions**
   - The code defines several functions, each handling different kinds of literals and operations. These functions take different types of arguments and return results based on those types.

2. **Type System**
   - The code uses a type system to ensure that operations are performed on compatible types. For example, it checks if types match before performing operations.

3. **Literal Handling**
   - The code includes functionality to handle various types of literals, including integers, durations, pitches, and notes. Each type is handled with specific operations.

4. **Conditional Logic**
   - The code uses conditional logic extensively, such as `if` statements to check type compatibility and perform operations accordingly.

5. **Array Operations**
   - Array operations are performed within the context of certain types, ensuring that operations like appending elements are type-safe.

6. **Error Handling**
   - Error handling is crucial in the code, with specific cases raising exceptions if types do not match.

### Key Observations

- **Type Conversion**
  - The code includes functions for converting between different types, ensuring that operations are performed correctly.

- **Array Manipulation**
  - Arrays are manipulated in a type-aware manner, with operations like appending to arrays handled within the context of specific types.

- **Error Conditions**
  - The code is designed to handle errors gracefully, raising exceptions when operations are performed on incompatible types.

- **Function Call Context**
  - Functions are called in a context that ensures they are used appropriately, with specific checks for type compatibility.

### Conclusion

The provided code snippet is a well-structured piece of functional programming code. It demonstrates the importance of type safety and error handling in modern programming languages. The use of conditional logic and specific type checks is crucial for ensuring that operations are performed correctly and safely.
let r = if x > y then 1 else 0 in Int(r), binenv
| Dur(n1,d1) as dur1, (Dur(n2,d2) as dur2) ->
    let ctype = dur_sub (dur1,dur2) in
    (match ctype with
     Dur(sn,sd) -> let r = if sn > 0 then 1 else 0 in
                    Int(r), binenv
     _ -> raise (Illegal_operation("Problem with dur_sub")))
| Pitch(p1,o1) as pitch1, (Pitch(p2,o2) as pitch2) ->
    let i1 = pitch_to_int pitch1 in
    let i2 = pitch_to_int pitch2 in
    let r = if i1 > i2 then 1 else 0 in
    Int(r), binenv
| _ as f, (_ as s) ->
    raise (Illegal_operation((c_type_str f) ">" (c_type_str s)))
    | Gte -> (match lit1, lit2 with
             Int(x), Int(y) ->
             let r = if x >= y then 1 else 0 in
             Int(r), binenv
             _ as f, _ as s ->
             raise (Illegal_operation((c_type_str f) "\geq" (c_type_str s)))
             | Lt -> (match lit1, lit2 with
                    Int(x), Int(y) ->
                    let r = if x < y then 1 else 0 in
                    Int(r), binenv
                    _ as f, _ as s ->
                    raise (Illegal_operation((c_type_str f) "<" (c_type_str s)))
                    | Lte -> (match lit1, lit2 with
                             Int(x), Int(y) ->
                             let r = if x <= y then 1 else 0 in
                             Int(r), binenv
                             _ as f, _ as s ->
                             raise (Illegal_operation((c_type_str f) "\leq" (c_type_str s)))
                             | Gte -> (match lit1, lit2 with
                                      Int(x), Int(y) ->
                                      let r = if x > y then 1 else 0 in
                                      Int(r), binenv
                                      _ as f, (_ as s) ->
                                      raise (Illegal_operation((c_type_str f) ">" (c_type_str s))))
Dur(sn, sd) \rightarrow \begin{align*}
&\text{let } r = \text{if } sn \leq 0 \text{ then } 1 \text{ else } 0 \text{ in} \\
&\text{Int}(r), \text{bienv}
\end{align*}

| _ \rightarrow \text{raise (Illegal_operation("Problem with dur_sub"))}

| Pitch(p1, o1) as pitch1, (Pitch(p2, o2) as pitch2) \rightarrow

| \text{let } i1 = \text{pitch_to_int } pitch1 \text{ in}

| \text{let } i2 = \text{pitch_to_int } pitch2 \text{ in}

| \text{let } r = \text{if } i1 \leq i2 \text{ then } 1 \text{ else } 0 \text{ in}

| \text{Int}(r), \text{bienv}

| _ \rightarrow \text{raise (Illegal_operation({c_type_str f} \ ^\"\text{<=}\ ^\" \ (c_type_str s)}))}

(* Processes a single statement; returns updated environment *)

in \text{let rec stmt stenv e =}

\text{if } stenv.return_val <> \text{None then stenv}

\text{else match e with}

Expr e \rightarrow \text{let } c, \text{renv} = \text{expr stenv e in renv}

| Vdecl(t, id) \rightarrow ()

\text{try}

| \text{let } _ = \text{NameMap.find id stenv.scope.variables}

\text{in raise (Duplicate_name(id \ ^\"\text{already defined}\"))}

\text{with Not_found -->}

| \text{let } typ = \text{type_from_str } t \text{ in}

| \text{let } new_entry typ = (\text{match typ with}

| \text{SArray } \rightarrow (\text{Array(arr_type_from_str } t, \[||\]), typ)

| \text{else } (\text{None, typ})

| _ \rightarrow (\text{None, typ})

| in let newval, newtyp = new_entry typ

| in let new_scope = {

| \text{parent} = stenv.scope.parent;

| \text{variables} =

| \text{NameMap.add id (newval, newtyp) stenv.scope.variables}

| in {

| \text{glob_env} = stenv.glob_env;

| \text{scope} = new_scope;

| \text{return_type} = stenv.return_type;

| \text{return_val} = stenv.return_val

| }

| }

| VdeclAsn(s, e) \rightarrow (**print_endline ("VdeclAsn");**)\n
| let newstenv = stmt stenv s in

| let c, newenv = expr newstenv e in newenv

| Return(e) \rightarrow (**print_endline ("Return");**)

| let c, renv = expr stenv e in

| let stype = c_to_s_type c in

| if stype <> renv.return_type

| then raise (Type_mismatch(

| "Function type " \ ^\" \ (str_from_type renv.return_type)

| \^\" \ (str_from_type stype)

| " does not match function return type " ^\" (str_from_type stype)

| )

| \text{else } {

| \text{glob Env} = renv.glob_env;

| \text{scope} = renv.scope;

| \text{return_type} = renv.return_type;
let block_scope = {
  parent = Some(stenv.scope);
  variables = NameMap.empty
} in
let block_env = {
  glob_env = stenv.glob_env;
  scope = block_scope;
  return_type = stenv.return_type;
  return_val = stenv.return_val
} in
let post_block_env = List.fold_left stmt block_env sl in
let post_block_scope = (match post_block_env.scope.parent with
  Some(parent) -> parent
| _ -> stenv.scope)
  in {
  glob_env = post_block_env.glob_env;
  scope = post_block_scope;
  return_type = stenv.return_type;
  return_val = stenv.return_val
}
| If(e,s_if,s_else) ->
  let c, ifenv = expr stenv e in {
    match c with
    Int(x) when x > 0 -> stmt ifenv s_if
    | Int(0) -> stmt ifenv s_else
    | _ -> raise (Type_error("Conditional must be of type int instead of " 
      (c_type_str c))
  )
| For(init,cond,iter,st) ->
  let cinit, initenv = expr stenv init in
  exec_loop initenv cond iter st
and exec_loop env cond iter st =
  let c_cond, condenv = expr env cond in
  match c_cond with
  Int(x) when x > 0 ->
    let loopenv = stmt condenv st
    let _, iterenv = expr loopenv iter in
    exec_loop iterenv cond iter st
  | Int(0) -> env
  | _ ->
    raise (Type_error("For loop conditional must be of type int instead of " 
      (c_type_str c_cond))
  )
  in stmt env block
and exec_fun env fxn args =
  let params = List.fold_left
    (fun amap (t, id) -> NameMap.add id (None, type_from_str t) amap)
  NameMap.empty fxn.formals
let funscope = {
  parent = None;
  variables = params
} in
let empty_fun_env = {
  glob_env = env.glob_env;
  scope = funscope;
  return_type = type_from_str fxn.ftype;
  return_val = None
} in
try
let _, fun_env = List.fold_left2
(fun (_, fenv) (t, id) c_expr ->
  try
    update_symtab fenv id c_expr (c_to_s_type c_expr)
  with Type_mismatch(s) -> raise (Type_mismatch("Invalid argument to function " ^ fxn.fname ^ ": " ^ id))
) (None,empty_fun_env) fxn.formals args
in
let post_fun_env = List.fold_left
(fun nextenv nextstmt ->
  translate nextstmt nextenv
) fun_env fxn.body
in
post_fun_env (* DO NOT MIX THIS UP WITH THE CALLER'S ENVIRONMENT *)
with Invalid_argument(s) -> raise (Invalid_argument(fxn.fname ^ ": Wrong number of arguments"))

let execute_prog (vdecls, fdecls) =
let print_env key (c, s) =
  print_endline (key ^ " = " ^ (Sast.c_type_str c)) in
let var_map = List.fold_left
(fun vmap (t, id) ->
  if NameMap.mem id vmap
  then raise (Duplicate_name(id ^ " already defined"))
  else NameMap.add id (None, (type_from_str t)) vmap
) NameMap.empty vdecls
in
let fun_map = List.fold_left
(fun map fdec -> NameMap.add fdec.fname fdec map

let globals = {
    variables = var_map;
    functions = fun_map;
} in
let scp = {
    parent = None;
    variables = NameMap.empty
} in
let env = {
    glob_env = globals;
    scope = scp;
    return_type = SInt;
    return_val = None
} in
let _, postexec_env = if NameMap.mem "compose" env.glob_env.functions then (), (exec_fun env (NameMap.find "compose" env.glob_env.functions) []) else (print_endline "No compose function"), env
in let ofile = open_out "play.out" in
NameMap.iter print_env postexec_env.scope.variables;
fprintf ofile "#!/bin/bash\njava CSVPlayer out.csv";
close_out ofile

%{
(*
Parser for CMajor
By PLT Sandwich
Andrew O'Reilly, Stephanie Huang
*)
open Ast
%
}%
%token PLUS MINUS TIMES DIVIDE LAYER REPEAT ARRCAT ASSIGN EQ NEQ GT GTE LT LTE EOF
%token LPAREN RPAREN LCBRACE RCBRACE LSBRACE RSBRACE COMMA SEMI RETURN IF ELSE FOR
%token PITCH_LIT
%token <int> INT_LIT
%token <int> PITCH_LETTER
%token <int> PITCH_SIGN
%token <string> ID
%token <string> TYPENAME
%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left ARRCAT
%left REPEAT
%left EQ NEQ
%left LT GT LTE GTE
%left PLUS MINUS
program: 
decls EOF { $1 }

decls:
  { [], [] } | decls vdecl SEMI { ($2 :: fst $1), snd $1 } | decls fdecl { fst $1, ($2 :: snd $1) }

vdecl:
  TYPENAME ID { $1, $2 } (* Variable declaration *) | TYPENAME LSBRACE RSBRACE ID { $1"["", $4 (* Array declaration *) }

fdecl:
  TYPENAME ID LPAREN formal_list RPAREN LCBRACE stmt_list RCBRACE 
  { 
    ftype = $1; 
    fname = $2; 
    formals = $4; 
    locals = []; 
    body = List.rev $7; 
  }

formals_opt:
  { [] } | formal_list { List.rev $1 }

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

stmt_list:
  { [] } | stmt_list stmt { $2 :: $1 }

stmt:
  expr SEMI { Expr($1) } | vdecl SEMI { Vdecl(fst $1, snd $1) } | vdecl ASSIGN expr SEMI { VdeclAsn(Vdecl(fst $1, snd $1), Asn(snd $1, $3)) } | RETURN expr SEMI { Return($2) } | LCBRACE stmt_list RCBRACE { Block(List.rev $2) } | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) } | IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) } | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt { For($3, $5, $7, $9) }

eexpr_opt:
```ml
(*
Semantic AST
by Andrew O'Reilly, Stephanie Huang
*)
open Ast

exception Type_mismatch of string
exception Missing_variable of string
exception Missing_function of string
exception Invalid_type of string

sast.ml
```
module StringMap = Map.Make(String) ;;

(* Types for semantic checking *)
type s_type = SInt | SPitch | SDur | SNote | SChord | SPhrase
| SScore | SArray | None

(* cmajor types *)
type c_type =
    Int of int
| Pitch of int * int
| Dur of int * int
| Note of c_type * c_type
| Chord of c_type array * c_type
| Phrase of c_type array
| Score of c_type array
| Array of s_type * c_type array
| None

(* Get the semantic type of a storage type *)
let rec c_to_s_type =
    function
    Int x -> SInt
| Pitch (l,o) -> SPitch
| Dur (a,b) -> SDur
| Note (p,d) -> SNote
| Chord (p,d) -> SChord
| Phrase x -> SPhrase
| Score x -> SScore
| Array (s,l) -> SArray
| None -> None

(* Get string representation of a c_type *)
let rec c_type_str =
    function
    Int(x) -> "Int(ˆstring_of_int x ˆ)"
| Pitch(l, o) -> "Pitch(ˆstring_of_int l)ˆ","ˆstring_of_int o ˆ)"*
| Dur(a,b) -> "Dur(ˆstring_of_int a)ˆ","ˆstring_of_int b)ˆ)"
| Note(p,d) -> "Note(ˆc_type_str p)ˆ","ˆc_type_str d ˆ)"
| Chord(p,d) -> "Chord(ˆc_type_str Array(SPitch, p)))ˆ","ˆc_type_str d)ˆ)"
| Phrase(p) -> "Phrase(ˆc_type_str Array(SChord, p)))ˆ)"
| Score(s) -> "Score(ˆc_type_str Array(SPhrase, s)))ˆ)"
| Array(s,a) -> "Array(ˆArray.fold_left (fun prev x ->
    if (prev = "") then (c_type_str x) else prev","ˆc_type_str x)
    ) " a")"
| None -> "None"

scanner.mll

(* Scanner for CMajor *)
By PLT Sandwich
Andrew O'Reilly, Stephanie Huang
(*)
open Parser

(* let pitch = $ [ 'A'-'G' ] [ '#' 'b' ]? <= Not used, but handy *)
let id = [ '_' 'A'-'Z' 'a'-'z' ] [ '_ '#'A'-'Z' 'a'-'z' '0'-'9' ]*
let typename = "int" | "pitch" | "dur" | "note" | "chord" | "phrase" | "score"

rule token = parse

[" ",
  
  
  
  "\t" ,
  
  
  "\n"] { token lexbuf } (* Whitespace *)

| "*/" |
| "//" |

{ bcomment lexbuf } (* Block comments /* */) *

{ lcomment lexbuf } (* Inline comments // *)

| '(' | { LPAREN } (* Punctuation *) |
| ')'
| '{'
| '}'
| '['
| ']' |

| ';' |
| ',' |
| '=' |
| '+' |
| '-' |
| '*' |
| '/' |
| 'ˆ'
| '**' |
| '+=' |
| '-=' |
| '>=' |
| '<=' |
| '==' |
| '!=' |

{ EQ }

{ NEQ }

{ GT }

{ GTE }

{ LTE }

{ TIMES }

{ MINUS }

{ PLUS }

{ ASSIGN }

{ ID(id_str) } (* identifiers *)

{ TYPENAME(type_str) } (* type names *)

{ INT_LIT(int_of_string lit) } (* integers *)

{ PITCH_LETTER(0) } (* Pitch literals are split into separate symbols to make parsing easier *)

{ ID(id_str) }

{ EOF } (* end-of-file *)

and bcomment = parse

| "\n" { token lexbuf } (* Should this be quoted? *)

| _

and icomment = parse

| _

and ptoken = parse

| "A" { PITCH_LETTER(0) }
```ml
let is_valid_pitch (letter, oct) =
  match letter, oct with
  | A, _ -> true
  | B, _ when 0 <= B && B <= 11 && 0 <= oct && oct <= 8 -> true
  | _, _ -> false

let raise_pitch n p =
  match n, p with
  | Int(x), Pitch(l, o) ->
    let o = o + (l + x) / 12 in
    let l = (l + x) mod 12 in
    Pitch(l, o)
  | _, _ -> raise (Invalid_type("raise_pitch: invalid type"))

let pitch_to_int p =
  match p with
  | Pitch(l, o) -> 9 + o * 12 + l
  | _ -> raise (Invalid_type("pitch_to_int: invalid type"))

let type_from_str t =
  match t with
  | "int" -> Sast.SInt
  | "pitch" -> Sast.SPitch
  | "dur" -> Sast.SDur
  | "note" -> Sast.SNote
  | "chord" -> Sast.SChord
  | "phrase" -> Sast.SPhrase
  | "score" -> Sast.SScore
  | "int[]" | "pitch[]" | "dur[]" | "note[]" | "chord[]" | "phrase[]" | "score[]" -> Sast.SArray
  | _ -> raise (Invalid_type("""\""""t""""\"""" is an invalid type""""))
```

let get_arr_type a = match a with Array(typ, dat) -> typ
| _ -> raise (Invalid_type ("Get array type from non-array"))

let get_arr_data a = match a with Array(typ, dat) -> dat
| _ -> raise (Invalid_type ("Get array data from non-array"))

(* Check that element and array type match up *)
let is_valid_elem elem arr_type =
if ((c_to_s_type elem) = arr_type) then true else false

(* Gets type of array from string *)
let arr_type_from_str t =
    let typ =
        if String.contains t '[' then
            if String.contains t ']' then String.sub t 0 (String.length t - 2)
            else raise (Invalid_type ("'"^t^"' is an invalid array type"))
        else raise (Invalid_type ("'"^t^"' is an invalid array type"))
in type_from_str typ

let str_from_type = function
| Sast.SInt -> "int"
| Sast.SPitch -> "pitch"
| Sast.SDur -> "dur"
| Sast.SNote -> "note"
| Sast.SChord -> "chord"
| Sast.SPhrase -> "phrase"
| Sast.SScore -> "score"
| Sast.SArray -> "array"
| None -> "none"

(* For reducing fractions *)
let rec gcd a_arg b_arg =
    let a = if a_arg < 0 then a_arg * -1 else a_arg in
    let b = if b_arg < 0 then b_arg * -1 else b_arg in
    if a = 0 then b
    else if b = 0 then a
    else if a = b then
        a
    else if a > b then
gcd (a - b) b
    else
gcd a (b - a)

(* Handles cases of dur / int, int / dur, dur / dur *)
let dur_divide (a1, a2) =
    let n, d = match a1, a2 with
        Dur(dn,dd), Int(i) -> dn, dd*i
        | Int(i), Dur(dn,dd) -> dd*i, dn
        | Dur(dn1,dd1), Dur(dn2,dd2) -> dn1*dd2, dd1*dn2
        | _ -> 0,0
    in
    let g = gcd n d in
    Dur(n / g, d / g)

(* Duration subtraction - returns negative values *)
let dur_sub = function
    Dur(n1,d1), Dur(n2,d2) ->
        let nilsub = d2 * n1 in
        Dur(n1-nilsub, d1-nilsub)
let d1sub = d2 * d1 in
let n2sub = d1 * n2 in
let n = n1sub - n2sub in
let g = gcd n d1sub in
Dur(n / g, d1sub / g)
| __ _ -> raise (Invalid_type("Error: invalid type. Dur expected."))

(* Same as dur_sub, but returns absolute value *)
let dur_sub_abs = function
  Dur(d, d) as d1, (Dur(d, d) as d2) ->
  { match dur_sub (d1,d2) with
    Dur(n,d) -> Dur(Pervasives.abs n, d)
  |
  |
  | __ _ -> raise (Invalid_type("Error: invalid type. Dur expected."))

(* Add durations *)
let dur_add d1 d2 = match d1, d2 with
  Dur(n1,d1), Dur(n2,d2) -> {
    let n = n1 * d2 + n2 * d1 in
    let d = d1 * d2 in
    let g = gcd n d in
    Dur(n / g, d / g)
  |
  | __ _ -> raise (Invalid_type("Error: invalid type. Dur expected."))

Makefile
# Andrew O'Reilly, Jonathan Sun
OBJS = ast.cmo parser.cmo scanner.cmo sast.cmo semantics.cmo compile.cmo execute.cmo
cmajor.cmo
# Choose one
YACC = ocamlyacc
# YACC = menhir --explain
SRCS = $(wildcard *.*
JAVA_SRCS = $(shell find java -name '*.java')
TESTS := $(shell find tests -name '*.cmajor' -o -name '*.out')
DEMOS = $(shell find finaldemo *.cmajor -name '*.cmajor')
TARFILES = Makefile $(SRCS) $(JAVA_SRCS) $(TESTS) $(DEMOS)
cmajor : $(OBJS) CSVPlayer
  ocamlc -g -o cmajor $(OBJS)
CSVPlayer :
  javac CSVPlayer.java
scanner.ml : scanner.mll
  ocamllex scanner.mll
parser.ml parser.mli : parser.mly
  $(YACC) parser.mly
%cmi : %ml
  ocamlc -g -c $<
%cmo : %ml
  ocamlc -g -c $<
.PHONY : clean
clean :
  rm -f cmajor parser.ml parser.mli parser.output scanner.ml \\
  * .cmo * .cml * .out * .diff * .log * .txt * .gz
.PHONY : cleanJava
cleanJava :
  rm -f java/function_* java/output.java java/* .class
.PHONY : test
test : cmajor testall.sh
  ./testall.sh
.PHONY : all
all : clean cmajor CSVPlayer

# Generated by ocamldep *.ml *.mli >> Makefile
ast.cmo :
  cmajor.cmo : parser.cmi execute.cmo compile.cmo
  cmajor.cmx : parser.cmx execute.cmx compile.cmx
  compile.cmo : semantics.cmo sast.cmo ast.cmo
  compile.cmx : semantics.cmx sast.cmx ast.cmx
  execute.cmo : sast.cmo compile.cmo
  execute.cmx : sast.cmx compile.cmx
  parser.cmo : ast.cmo parser.cmi
  parser.cmx : ast.cmx parser.cmi
  sast.cmo : ast.cmo
  sast.cmx : ast.cmx
  semantics.cmo : sast.cmo ast.cmo
  semantics.cmx : sast.cmx ast.cmx
  parser.cmi : ast.cmo # this was extra, after an unsuccessful make.

CSVPlayer.java

/**
 *  Jonathan Sun
 */

import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Scanner;

public class CSVPlayer {
  
  final static int MEASURE_DUR = 2000; // Default measure duration, milliseconds.
```java
final static int PITCH = 0;
final static int DUR = 1;

public static void printUsage() {
    System.err.println("Usage: java CSVPlayer <my/path/to/file.csv>"
    );
    System.exit(1);
}

public static int[] calcDurs(int[] dur1s, int[] dur2s) {
    int[] durs = new int[dur1s.length];
    for (int i = 0; i < dur1s.length; i++)
        durs[i] = ((double)MEASURE_DUR * ((double)dur1s[i] / (double)dur2s[i])
    );
    return durs;
}

/**
 * Converts a string of comma separated integers to an array of integers.
 * @param csv comma separated integers
 * @return integer array
 */
public static int[] scanInts(String csv) {
    String[] nums = csv.split(",");
    int[] ints = new int[nums.length];
    for (int i = 0; i < nums.length; i++)
        ints[i] = Integer.parseInt(nums[i]);
    return ints;
}

public static ArrayList<int[][]> buildScore(Scanner input) {
    ArrayList<int[][]> score = new ArrayList<int[][]>();
    /* no error handling... */
    while (input.hasNextLine()) {
        int[][] voice = new int[2][];
        int[] pitches = scanInts(input.nextLine());
        int[] dur1s = scanInts(input.nextLine());
        int[] dur2s = scanInts(input.nextLine());
        int[] durs = calcDurs(dur1s, dur2s);
        voice[PITCH] = pitches;
        voice[DUR] = durs;
        score.add(voice);
    }
    return score;
}

public static void play(ArrayList<int[][]> score) {
    ArrayList<NotesPlayer> voices = new ArrayList<NotesPlayer>();
    for (int[][] voice : score)
        voices.add(new NotesPlayer(voice));
    for (NotesPlayer notes : voices)
        (new Thread(notes)).start();
}
```
```java
public static void main(String[] args) throws FileNotFoundException {
    if (args.length != 1) printUsage();
    Scanner input = new Scanner(new File(args[0]));
    ArrayList<int[][]> score = buildScore(input);
    input.close();
    play(score);
    System.out.println("Thank you for your patronage. Have a nice day."");
}
```

**NotesPlayer.java**

```java
/**
 * Jonathan Sun
 */
import javax.sound.midi.MidiSystem;
import javax.sound.midi.MidiChannel;
import javax.sound.midi.MidiUnavailableException;
import javax.sound.midi.Synthesizer;
import java.lang.Runnable;
import java.lang.Thread;

public class NotesPlayer implements Runnable {

    final static int VOLUME = 100;
    final static int PITCH = 0;
    final static int DUR = 1;
    final static int MEASURE_DUR = 2000; // Default measure duration

    public NotesPlayer(int[][] notes) {
        this.notes = notes;
    }

    @Override
    public void run() {
        try {
            playNotes();
        } catch (MidiUnavailableException e) {
            threadMessage("Midi Unavailable");
        } catch (InterruptedException e) {
            threadMessage("Thread interrupted.");
        }
    }

    private void playNotes() throws MidiUnavailableException, InterruptedException {
        Synthesizer synth = MidiSystem.getSynthesizer();
        MidiChannel[] channels = synth.getChannels();
        synth.open();
        for (int i = 0; i < notes[0].length; i++) {
            System.out.println("Playing: " + notes[PITCH][i]);
            if (notes[PITCH][i] == -1) {
                Thread.sleep(notes[DUR][i]);
            }
        }
    }
```
8.2 Demos

Row Your Boat

```java
int compose() {
    pitch[] pitches = "$C ** 3
    ++ $D ++ $E
    ++ $E ++ $D ++ $E ++ $F ++ $G
    ++ $C ** 3 ++ $G ++ $3
    ++ $E ++ $C ** 3
    ++ $G ++ $F ++ $E ++ $D ++ $C;
    dur dot8 = (3,16);
    dur trip8 = (1,4) / 3;
    dur[] durations = (1,4) ++ 2
    ++ dot8 ++ (1,16) ++ (1,4)
    ++ dot8 ++ (1,16) ++ dot8 ++ (1,16)
    ++ (1,2)
    ++ trip8 ++ 12
    ++ dot8 ++ (1,16) ++ dot8 ++ (1,16)
    ++ (1,2);
    phrase mainphrase = pitches ~ durations;
    note rest = ($R,(1,1));
    int i;
    score song = newscore();
    for(i = 0; i < 4; i = i + 1) {
        int j;
        phrase round = mainphrase;
        for(j = 0; j < i; j = j + 1) {
            round = rest + round;
        }
    }
```
```c
int compose() {
  dur d = (1,8);
  pitch[] pitches = $C ** 8;
  pitch base = $C0;
  pitch max = $C8;

  int i;
  int n = 40;

  // Initialize an array of pitches
  for(i = 0; i < 8; i = i + 1) {
    pitches[i] = base + i * 12;
  }

  // Main loop
  phrase ph = newphrase();
  for(i = 0; i < n; i = i + 1) {
    int j;
    for(j = 0; j < 8; j = j + 1) {
      pitches[j] = pitches[j] + 1;
      if(pitches[j] == max)
        pitches[j] = base;
    }

    // Put them all on top of one another
    chord ch = newchord(d);
    for(j = 0; j < 8; j = j + 1) {
      ch = ch ^ pitches[j];
    }
    ph = ph + ch;
  }
}
```

The Shepard Scale

```c
  song = song ^ round;
}
play(song);
}
chord newchord(dur d) {
  note n1 = ($R, d);
  return n1 ^ $R;
}
phrase newphrase() {
  chord c1 = newchord((0,1));
  chord c2 = c1;
  return c1 + c2;
}

score newscore() {
  phrase pl = newphrase();
  return pl ^ pl;
}
```
play(ph);

}

chord newchord(dur d) {
  note n1 = ($R,d);
  return n1 ^ $R;
}

phrase newphrase() {
  chord c1 = newchord((0,1));
  chord c2 = c1;
  return c1 + c2;
}