HOLLABACH
Final Report

COMS W4115: Programming Languages and Translators
Craig Darmetko (cd2698)
August 22, 2014

1. Introduction

HOLLABACH is a language for composing MIDI based music. With HOLLABACH, composers can create compositions similar to writing programming code and utilize common programming tools and resources with their compositions. The language also includes support for loops, conditional statements and variables for the often repetitive or algorithmic nature of musical compositions. HOLLABACH is designed to help composers create music efficiently, compose music collaboratively and abstract away the semantics of the MIDI specification.

1.1. MIDI

MIDI is a protocol and digital interface for digital music recording and playback. It was published in 1983 and has been the most popular protocol for composing and playing electronic music. MIDI is widely supported by most digital instrument creators and most modern computers have built in software to play back recorded MIDI music. HOLLABACH produces MIDI files as final compiled output.

2. Language Tutorial

A HOLLABACH composition is a group of instruments with a sequence of musical statements for each instrument. These statements can be musical measures, loop structures, conditional statements, variables or time signature declarations.

2.1. Hello World!

```
inst Piano {

    [C3/1]

}
```

This example program creates a composition that contains one instrument, a Piano, and has that instrument play a C note for four beats (one measure). The inst construct declares the instrument and the body of the construct is the musical sequence to be played by the instrument.

In this case we have one measure in the sequence, declared by the brackets, with one note in the measure. The note is a C note, as specified by the starting letter. The following digit specifies the octave and the trailing digit specifies the inverse of the length. A value of one means 1 measure/ 1 = 1 measure. If

the value had been a 4, the length would be 1 measure / 4 = ¼ measure, or a quarter note. Valid note lengths are discussed in more detain the section 3.2.2.

To make this more interesting, we can add more notes and measures to our original example:

```
inst Piano {

     [C3/1]

     [C3/4+G3/4 R C/3/4+G3/4 A3/4]

}
```

In this composition, we have the original measure, followed by a more complex measure. During this new measure, we have two quarter note chords, denoted by the + sign between the notes of the chord. We also have a rest note, denoted by the single character 'R'. The notes in the measure are played left to right and the starting beat in the measure is denoted by

Position / (total note positions in the measure)

In this case, the second measure has four positions, so the second chord(in position 3) would be played on beat 3 of the measure.

2.2. Compiling and running

Compiling a HOLLABACH program uses the executable hollabach. Hollabach generates the .holla bytecode by running:

```
$ ./hollabach –c mycomp.holla < mycomp.bach
```

From here, the bytecode can be translated into a MIDI file by running

```
$ cd CSV2JAVA; java ../mycomp.holla ../mycomp.midi;
cd ..
```

From here, the composition can be played using any MIDI compatible playback device or program.

2.3. More Examples

In this example, we create a simple composition of the folk song, Hot Cross Buns:

```
inst Piano {

  loop 2 {

         hcb = [C3/4 B3/4 A3/4 R]

  }

  [A3/4 A3/4 B3/4 B3/4]

  hcb

}
```

We see two new language features here, the loop and the variable. The loop allows us to repeat a section of code a set amount of times before continuing. In this case we repeat the enclosing measure twice. The enclosing measure is also declared bound to the variable hcb. We can then use hcb in the last line of the composition to insert that measure without having to retype it.

3. Language Manual

3.1. Lexical Conventions

3.1.1 Comments

A comment is preceded by '/*' and ended with '*/'. These comments may span several lines but cannot be embedded in another comment.

3.1.2 Identifiers

Identifiers are a sequence of letters, digits, and underscores. Identifiers cannot begin with a digit and cannot be a reserved keyword.

Identifier -> letter(letter | digit | underscore)

3.1.3 Reserved Keywords

The group of keywords that exist for defining language related function functionality are:

loop, timesig, inst, if, else

These keywords cannot be used for identifiers.

3.1.4 Whitespace

A whitespace character is used to separate tokens and is otherwise ignored. Whitespace characters include the space, tab and newline characters.

3.2. Basic types/constants:

3.2.1 Integers:

Integer Constants are a sequence of digits separated by whitespace. Some uses for integers include representing Time Signature declarations, If statement conditionals and loop iteration counts.

Constant -> digit+

3.2.2 Note

A Note is a single contiguous musical sound. An example note is:

```
A#3/4
```

This defines a note of pitch A# in octave 3 with a length of a quarter note (1/4). The backslash denotes the beginning of the declaration of note length. Length is determined by the denominator of the fraction of a measure it lasts, such as 4 for quarter note, 2 for half note, 8 for eigth note. A sixteenth note is specified with just ‘6’ to make note length consistent. Valid lengths include:

1 - whole note
2 - half note
4 - quarter note
8 - eighth note
6 - sixteenth note.

The available pitches include A-G and an additional # or b character to represent sharp or flat notes. The octave section can contain a single digit between 0 and

6. A rest note is declared with just the letter R. The overall lexical pattern for identifying a note is

Note -> [‘A’-‘G’, ‘R’][‘#’|’b’]?[‘0’-‘6’][‘/’][‘1’|’2’|’4’|’8’|’6’]

3.3. Complex types and Expressions

3.3.1 Chord

A chord is one or more notes that will be played simultaneously. An example chord:

```
C2/4+G3/4+A3/4+C3/4
```

This defines a chord of four notes that lasts for a quarter beat. Notes in the chord may end at different time. The goal is that all of the notes in a chord start at the same time.

chord -> note | note + chord

3.3.2 Measure

A Measure is a musical unit in a composition. It contains an array of chords and Rests. The position in the array indicates when the chord is played based on the length of the array. The array can be up to 32 units in length and it subdivides the measure into as many note start positions as there are entries. Valid lengths are 0, 1, 2, 3, 4, 8, 16, 32. For example,

```
[A2/8 A2/8 A2/8 A2/8]
```

will play an eighth note every quarter beat for the measure. The array is declared using an open bracket (‘[‘) and finished using a closed bracket (‘]’) and split using whitespace.

*Measure -> chord **

3.3.2.1 Declaring a Measure as a Variable

Measures may also optionally be declared with an identifier for reuse in later lines in a composition. For example:

```
aEveryFour = [A2/8 A2/8 A2/8 A2/8]
```

It can then be used in later lines with simply:

```
aEveryFour
```

If a measure identifier is redefined the compiler will override the current definition with the new definition, but all previous uses of it will remain unchanged.

3.3.3 Rests

A rest can be declared in a measure using the token 'R' in place of a note. No notes will be started at that position, although a previous note may sustain through the rest. Whole measure rests can be declared by omitting the body of the measure like so:

```
[]
```

3.3.4 Instrument

An Instrument creates a musical 'thread', a tone and series of notes that will be attributed to one musical instrument in a composition. The optional body contains a list of sequential measures and optional loop, conditional and time signature statements. Multiple Instruments are allowed per file. Each can be thought of as a different musician. Instruments are declared using the 'inst' tag.

```
inst Piano{
     timesig =4
     [G2/8 A2/8 B2/8 C2/8 D2/8 E2/8 F2/8 G3/8]
     loop 2 {
          [G2/4+C2/4+G3/4 R G2/4+C2/4+G3/4 R]
     }

}
```

The Instrument tone is declared after the inst tag and the Instrument definition appears between the '{' and '}' characters. Each measure, time signature change or loop is separated by a newline character and proceed sequentially.

*Instrument -> 'inst' string_lit '{' stmt * '}'*

3.3.5 Time Signature

Time Signature is an attribute that determines how many beats occur in a measure. Dividing this value by the beats per minute gives you the absolute time length of a single measure. Time Signature is declared with the tag 'timesig=' and defined with a positive integer that follows. An example declaration is:

```
timesig=4
```

After declaring a Time Signature value, all measures that follow it until another time signature is declared will inherit this value. If no value is declared in the

instrument, the value is assumed to be 4.

3.3.6 Loops

A loop statement allows a measure or set of measures to be repeated a given amount of times. A loop is declared with the keyword ‘loop’ and followed by an integer representing the number of repetitions. Following the integer, the loop contents are declared with a leading ‘{‘ and a trailing ‘}’. For example:

```
loop 2 {
          [G2/4+C2/4+G3/4 R G2/4+C2/4+G3/4 R]
}
```

A loop contents can include any combination of measures, if statements, variables or embedded loops.

*Loop -> ‘loop’ literal ‘{‘ stmt * ‘}’*

3.3.7 Conditional Statements

A composer can use an conditional statement to change the behavior of certain loop iterations in a loop. The composer declares the conditional using an If statement, followed by the loop iteration number that the If statement should trigger. The iteration starts at 0 and increments until it is equal to the specified number. This means an if statement meant to trigger on the first iteration should use 0 as it’s conditional. The composer can also use an else statement after an If to trigger on the negation case of the If statement. The If body is declared after the conditional between brackets. The body can contain measures or additional loops.

If -> ‘if’ literal ‘{‘ stmt* ‘}’ ‘else’ ‘{‘ stmt* ‘}’ | ‘if’ literal ‘{‘ stmt* ‘}’

3.3.8 Composition

A composition a single ’program’ in HOLLABACH. It is essentially a list of Instrument declarations and their definitions. The order of Instruments does not matter outside of the track ordering in the final MIDI output.

*Comp -> inst **

3.4. Variable Scope

3.4.1 Measure Identifier

The scope of a measure identifier in HOLLABACH is global to the composition once it is defined. Any references before it is defined in the composition will cause

compiler errors. If the same identifier is declared twice in a composition, the second value will override the original value for all uses after the second declaration.

3.4.2 Loop

A loop has an implicit variable, the loop iteration number. The scope of this variable is inside the loop (between the brackets) and is exclusively used in if statements. An if statement conditional always refers to the lowest loop. For example, if we have loop B embedded in loop A and an if statement in loop B, the conditional will refer to the iteration number of loop B. If an If statement is declared not inside a loop, it will only trigger if the conditional is 0, as the loop iteration number not inside a loop is always 0.

4. Project Plan

4.1. Project Timeline

The following time periods were set for major development objectives:

Date	Task
6/2- 6/11	Language Proposal
6/12- 7/2	Language Design and Reference Manual
7/3 - 8/8	Environment set up and development of End to End execution, completed Scanner and Parser
8/9 – 8/22	Feature expansion, Testing Framework development, composing Final Report

4.2. Programming Style Guide

Effort was made to document as much of the code as possible and try to adhere to the programming style guide created by the maintainers of Ocaml at:

http://caml.inria.fr/resources/doc/guides/guidelines.en.html

4.3. Development Environment

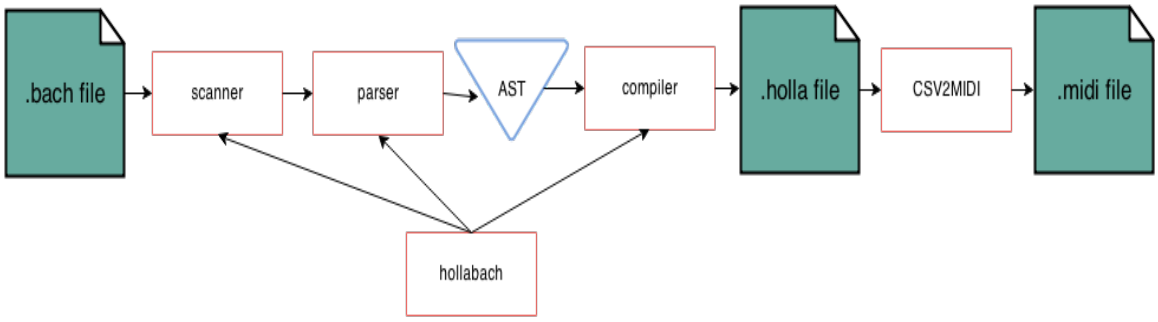
The project was developed on OSX using Git and Github for version control and Make for building. The compiler was built in Ocaml and the translator/interpreter was built in Java. Ocamllex was used **for lexing and**

ocamlyacc was used to create the grammar for the language. Shell scripting was used for executing automated tests.

4.4. Project Log

Date	Progress
6/2?	Project Start
6/11	Language Proposal finished
6/24	1st draft of Language Grammar
7/2	Language Reference Manual submitted
7/14	Revised Language Grammar based on feedback
7/18	Development Environment and Code Repository set up
8/11	End to End execution and first test programs created
8/12	MIDI output
8/19	Automated Test Suite working
8/something	**Added features and stuff**
8/22	Submitted final code and Report

5. Architectural Design



HOLLABACH follows a similar pattern to most compilers, using lexical analysis, a parser and a compiler to translate source code into bytecode. The system takes input of a .bach file of HOLLABACH source code and the scanner uses Ocamllex, a version of lex for Ocaml, to generate tokens. The parser then converts these tokens into an Abstract Syntax tree by rules defined using Ocamlyacc, a version of yacc for Ocaml. This AST is used as input to the compiler, which converts the data into bytecode that is easy for the interpreter to handle and then writes this data out to a .holla file. The holla file is

essentially a CSV file consisting of track and note information. This file is then passed into the interpreter, CSV2MIDI, which generates a MIDI file from the data.

5.1. Individual Contributions

Component	Contributors
Scanner	Craig Darmetko, Stephen Edwards
Parser	Craig Darmetko, Stephen Edwards
Compiler	Craig Darmetko
Interpreter (CSV2MIDI)	Craig Darmetko, Stephen Steffes
Testing	Craig Darmetko, Stephen Edwards

5.2 CSV2MIDI

The Interpreter and bytecode representation for HOLLABACH is largely based on CSV2MIDI, a java program written by Stephen Steffes to translate data in a CSV file to a MIDI file. HOLLABACH extended the original functionality to support some of the unique features in the HOLLABACH language.

6. Test Plan

6.1. Test Framework

Although not exhaustive, the test plan for HOLLABACH is meant to exercise important features in a simple and repeatable manner. HOLLABACH uses a series of test programs, detailed in Section 6.3, and a shell script that compiles the program and compares it against ground truth to report any discrepancies. Each test program exercises a certain feature or the composition of features. Having multiple separate test programs isolates errors and speeds up resolution of bugs. Tests can be run by executing the program testall.sh.

6.2. Example Program and Output

Below is an example program that plays the first verse of the song Chopstix with two instruments, a Piano and an Acoustic Bass.

```
inst Piano{
    timesig=3
    loop 2{
        loop 2{
            [G3/4+F3/4 G3/4+F3/4 G3/4+F3/4]
        }
        loop 2{
            [G3/4+E3/4 G3/4+E3/4 G3/4+E3/4]
        }
        [B3/4+D3/4 B3/4+D3/4 B3/4+D3/4]
        [B3/4+D3/4 A3/4+E3/4 B3/4+D3/4]

        [C3/4+C4/4 R C3/4+C4/4]
        if 1 {
         [C3/2+C4/2 R]
        }
        else{
         [C3/4+C4/4 B3/4+D3/4 A3/4+E3/4]
        }
    }
}
```

```
inst AcousticBass{
    timesig=3
    loop 2{
        loop 2{
         beg = [C3/8 G3/4+F3/4 G3/4+F3/4]
        }
        loop 2{
         [C3/8 G3/4+E3/4 G3/4+F3/4]
        }
        [D3/8 G3/4+F3/4 G3/4+F3/4]
        [D3/8 G3/4+E3/4 G3/4+E3/4]
        beg
        if 1 {
         [C3/2 R]
        }
        else{
         beg
        }
    }
}
```

This program generates the bytecode output below:

Timing Resolution (pulses per quarter note)							
4							
Instrument	0	Piano	Instrument	32	AcousticBass		
Tick	Note	Velocity	Length	Tick	Note	Velocity	Length
0	53	80	4				
0	55	80	4				
4	53	80	4				
4	55	80	4				
8	53	80	4				
8	55	80	4				
12	53	80	4				
12	55	80	4				
16	53	80	4				
16	55	80	4				
20	53	80	4				
20	55	80	4				
24	52	80	4				
24	55	80	4				
28	52	80	4				
28	55	80	4				
32	52	80	4				
32	55	80	4				
36	52	80	4				
36	55	80	4				
40	52	80	4				
40	55	80	4				
44	52	80	4				
44	55	80	4				
48	50	80	4				
48	59	80	4				
52	50	80	4				
52	59	80	4				
56	50	80	4				
56	59	80	4				
60	50	80	4				
60	59	80	4				
64	52	80	4				

64	57	80	4				
68	50	80	4				
68	59	80	4				
72	60	80	4				
72	48	80	4				
80	60	80	4				
80	48	80	4				
84	60	80	4				
84	48	80	4				
88	50	80	4				
88	59	80	4				
92	52	80	4				
92	57	80	4				
96	53	80	4				
96	55	80	4				
100	53	80	4				
100	55	80	4				
104	53	80	4				
104	55	80	4				
108	53	80	4				
108	55	80	4				
112	53	80	4				
112	55	80	4				
116	53	80	4				
116	55	80	4				
120	52	80	4				
120	55	80	4				
124	52	80	4				
124	55	80	4				
128	52	80	4				
128	55	80	4				
132	52	80	4				
132	55	80	4				
136	52	80	4				
136	55	80	4				
140	52	80	4				
140	55	80	4				
144	50	80	4				
144	59	80	4				
148	50	80	4				
148	59	80	4				

152	50	80	4				
152	59	80	4				
156	50	80	4				
156	59	80	4				
160	52	80	4				
160	57	80	4				
164	50	80	4				
164	59	80	4				
168	60	80	4				
168	48	80	4				
176	60	80	4				
176	48	80	4				
180	60	80	8				
180	48	80	8				
				0	48	80	2
				4	53	80	4
				4	55	80	4
				8	53	80	4
				8	55	80	4
				12	48	80	2
				16	53	80	4
				16	55	80	4
				20	53	80	4
				20	55	80	4
				24	48	80	2
				28	52	80	4
				28	55	80	4
				32	53	80	4
				32	55	80	4
				36	48	80	2
				40	52	80	4
				40	55	80	4
				44	53	80	4
				44	55	80	4
				48	50	80	2
				52	53	80	4
				52	55	80	4
				56	53	80	4
				56	55	80	4
				60	50	80	2
				64	52	80	4

				64	55	80	4
				68	52	80	4
				68	55	80	4
				72	48	80	2
				76	53	80	4
				76	55	80	4
				80	53	80	4
				80	55	80	4
				84	48	80	2
				88	53	80	4
				88	55	80	4
				92	53	80	4
				92	55	80	4
				96	48	80	2
				100	53	80	4
				100	55	80	4
				104	53	80	4
				104	55	80	4
				108	48	80	2
				112	53	80	4
				112	55	80	4
				116	53	80	4
				116	55	80	4
				120	48	80	2
				124	52	80	4
				124	55	80	4
				128	53	80	4
				128	55	80	4
				132	48	80	2
				136	52	80	4
				136	55	80	4
				140	53	80	4
				140	55	80	4
				144	50	80	2
				148	53	80	4
				148	55	80	4
				152	53	80	4
				152	55	80	4
				156	50	80	2
				160	52	80	4
				160	55	80	4

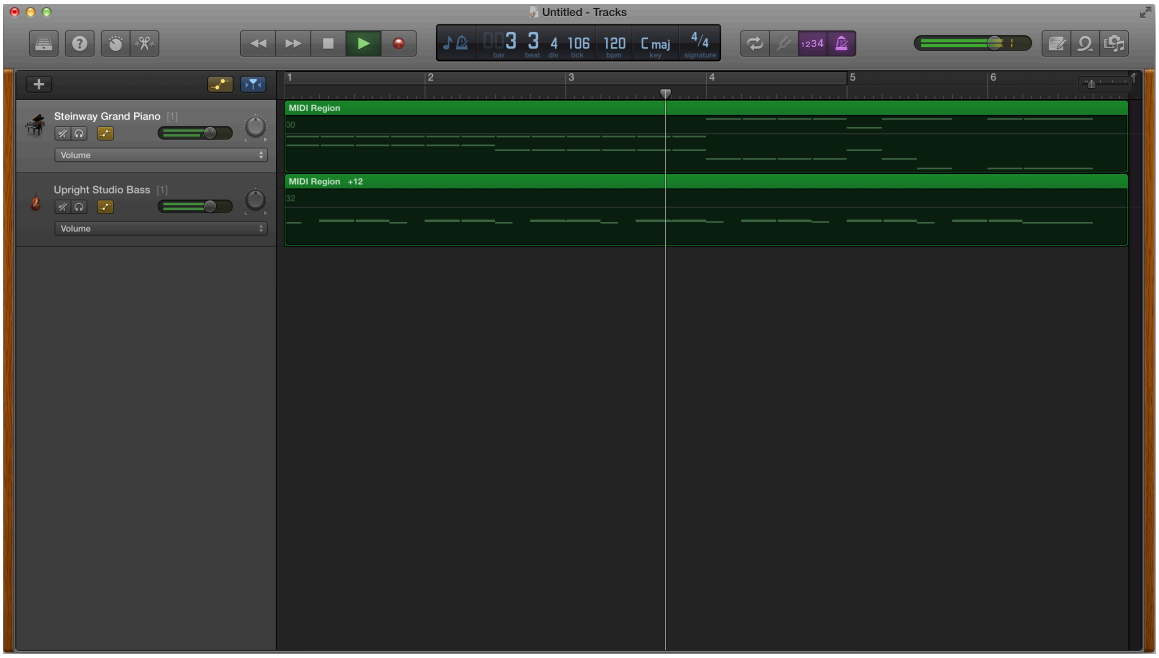
				164	52	80	4
				164	55	80	4
				168	48	80	2
				172	53	80	4
				172	55	80	4
				176	53	80	4
				176	55	80	4
				180	48	80	8

The first section of the bytecode is a static value representing the subdivision of a quarter note into 4 ‘pulses’. Thus, all length and offset(tick) values are 4 * the quarter note position.

The next section declares the instruments and their order in the note declaration body.

In the note declaration, each instrument gets four columns. For example, column 1 denotes the offset position for notes played by the first instrument and column 5 denotes offsets for the second instrument. Each note is given a row and allowed to specify an offset, pitch, loudness and length.

The bytecode is then compiled into a MIDI file. As MIDI files are not readable in binary form, I have not included the raw output, but a screenshot of the file executing in Garage Band can be found below:



6.3. Tests

Filename	Type	Output
test-hello.bach	Tests composition and measures generation.	A single instrument composition with one measure and one note.
test-empty.bach	Tests an empty instrument.	A valid composition with no notes.
Test-chord.bach	Tests use of chords in a measure	A composition with a two note chord
Test-chord-big.bach	Tests use of larger chords	A composition with a four note chord
Test-empty-meas.bach	Tests use of measures with no notes	A valid composition with two measures with notes and one measure without any.
Test-if.bach	Tests use of if statements	A valid composition based on input program
Test-inst.bach	Tests use of a different instrument	A composition with a Tenor Sax instrument
Test-len.bach	Tests changing the time signature	A composition with a measure length and time signature of 2 beats.
Test-loop.bach	Tests the loop construct	A composition with a measure repeated 4 times
Test-loop-rec.bach	Tests an embedded loop	A composition that repeats a measure progression 4 times. This progression also contains a measure that is repeated twice.
Test-mult.bach	Tests multiple measures	A composition that contains a sequence of two measures
Test-rest.bach	Tests rest notes	A composition with a measure that contains two one beat rests and two one beat notes.
Test-var.bach	Tests measure variable	A compositon with two

	declaration and use	measures that are identical
Test-prog.bach	Tests the composition of features	A valid complex composition that adhears to all of the constructs used.
Test-chopstix.bach	Tests the composition of features	A valid composition that reflects the original song

7. Lessons Learned

Because this was my first time working with Ocaml and it has many philosophical differences from my most used languages, it often took longer to write the code and get it running than I expected. Getting started earlier would have allowed me to work at a more reasonable pace by giving me a buffer to handle the Ocaml issues I ran into related to errors and debugging without running into deadlines. I found many of the error messages from Ocaml opaque and misleading, but this may be because of my inexperience with the language.

Also, looking at the lessons learned of future projects would be very helpful to do near the start of the project, rather than when you are writing your final report.

8. Appendix

Code listing of all the Ocaml files and the main CSV2MIDI.java file. The whole code repository and commit log is also available at:

https://github.com/sinflood/HOLLABACH

8.1. hollabach.ml

```
open Printf

type action = Ast | Compile

exception LexErr of string

let _ =

    let action = if Array.length Sys.argv > 1 then
```

```
    List.assoc Sys.argv.(1) [ ("-a", Ast);
                              ("-c", Compile) ]
  else Compile in

    let outfile = Sys.argv.(2) in
   (*remove the output file if it exists as it is likely a past
version*)
    let delIfExists o =
                if Sys.file_exists o then Sys.remove(o) else ()
    in

  let lexbuf = Lexing.from_channel stdin in
  let program =
         List.rev (Parser.program Scanner.token lexbuf) in
  match action with
    Ast -> let listing = Ast.string_of_program program
           in print_string listing
  | Compile -> delIfExists outfile;Compile.compile program outfile;()
```

8.2. scanner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '['      { LBRACK }
| ']'      { RBRACK }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "if"     { IF }
| "else"   { ELSE }
| "loop"   { LOOP }
| "inst"   { INST }
| "timesig" { TIMESIG }

| ['R'] | (['A'-'G']['#' 'b']?['0'-'6']['/'](['1' '2' '4' '8' '6' ]))
as lxm {
        NOTE(lxm) }

| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
```

```
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char))
}

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

8.3. parser.mly

```
%{ open Ast %}
%{ open Printf %}
%{ open Parsing %}
%{ open Lexing %}
%token SEMI LBRACK RBRACK LBRACE RBRACE COMMA LPAREN RPAREN
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token IF ELSE LOOP
%token INST BPM TIMESIG
%token <string> NOTE
%token <int> LITERAL
%token <string> ID
%token EOF
%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%left TIMESIG

%start program
%type <Ast.program> program

%%

program:
        | inst { [$1] }
        | program inst { $2 :: $1 }

inst:
        INST ID LBRACE stmt_list RBRACE {{instStr=$2; body=(List.rev
$4)} }

mdecl:
    LBRACK note_list RBRACK    { { id = "none";
                body = List.rev $2; timesig = 4; } }
    | ID ASSIGN LBRACK note_list RBRACK   { { id = $1;
                    body = List.rev $4; timesig = 4; } }
    | LBRACK RBRACK {{ id="none"; body=[]; timesig=4;}}
    | ID ASSIGN LBRACK RBRACK {{id=$1;body=[]; timesig=4}}
    | error             { raise(Failure("Malformed measure" ) );  }
```

```
note_list:
       chord { [Chord($1)]}
      | note_list chord { Chord($2) :: $1 }
      | error             { raise(Failure("Malformed note")) }
note_plus:
        NOTE { [Note($1)] }
      | note_plus PLUS NOTE { Note($3) :: $1 }
chord:
        note_plus { $1 }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt {  $2 :: $1 }
stmt:
        IF  expr LBRACE stmt_list RBRACE %prec NOELSE { If($2,
        (List.rev $4), []) }
  | IF  expr LBRACE stmt_list RBRACE ELSE LBRACE stmt_list RBRACE
  { If($2, (List.rev $4), (List.rev $8)) }
  | LOOP LITERAL LBRACE stmt_list RBRACE { Loop(Literal($2), (List.rev
$4)) }
  | mdecl                        {  Measure($1) }
  | TIMESIG ASSIGN LITERAL { TimeSig($3)}
  | ID     { Id($1) }

expr:
    LITERAL          { Literal($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
```

8.4. ast.ml

```
open Lexing
open Parsing
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater |
Geq

type note =
        Note of string
type expr =
    Literal of int
  | Chord of note list
  | Binop of expr * op * expr
  | Assign of string * expr
  | Noexpr

type meas_decl = {
```

```
        id : string;
        body :  expr list;
        mutable timesig : int;
}
type stmt =
  | Expr of expr
  | If of expr * stmt list * stmt list
  | Measure of meas_decl
  | Loop of expr * stmt list
  | TimeSig of int
  | Id of string


type  inst = {
        instStr : string;
        body: stmt list;
}
type program = inst list


let rec string_of_note = function
        Note(n) -> n
let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Chord(c) -> String.concat "+" (List.map string_of_note c)
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
    Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=") ^ " "
^
      string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Noexpr -> ""

let string_of_meas_decl md =
        md.id ^ "[" ^ String.concat "," (List.map string_of_expr
md.body) ^ "]"
let rec string_of_stmt = function
   Expr(expr) -> string_of_expr expr ^ ";\n";
  | Id(s) -> s
  | TimeSig(m) -> "timesig = " ^ string_of_int m
(*  | If(e, s, []) -> "if (" ^ string_of_expr e ^ ")\n" ^
string_of_stmt s*)
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
          String.concat "\n" (List.map string_of_stmt s1) ^ "else\n" ^
          String.concat "\n" (List.map string_of_stmt s2)
  | Loop(e, s) -> "loop " ^ string_of_expr e ^ " { " ^ String.concat
"\n"
  (List.map string_of_stmt s) ^ " }"
  | Measure(m) -> string_of_meas_decl m


let string_of_comp (stmts) =
        String.concat "" (List.map string_of_stmt stmts.body) ^ "\n"
```

```
let string_of_program comps =
  String.concat "\n" (List.map string_of_comp comps)
```

8.5. compile.ml

```
open Ast
open Printf

module StringMap = Map.Make(String)
let header  = "Timing Resolution (pulses per quarter note)\n4\n\n"

(* creates the CSV header*)
let writeHeader fname insts = let oc= open_out_gen [Open_creat;
Open_wronly; Open_append;
Open_text] 0o666 fname in
fprintf oc "%s%s" header insts; close_out oc

(*writes out the measure contents to the CSV out file. Expects the
input to be a
 * list of measures, the outfile name and an integer representing the
 * instrument's position. *)
let writeOutput k  fname trackNum= let oc = open_out_gen [Open_creat;
Open_wronly;
Open_append; Open_text] 0o666 fname in
(* returns the duration of the note in ticks*)
let getLen notestr =
        match notestr.[ (String.length notestr) -1] with
        '1' -> 16 (* whole note*)
        | '2' ->8
        | '4' -> 4
        | '8' -> 2
        |  '6' -> 1 (*16th note*)
        | _ -> raise (Failure("Invalid note length on " ^ notestr))
in
(*adjusts pitch for sharps and flats*)
let getMod noter =
        match noter.[1] with
        'b' -> -1
        | '#' -> 1
        | _ -> 0
in
(* returns the modifier for the note octave *)
let getOctave noter =
        if getMod noter != 0 then
                ((int_of_char noter.[2] -48)* 12 + 12)
        else
                (int_of_char noter.[1] -48)* 12 + 12
in
(*translates the letter note to the integer value expected by CSV2MIDI.
Includes
 * Octave. *)
let getNote noter =
        match noter.[0] with
        'A' -> 9 + getOctave noter
```

```
        | 'B' -> 11 + getOctave noter
        | 'C' -> 0 + getOctave noter
        | 'D' -> 2 + getOctave noter
        | 'E' -> 4 + getOctave noter
        | 'F' -> 5 + getOctave noter
        | 'G' -> 7 + getOctave noter
        | _ -> raise (Failure ("Not a note value! Note: " ^ noter))

in
(*Adds empty columns for previous tracks to the row*)
let rec getPrefix k =
        if k = 0 then""
        else ",,,,"^ getPrefix (k-1)
in
(* Returns the CSV line for a note*)
let getNoteString no offset =
         match no with
         Note(n) -> getPrefix trackNum ^ string_of_int offset ^ "," ^
string_of_int ((getNote n) +
         (getMod n)) ^ "," ^ string_of_int 80 ^ "," ^ string_of_int
(getLen n)
in
(*prints a note to the CSV file*)
let printNote naw offset=
        match naw with
        Note(n) -> if n.[0] != 'R' then
                fprintf oc "%s\n" (getNoteString naw offset)
in
(* prints a whole measure to the CSV file*)
let printMeasure off n =
        match n with
        Measure(m) ->
                let noteOff = (float_of_int m.timesig /. float_of_int
                (List.length m.body)) *. 4.0 in
                List.fold_left (fun note_num naw ->
                        match naw with
                        (*Note(n) -> fprintf oc "%s\n" (getNoteString
naw (off+note_num)); note_num + noteOff
                        |*) Chord(cr) -> List.iter (fun nat ->
printNote nat
                        (off+note_num)) cr; note_num + int_of_float
noteOff
                        | _ -> raise(Failure("Trying to print something
that
                        isn't a measure!"))
        )  0 m.body; off + (m.timesig * 4)
        | _ -> raise (Failure("ERROR, NOT A MEASURE! value:" ^
        string_of_stmt n))
in
List.fold_left printMeasure 0 k ;
close_out oc

(*contains all of the encountered varables and values for them *)
let  vars = ref StringMap.empty

(*This function is the main compiler and is called by hollabach.ml. It
will
```

```
 * evaluate/expand the input stmts and write out the bytecode to the
output file
 * outfile*)
let compile stmts outfile =
        (*evaluates expressions*)
        let rec eval env = match env with
                Literal (l) -> l
               (* | Note(n) -> Note(n), env*)
                | _ -> raise (Failure("Currently only handles literals
for
                conditionals."))
        in
        (*returns the integer representation expected by CSV2Midi for
an instrument*)
        let getInstr i =
        match i with
        "Banjo" -> "105"
        | "Clarinet" -> "71"
        | "AcousticBass" -> "32"
        | "AltoSax" -> "65"
        | "BagPipe" -> "109"
        | "Flute" -> "73"
        | "Piano" -> "0"
        | "TenorSax" -> "66"
        | "Trombone" -> "77"
        | "Trumpet" -> "56"
        | "Violin" -> "40"
        | _ -> "0"
in
 (*returns the line for declaring instruments in CSV2MIDI *)
let getInstrumentLine tracks =
        List.fold_left (fun s c ->
                ("Instrument,"  ^ (getInstr c.instStr) ^ "," ^
c.instStr ^ ",") ^ s ) "" tracks
in
(*returns the string for the column names in the note declaration
section of
 * CSV2MIDI *)
let rec getColumnNames inst_count =
        if inst_count > 0 then
                "Tick,Note(0-127),Velocity(0-127),Length,"
^getColumnNames (inst_count -1)
        else
               ""
in

(* the current time signature value during execution. This can be
modified by
 * encoutering TimeSig statements *)
let currTimeSig = ref 4
in
(*helper function for validating and processing a Measure statement.
Will check
 * if the length is valid. If true, it will add itself to the variable
list,
 * alter it's Time Signature and append itself to the output *)
let processMeasure me outp =
```

```
        let meas = Measure(me) in
        match meas with
        Measure(m) -> if (List.mem (List.length m.body) [0;1; 2; 3; 4;
8; 16; 32]) then
                ((vars := StringMap.add m.id m !vars); m.timesig<-
!currTimeSig;
                Measure(m) :: outp) else raise(Failure("Malformed
measure. Incorrect number of notes/chords
in the measure. Count:" ^ string_of_int (List.length m.body)))

        |_ -> outp
in
        (*evaluate a statement and return the updated measure
progression *)
        let rec exec ite out env = match env with
                Measure(m) -> (processMeasure m out)
                | TimeSig(m) ->  currTimeSig := m; out
                | Loop(c, b) -> let rec callLoop i body =
                        if i>=0 then
                            (List.fold_left (exec i) [] body) @
(callLoop (i-1)
                            body)
                        else []
                        in   (callLoop ((eval c)-1) b) @ out
                | Id(i) -> Measure((StringMap.find i !vars)) :: out
                | If(i,b,eb) -> if (eval i) = ite then
                        (List.fold_left (exec ite) [] b) @ out
                else (List.fold_left (exec ite) [] eb) @ out
                | a -> out
        in
        writeHeader outfile ((getInstrumentLine (List.rev stmts)) ^
"\n\n" ^ getColumnNames
        (List.length stmts) ^ "\n");
        List.fold_left (fun i c ->
        let comped = List.fold_left (exec 0) [] c.body
        in
        writeOutput (List.rev comped) outfile i; i+1 ) 0 stmts;()
```

8.6. CSV2MIDI.java

```
/**
 * CSV2MIDI.java
 * June 11, 2003
 * @author: Stephen Steffes
 * Purpose:  Converts a .csv file to a MIDI file according to
ExampleMIDI.csv
 */


import java.io.*;
import javax.sound.midi.*;
import java.lang.*;


public class CSV2MIDI{
```

```
    public static void main(String[] args)  throws
InvalidMidiDataException {

        //***** Get Inputs *****
        if (args.length != 2)
            printUsageAndExit();

        File outputFile = new File(args[1]);
        Sequence sequence = null;

        //Open and save the CSV file
        CSV csvFile=new CSV(args[0]);
        csvFile.fillVector();


        //figure out how many channels there are
        //nChannels=number of integers in the first line containing any
numbers, skipping the first number encountered
        int nChannels=0,temp=0;
        for(int i=0;i<csvFile.data.size();i++){
            try{
//check if this is an integer
                Integer.parseInt(csvFile.data.elementAt(i).toString());
                temp++;
//counts number of instruments
            }catch(NumberFormatException e){
//not a number
                if(temp>1){
//if other than first number

if(csvFile.data.elementAt(i).toString().compareTo("\n")==0){
//if a new line
                        nChannels=temp-1;
                        break;
//found nChannels, so stop for loop.  this is the number of instruments
counted
                    }
                }
            }
        }


/*      for(int i=0;i<csvFile.data.size();i++)
            System.out.println(csvFile.data.elementAt(i));
*/


        //***** Read in timing resolution and instruments *****
        int currentCSVPos=0, timingRes=1, instrument[]=new
int[nChannels];

        //read in timing resolution
        for(;currentCSVPos<csvFile.data.size();currentCSVPos++)
            try{
//check if this is an integer

timingRes=Integer.parseInt(csvFile.data.elementAt(currentCSVPos).toStri
```

```
ng()); //this is the first number, therefore, it's the timing
resolution
                System.out.println("\nTiming Resolution set to
"+timingRes+" PPQ\n");
                currentCSVPos++;
                break;
            }catch(NumberFormatException e){
            }

        //read in instrument numbers
        temp=0;
        for(;currentCSVPos<csvFile.data.size();currentCSVPos++)
            try{
//check if this is an integer

instrument[temp]=Integer.parseInt(csvFile.data.elementAt(currentCSVPos)
.toString()); //this is a number, it has to be an intrument
                System.out.println("Instrument set to
"+instrument[temp]+" on channel "+temp);
                temp++;
                if(temp>=nChannels){
//collect numbers until you've reached the number of channels
                    currentCSVPos++;
                    break;
                }
            }catch(NumberFormatException e){
            }


        //***** Initialize Sequencer *****
        try{
            sequence = new Sequence(Sequence.PPQ, timingRes);
//initialize sequencer with timingRes
        }catch (InvalidMidiDataException e){
            e.printStackTrace();
            System.exit(1);
        }


        //***** Create tracks and notes *****
        /* Track objects cannot be created by invoking their
constructor
           directly. Instead, the Sequence object does the job. So we
           obtain the Track there. This links the Track to the Sequence
           automatically.
        */
        Track track[] = new Track[nChannels];
        for(int i=0;i<nChannels;i++){
            track[i]=sequence.createTrack();
//create tracks

            ShortMessage sm = new ShortMessage( );
                sm.setMessage(ShortMessage.PROGRAM_CHANGE, i,
instrument[i], 0);  //put in instrument[i] in this track
                track[i].add(new MidiEvent(sm, 0));
        }
```

```
        int channel=0,note=0,tick=0,velocity=90,column=0,length=0;

        //go through each of the following lines and add notes
        for(;currentCSVPos<csvFile.data.size();){
//loop through rest of CSV file
            try{
//check that the current CSV position is an integer

tick=Integer.parseInt(csvFile.data.elementAt(currentCSVPos).toString())
;  //first number is tick
                currentCSVPos+=2;

note=Integer.parseInt(csvFile.data.elementAt(currentCSVPos).toString())
;  //next number is note
                currentCSVPos+=2;

velocity=Integer.parseInt(csvFile.data.elementAt(currentCSVPos).toStrin
g());  //next number is velocity
                currentCSVPos+=2;

length=Integer.parseInt(csvFile.data.elementAt(currentCSVPos).toString(
));
                currentCSVPos++;
                channel=column/4;
                column+=2;

track[channel].add(createNoteOnEvent(note,tick,channel,velocity));
//add note to this track

track[channel].add(createNoteOffEvent(note,tick+length,channel));
            }catch(NumberFormatException e){
//current CSV position not an integer

if(csvFile.data.elementAt(currentCSVPos).toString().compareTo("\n")==0)
{  //if it's a new line
                    column=0;
//go back to 1st column
                }else
if(csvFile.data.elementAt(currentCSVPos).toString().compareTo(",")==0){
//if it's just a comma
                    column++;
                }
                currentCSVPos++;
            }
        }


        // Print track information
        System.out.println();
        if ( track != null ) {
            for ( int i = 0; i < track.length; i++ ) {
                System.out.println( "Track " + i + ":" );

            for ( int j = 0; j < track[i].size(); j++ ) {
                    MidiEvent event = track[i].get( j );
                    System.out.println(" tick "+event.getTick()+",
"+MessageInfo.toString(event.getMessage()));
```

```
                } // for
            } // for
        } // if


        /* Now we just save the Sequence to the file we specified.
           The '0' (second parameter) means saving as SMF type 0.
           (type 1 is for multiple tracks).
        */
        try{
            MidiSystem.write(sequence, 1, outputFile);
        }catch (IOException e){
            e.printStackTrace();
            System.exit(1);
        }
    }




  //turns note on
    private static MidiEvent createNoteOnEvent(int nKey, long lTick,int
channel,int velocity){
        return
createNoteEvent(ShortMessage.NOTE_ON,nKey,velocity,lTick,channel);
    }

    //turns note off
    private static MidiEvent createNoteOffEvent(int nKey, long
lTick,int channel){
        return
createNoteEvent(ShortMessage.NOTE_OFF,nKey,0,lTick,channel);  //set
note to 0 velocity
    }

    //turns note on or off
    private static MidiEvent createNoteEvent(int nCommand,int nKey,int
nVelocity,long lTick,int channel){
        ShortMessage message = new ShortMessage();
        try{
            message.setMessage(nCommand,channel,nKey,nVelocity);
        }catch (InvalidMidiDataException e){
            e.printStackTrace();
            System.exit(1);
        }
        MidiEvent event = new MidiEvent(message,lTick);
        return event;
    }

    private static void printUsageAndExit(){
        out("usage:");
        out("java CSV2MIDI <infile.csv> <outfile.midi>");
        System.exit(1);
    }
```

```
    private static void out(String strMessage){
        System.out.println(strMessage);
    }
}
```