Henry Mattinson
Christ's College

Computer Science Tripos - Part II Project

Excello: End-user music programming in Excel

April 18, 2019

Chapter 1

Implementation

This chapter shall first explain how turtles are defined along with the remaining features of the initial prototype. The format and results of the formative evaluation using this initial prototype shall be summarised. I shall then cover the design decisions and changes that were made to the Excello prototype during this participatory design process. Then, the technical details of Excello and the MIDI to Excello converter will be covered. Concluding with an overview of the project repository.

1.1 Initial Prototype

Notes and turtles can be defined in any cell. The interpretation of cell contents by turtles is shown in table 1.1. When the Excello add-in is opened, a window will open in the right side of Excel. A play and stop button can be used to launch all the turtles defined in the spreadsheet and initiate playback. Playback is a realistic piano sound.

1.1.1 Turtles

Turtles are defined as follows:

```
!turtle(<Starting Cell>, <Movement>, <Speed>, <Number of Loops>)
```

Table 1.1: Definition of notes in cells.

Interpretation	**Format**
Note	Note name (A-G), optional accidentals and octave number e.g. `F#4`
Sustain	`s`
Multiple notes subdivided in time	Notes, rests or sustains separated by a comma. Rests must be a space or an empty string e.g. `E4,,C4,s`
Rest	Any cell not interpreted as a note, sustain or multi-note.

Activation

"`!`" dictates that the turtle will be activated when the play button is pressed. Just as digital audio workstations allow muting and soloing of tracks, this can be used to quickly modify which turtles will play without losing their definitions.

Starting Cell

The starting cell of the turtle, which is also played, is given by the cell reference. As with Excel formulae, this is a concatenation of letters for the column and numbers for the row.

As each turtle only plays one note at a time, multiple turtles must be defined to play polyphonic music such as chords. It was believed that user would define turtles following identical paths but in adjacent rows or columns. Multiple turtles following identical paths but starting from adjacent cells can be defined using the existing Excel range notation for the starting cells. "`A2:A5`" would define four turtles in the cells A2,A3,A4,A5. This prevents the writing of multiple turtle definitions differing in only the start cell row.

Movement

Turtles start facing north. The language for programming turtle movement has been discussed in the preparation chapter. Using brackets to repeat movements within the turtle's instructions was not implemented by the start of the participatory design process.

Speed

An optional third argument declares the speed the turtle moves through the grid relative to 160 cells per minute. The default is 1 corresponding to 160 cells per minute. If the argument "`2`" was provided, this would move through the grid at 320 cells per minute. This relative system was used so it would be easier to tell the speed relation between two turtles. This would be particularly beneficial for phase music. Arbitrary maths can be provided for this argument allowing turtles' speeds to be irrational multiples of each other.

Number of Loops

An optional fourth argument defines the number of repetitions of the turtle's path. By default, the turtle will loop infinitely. This was included so that repeating parts (e.g. the cello of Pachabel's Canon in D) need only defining once.

1.1.2 Highlighting

To assist in the recognition of notes and turtles, when the play button is pressed, cells are highlighted, conditional on their contents. Cells containing activated or deactivated turtle definitions are highlighted green. Cells containing definitions of notes, or multiple notes,

are highlighted red. Sustain cells are highlighted a lighter red, to show a correspondance to notes whilst maintaining differentiation.

1.1.3 Chord input

In order to use musical abstractions of chords and arpeggios[1] whilst keeping the paradigm of a turtle being responsible for up to one note at any time, a tool to add them is included. The note, type, inversion and starting octave of the chord are inputted in four drop-down selectors and the insert button enters the notes making up that chord into the grid. If a single cell or range taller than it is wide is highlighted in the spreadsheet, the notes will be inserted vertically starting at the top-left of the range. Otherwise, the notes will be inserted horizontally. This means whether the turtles are moving horizontally or vertically both chords and arpeggios can be easily defined. Thus, helpful musical abstractions are still available whilst keeping the cleanness of the turtle system.

1.2 Formative Evaluation

To guide development to best suite users, participants were involved in formative evaluation. 21 participants took place in the participatory design process. Participants were all musical University of Cambridge students, across a range of subjects. Initially, one-on-one tutorials of the initial prototype were given followed by the user carrying out of a short exercise. After both the tutorial and the exercise, users were interviewed on how they found Excello, drawing particular attention to actions that they found particularly unintuitive or requiring notable mental effort. Comparisons were made to the musical interfaces that participants were already familiar with. The sessions were audio recorded in order to prevent the jotting down of notes causing delays. Notes were later made from these recordings. The ethical and data handling procedures shall be discussed in the evaluation chapter.

For realistic simulation of the ways in which Excello would be used, participants were given the freedom to carry out an exercise of their choice. In many cases this was transcribing an exiting piece from memory or traditional western notation into the Excello notation. Two tasks were provided to choose from if participants had no immediate inspiration; transcribing a piece of western notation music or making changes to existing Excello notation.

These sessions were carried out at the beginning of Lent term 2019. Participants were asked if they would be willing to continue using Excello personally until the summative evaluation sessions, eight weeks later. This allowed additional feedback to be given as participants used Excello in their own time. It also ensured that the summative evaluation would be done using users with sufficient experience of the interface.

[1] Where the notes of a chord are played in rising or descending order

1.2.1 Issues and Suggestions

The issues and suggestions from the participatory design process have been summarised below.

Turtle Notation

Dynamics in the turtle instructions made it harder to extract the turtle's path as not all instructions related movement. As the dynamics weren't next to the notes they corresponded to, it was challenging to know the of volume a note or where to place the dynamics within the turtle to affect a certain note elsewhere in the spreadsheet. The initial prototype had no way to assign a dynamic to the first note without having the starting cell being empty. This empty cell could be inconvenient for looping parts as it would be included in the loop. Users not familiar with the dynamics of western notation found them unintuitive. Furthermore, these discrete markings do not make available the continuous volume scale possible with the interface.

When transcribing a piece with exact tempo, dividing the speed by 160 to enter the relative speed caused unnecessary work. There was also forgetfulness as to whether relative speed referred to how long the turtle spent in each cell or how quickly the turtle moved. Having completed the tutorial, users often had to check the position and meaning of turtle arguments.

As the number of dynamics and movement instructions grew, the instructions became long and it could be tough to parse and then establish turtle behaivour. Also, becuase "`s`" could be used to indicate sustain within cells, some users confused the "`s`" within the turtle instructions to mean sustain and not south.

Feedback

It was often unknown if pressing play registered, especially if the Excel workbook was saving delayed Excello being able to access the spreadsheet. Users also requested to see a summary of where the active turtle were in addition to the highlighting. If a turtle had accidentally been left activated, the entire grid had to be searched in order to locate it.

MIDI conversions

Many users who use production software said importing and exporting MIDI files would be helpful. If working with an existing MIDI file, it would be convenient to be able to convert that into the Excello notation. Exportation would allow Excello to be used to create chord sequences, bass lines and the piece structure, before adding additional effects and recorded lines in their digital audio work stations.

Sources of effort when writing

Once notes had been inputted into the grid, the number of cells the turtle had to move had to be counted. This is often in a straight line. Whilst the Excel status bar allows users to highlight a selection of cells and immediately see how many cells are highlighted, this is unproductive. This was particularly inconvenient when users were writing out a piece and periodically testing what they had written so far. Some users simply instructed turtles to move forward significantly more steps than required to prevent this counting. This is not feasible for looping parts. It was suggested that turtles figure out how far they should move.

Instructions involving repeated movements such as moving to the end of a line and jumping down to the beginning of a line below, instructions within the turtles required a lot of repetition.

Many of the notes in melodic lines would take place in the same octave. As such, repeatedly writing out the octave number was tiresome. One user made a comparison to LilyPond [4] where if the length of a note is not defined, the last defined note length would be used.

Some users find it more intuitive to think of a melodic line as the intervals between notes as opposed to note names. A modulated[2] melody line required the modulated part to be written out again and could not be derived quicker from the original version.

Chords

Most users used a very small subset of the available chords but had to scroll through the whole list to find these. Separating the more common chords for easier access was requested. Initially, notes inserted vertically had the lowest note at the top with notes increasing in pitch proceeding down the column. In western staff notation, higher pitch notes appear higher up the staff. As a result, it was suggested that inverting the order would be more intuitive. In the initial interface it was also unclear what the different drop downs corresponded to, with some users selecting the 7 from the octave number in order to try and insert a Maj7 chord.

Activation of turtles

When toggling the activation of a turtle, entering the edit mode for each cell containing a turtle definition to add or remove the exclamation mark was very tedious.

1.3 Second Prototype

Following the formative evaluation sessions and feedback, a series of additions and modifications were made to solve the problems and opportunities brought up.

[2]Where every note has been moved up or down in pitch by the same amount.

1.3.1 Dynamics

To help extract the path that the turtles follow and pair notes with their volume, dynamics are instead inserted in the cells along with the notes. A dynamic instruction is added after the note, separated by a space as in Manhattan [3]. As before, this will persist for all following notes until the volume is redefined. A single turtle definition with multiple start cells can now play parts of different volume. However, notes in the grid are limited to only being played at their given volume. To play the same notes at a different volume, a new path must be made where the cells defining the volume are replaced. Overall, the new system was believed to be more preferable.

In order to be able to make use of a full continuous dynamic scale, in addition to the existing dynamic symbols, a number between 0 and 1 can be provided where 0 will be silent and 1 is equivalent to fff.

1.3.2 Nested Instructions

Nested instructions with repetitions reduces the length of turtle instructions and allows for repeated sections or movements to be more easily incorporated. A series of instructions placed within parentheses with a number immediately following the closing parenthesis will be repeated that number of times. Whilst the fourth argument of the turtle will simply repeat the entire musical output of the turtle, repetitions within the turtle instruction allow paths to be defined more concisely.

1.3.3 Absolute Tempo

The turtle's speed is defined by cells per minute, rather than the relative value used initially. However, values less than 10 were interpreted in the original relative way to maintain backwards compatibility for the participant's existing work. To maintain consistency in a production version, this is removed so speed must be defined absolutely. The values given for speed and dynamics will be of different orders of magnitude and hence reduce the confusion that can occur between them.

1.3.4 Custom Excel Functions

Two custom Excel functions were implemented to aid composition. One to insert turtles into the grid and a second to transpose notes.

Excello.Turtle

By adding custom Excel functions, the existing formulae writing tools provided within Excel can be utilised. When using a built in formula, a prompt appears informing users which arguments go where and whether they are optional. The output of this function is text used to define a turtle if written manually. Other cells can be referenced for the arguments of the turtle function. For example all turtles could reference a single cell for their speeds. This allows relative tempos to be easily implemented as the speed argument

of each turtle could be a relative speed mutliplied by this global speed as shown in figure 1.1.

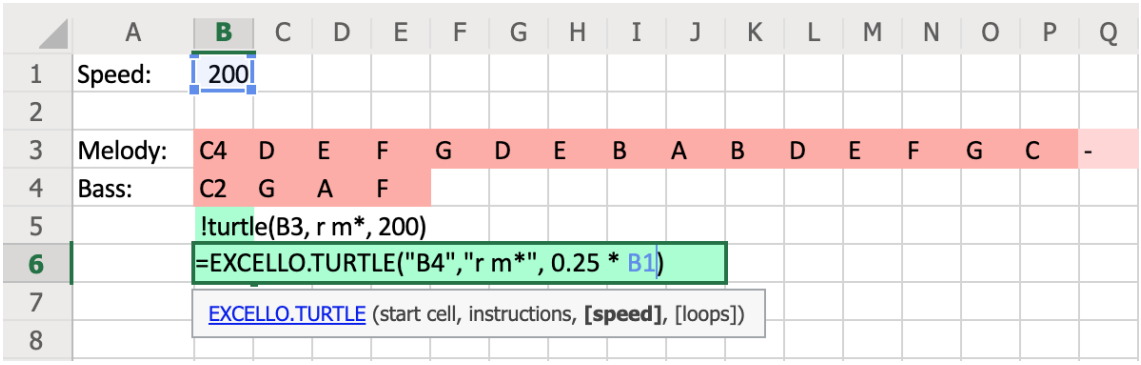

Figure 1.1: Defining a turtle using the EXCELLO.TURTLE function.

Excello.Modulate

A function to modulate notes provides easy modulation of existing sections of a piece and also the definition of a melodic line by the intervals between the notes. The function takes a cell and an interval and outputs any notes defined in that cell transposed by that interval, maintaining any dynamic definition. A section can be modulated by calling this function on the first note with a provided interval and using the existing drag-fill functionality of Excel to modulate all notes. By using the previous note that has just been transposed and one of a series of intervals as the arguments, a melodic line can quickly be produced from a starting note and a series of intervals as shown in figure 1.2.

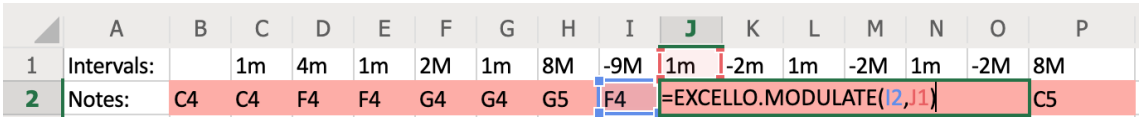

Figure 1.2: Transposing notes using the EXCELLO.MODULATE function.

1.3.5 Sustain

To prevent confusion between the instruction for a turtle to face south and sustains. The symbol "`-`" sustains a note. This was chosen because it is light and also has some similarity to a tie[3]. "`s`" in a cell is still interpreted as a sustain to maintain backwards compatibility for the existing work of the participants.

1.3.6 Active Turtles

In order to provide feedback that turtle definitions have been recognised, a list of the active turtles is given below the play button. This also helps find spurious turtles that were not intended to be activated.

[3]A line to increase the length of a note by joining to another.

1.3.7 Automatic Movement

To prevent counting the number of cells in a line, `m*` instructs a turtle to move as far as there are notes defined in the direction it is currently facing. If more notes are added on this line, the turtle instructions do not need editing before pressing play. There may be cases where a part is meant to finish with a number of rests. As a rest is notated with a blank cell, a method of increasing the length of the path to include these rests is required. A cell can be explicitly defined as a rest with ".". This would be required if multiple turtles were defining a repeating section where one does not have the final cell of the section being a note, sustain or multi-note cell. Without an explicit rest the turtle would stop and repeat too soon and the parts would be out of phase.

1.3.8 Inferred Octave

The octave number can be inferred by the program if omitted. Two methods were under consideration. Firstly, given that most intervals within music are small, the nearest note could be played. Whilst this method would likely require the least explicit statement of octave number it would be non-trivial to figure out the octave of a given note. The last defined octave in the path would need finding and then all subsequent notes walked through keeping track of the octave. The second consideration was to always use the last defined octave. Whilst this may require many octave definition around the boundary between octaves, it is easier to find what octave a note is played at as it is simply the last defined octave in the path. The second option was implemented.

1.3.9 Chords

To aid entering common chords, common types are repeated in a separate group at the top of the type drop-down. The layout of the chord drop-downs was improved with labels added making it clearer what the values refer to. If the notes were entered vertically, the order was reversed to have a greater correspondence with traditional staff notation.

1.3.10 Activation of turtles

A "Toggle Activation" button was added to the add-in window. When a cell or range is highlighted in the spreadsheet, the activation of any turtle definitions in this range will be toggled when the button is pressed. This significantly increases the ease with which turtles can be seleted as only two clicks are required as opposed to having to enter the cell edit mode and add or remove an exclamation mark.

1.4 Final Prototype Implementation

This section discusses the underlying implementation of the final prototype, following the participatory design. Excello consists of three main parts. The first, and largest, is the turtle system for playing the grid contents. The second is the method for inserting

the notes of chords into the grid. Thirdly, turtle input and modulation is made available through the custom Excel functions.

When the play button in the add-in window is pressed, the turtle definitions within the grid are identified. For each, the starting cell and movement instructions are used to establish the contents of the cells which it passes through. This is converted to a series of note definitions - pitch, start time, duration, volume. The speed and loop parameter are used to create the structure interpreted by the Tone.js library to schedule and initiate playback. An overview of the data flow and subtasks required to create the musical playback is show in figure 1.3.

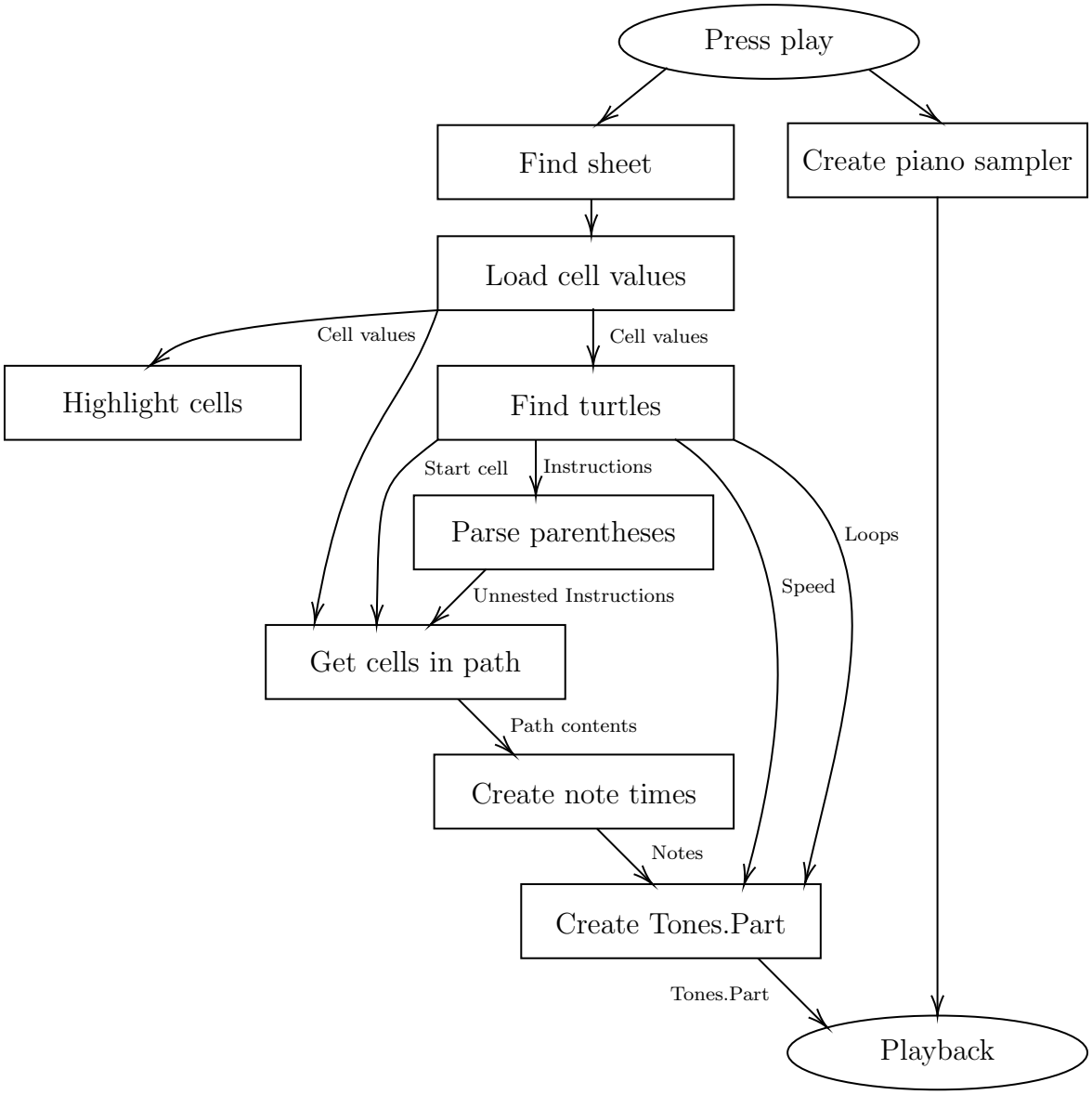

Figure 1.3: An overview of the playback algorithm and dataflow of Excello

An extension of the Tone instrument class is a Sampler. This interpolates between a set of pitched samples to create notes of arbitrary pitch and length. A sampler is loaded using the Salamander grand piano samples which includes four pitches (out of a possible 12) per octave. This accurately interpolates notes whilst reducing loading times and storage requirements.

1.4.1 Identifying Cells

Using the office API the sheet names are loaded and a drop-down menu for sheet selection is populated. When a user presses play, the cell values from the selected sheet are loaded. Then, the cell contents can be analysed for highlighting and calculating the musical output. Cells containing at least one definition of a note are highlighted red. A cell defining a note must contain a note name, an optional accidental, optional octave number, and optional volume instruction following a space in the form of a dynamic marking or number between 0 and 1. Notes are identified using the following regular expression:

```
'^[A-G](#|b|)?[1-9]?( (0(\.\[0-9]+)?|1(\.0)?|ppp|pp|p|mp|mf|f|ff|fff))?$'
```

Cells containing multiple definitions are split using commas. The resulting strings are trimmed of starting and ending whitespace and then must either be a note, a sustain ("`-`" or "`s`"), explicit rest ("`.`") or an empty string (created from trimming a rest). Cells matching "`-`","`.`" or "`s`" are highlighted a lighter red. Turtle definitions are tested using:

```
'^(!turtle\().*(\))$'
```

and cells containing turtle definitions are highlighted green. The same regex is used to identify definitions of turtles. The address of cells containing a turtle definition are added as text nodes to the live turtle section of the add-in window.

1.4.2 Parsing Movement Instructions

First the movement instructions are converted to a single unnested list of movements (e.g. "`(r m2)2"` becomes "`[r, m2, r, m2]`") so the turtle's path can be established. Initially the `parse` method of the Parenthesis[4] library seemed suitable for aiding in this string manipulation. This parses strings containing parenthesis into a nested array. For example, `parse('a(b[c{d}])')` gives `['a(', ['b[', ['c{', ['d'], '}'], ']'], ')']`.

This suggests that a string like "`(r m2)2`" would become `['(', ['r m2'], ')2']`. By removing the brackets from the strings within the array, a simple recursive method could be built to output `'r m2 r m2'` from `[['r m2'], '2']`. However upon testing this, an undefined array was outputted. From investigation of the source code I established that strings with a number following a closing parenthesis would all produce an error. Substituting characters for the numbers or placing a symbol before all numbers and then

[4]https://www.npmjs.com/package/parenthesis

later reverting this change would allow the library to be used. Instead, using the method employed by Parenthesis as inspiration, I implemented my own parsing function tailered for the needs of Excello.

This has two main steps. Firstly, the deepest bracketed expression is identified and stored in an array with the brackets removed. This expression is replaced in the original string with the string '___x___' where x is the index of this expression stored in the array. This is repeated until the original string contains no brackets. Secondly, a recursive function uses the indices placed between the '___' to reconstruct the string in the desired array format. This method is outlined in algorthitm 1. The Typescript implementation is in appendix A.1.

Having submitted a bug report on the Parenthesis Github, and implemented my own method for parsing the turtle movement instructions, I implemented a fix to the Parenthesis library. The existing function performed the initial replacement with the string '___x'. Therefore the x and following numbers would concatenate forming a single number, causing the library to fail. Utilising my method of having an identifier before and after the index number fixed the issue. I added this fix and additional tests to the Parenthesis to verify my method and ensured that previous tests all passed before submitting a pull request to the developers. This has since been merged into the library and published.

I wrote an additional recursive method to unnest the outputted array of this function into a single stream of instructions. An empty string, s, is initialised. For each item in the array, if it is an array, unnest the contents recursively. If not, it will be one or more single movement instructions. If the last item was an array the result of that array being unnested is added to s. If the first item in the single movement instructions is a number, the result of the array being unnested is added to s that number of times. The remaining instructions are added to s. This is outlined in algorithm 2. The implementation is shown in appendix A.1.

1.4.3 Getting Cells in Turtle's path

If the first argument in the turtle is defining a range of starting cells the cell addresses within this range are calculated. For each starting cell, the unnested instructions and sheet values are used to determine the contents of the cells the turtle passes through. Volume was also returned when dynamics were defined within the turtle. This is now handled in the next step. This process models the movement of the turtle within the grid. Keeping track of where it is positioned and which was it is facing. For each instruction the position and direction is updated as required and the contents of any new cells entered added to a list of notes.

Additional computation is required for the "`m*`" instruction, as the number of steps the turtle should take must be computed. Given the current position of the turtle and direction it is facing, an array of all the cells in front of it are taken from the sheet values.

Algorithm 1 Parsing bracketed expression. `str.replace(regex,f)` performs `f(s)` on the first substring, `s`, of `str` matching the regular expression `regex`.

```
1: procedure parseBrackets(str)
2:     idPadding ← '___'
3:     unnestedStr ← []
4:     deepestLevelBracketsRE ← RegExp('\\([^\\(\\)]*\\)')
5:     replacementIDRE ← RegExp('\\' + idPadding + '([0-9]+)' + idPadding)
6:
7:     procedure replaceDeepestBracket(x)
8:         unnestedStr.push(x.substring(1, x.length-1))
9:         return idPadding + (unnestedStr.length - 1) + idPadding
10:    end procedure
11:
12:    while deepestLevelBracketsRE.test(str) do
13:        str = str.replace(deepestLevelBracketsRE,replaceDeepestBracket)
14:    end while
15:    unnestedStr[0] = str
16:
17:    procedure reNest(outerStr)
18:        renestingStr ← []
19:        while There is a match of replacementIDRE in outerStr do
20:            matchIndex ← index of the match in outerStr
21:            matchID ← ID of the match (number between padding)
22:            matchString ← matched string
23:
24:            if matchIndex > 0 then
25:                renestingStr.push(outestStr.substring(0, matchIndex))
26:            end if
27:            renestingStr.push(reNest(unnestedStr[firstMatchID]))
28:            outestStr = outestStr.substring(matchIndex + matchString.length)
29:        end while
30:        renestingStr.push(outestStr)
31:        return renestingStr
32:    end procedure
33:
34:    return reNest(unnestedStr[0])
35: end procedure
```

Algorithm 2 Unnesting a parsed bracketed expression.

```
1: procedure processParsedBrackets(arr)
2:     s ← ''
3:     previousArr
4:     for v in s do
5:         if v is an array then
6:             previousArr ← processParsedBrackets(v)
7:         else
8:             if previous instruction was an array then
9:                 s ← s + previousArr
10:                if next instruction is a number then
11:                    s ← s + previousArr, that number of times minus one
12:                end if
13:            end if
14:            s ← s+ remaining instruction in v
15:        end if
16:    end for
17:    return s
18: end procedure
```

The turtle should step to the last cell that defines a note, sustain or explicitly defines a rest. The number of steps is the length of the array minus the index of the first element satisfying this criteria in the reversed array.

1.4.4 Creating Note Times

For each turtle, the cells moved through are calculated. This is used to create a data structure containing the information for playback to be initiated using the Tone library. For each turtle, the following array is produced: `[[<Note 1>,,<Note N>], <number of cells>]` (note sequence array). Each note is as follows: `[<start time>, [<pitch>, <duration>, <volume>]]`. Dynamics and the Octave are also added to each note if they had been omitted from a cell.

The Tone library has many different representations of time. I opted to use Transport Time for all time measurements - start times and durations. This is in the form `'BARS:QUARTERS:SIXTEENTHS'` where the numbers can be non-integer. The quarters value is used to represent number of cells so exact times, ticks or what musical note a length corresponds to (tricky for arbitrary subdivisions) need not be considered.

The note sequence array is initiated by counting the number of notes that are defined in the cell contents using the regular expressions for identifying notes and multi-note cells. The cells are iterated through keeping track of the active note and adding it to the note sequence when it ends. Outside of this loop, variables are defined to keep count of how many cells and notes through the process the algorithm is and whether the current value

is a rest or note. Variables keep track of the note currently being played - when it started (`currentStart`), the pitch (`currentNote`) and volume (`currentVolume`). As volume and octave number may be omitted, variables are also required to keep track of these.

Table 1.2 outlines the actions carried out when a cell is read. When a note is added to the note sequence, it is added in the form `[currentStart, [currentNote, "0:" + noteLength + ":0", currentVolume]]`.

Table 1.2: The actions taken when processing each cell to create note times. The beat count corresponds to the cell number being processed and is incremented each time.

Cell	State	Action		
Note	Note	Note, octave and volume established from cell contents and previous values	Previous note added to note sequence	currentStart = '0:'+beatCount+':0' currentNote = value noteLength = 1 currentVolume = volume
	Rest		inRest = false	
Sustain	Row	noteLength++		
	Rest	Nothing (has no semantic value)		
Rest	Note	Previous note added to note sequence inRest = true		
	Rest	Nothing		

The same method is used for multi-note cells, except the note length and cell count must be incremented by the appropriate fraction for each item in the cell. If at the end of the final cell, the state is in a note, this is ended and added to the note sequence.

The values in the note sequence are sufficient to play a note with the piano sampler using the `triggerAttackRelease` function. The `Tone.Part` class allows a set of calls to this method to be defined which can be started, stopped and looped as a single unit. Using the note sequence ("`noteTimes`"), number of cells ("`beatsLength`") from creating the note times, number of repeats ("`repeats`") and the evaluated speed argument ("`speedFactor`"), playback is scheduled with the following code (types have been ommitted for brevity):

```
var turtlePart = new Tone.Part(function(time, note){
  piano.triggerAttackRelease(note[0], note[1], time, note[2]);
}, noteTimes).start();
if (repeats>0){
  turtlePart = turtlePart.stop("0:" + (repeats*beatsLength/speedFactor) + ":0");
}
turtlePart.loop = true;
turtlePart.loopEnd = "0:" + beatsLength + ":0";
turtlePart.playbackRate = speedFactor;
```

1.4.5 Chord Input

When the insert button is pressed, the note, type, inversion[5] and octave of the chord are extracted from their HTML elements. The tonal library can then be used to generate the notes of the scale:

```
var chordNotes = Chord.notes(chordNote, chordType).map(x => Note.simplify(x));
```

The tonal simplify function reduces note definition involving multiple accidentals to contain at most one, as required by Excello. This provides a list of notes in ascending order without octaves or taking into account the inversion. In order to reach the correct inversion of the chord, the array of notes is rotated by the inversion number.

Octave numbers are added by iterating through the notes. A dictionary matches note names to position in the chromatic scale starting at C (the first note of the octave in SPN). This is also accounts for enharmonic notes[6]. The given octave number is appended to the first note in the chord. For each preceding note, if it appears in an equal or lower position in the scale than its predecessor, the octave number is incremented before appending. Otherwise, it is in the same octave so the octave number is appended without modification.

The range selected by the user is acquired with the Office API. The notes of the chord are entered starting at the top-left corner of this range. If the height of the range is greater or equal to its width, the notes are entered vertically going down from the starting cell. Otherwise they are entered horizontally going right. This is done by building the 2D array where the chord will be entered and setting that range using the Office API.

1.4.6 Custom Excel Functions

Custom functions are implemented using another add-in. As opposed to offering a separate window as the main Excello add-in does, this allows additional functions to be used in cells by using the prefix "=EXCELLO.". The file structure was generated with the Yeomann generator. The name, description, result type, and parameter names and types are store in a JSON schema. This is used by Excel to provide argument prompts and autofill for the user when editing the formula. Functions are given an identifier to link them to a typescript file where they are defined.

The turtle function concatenates the arguments into the correct format for Excello to recognise as a turtle. This allows other cells to be referenced, for example the speed variable can reference a global tempo variable as shown in figure 1.1.

[5] which note of the chord is the lowest, the chord ascends from this.

[6] Notes that are the same pitch but different names, such as Ab and G#

For every note defined in a cell, if there is a volume defined, the note is separated, modulated using the tonal `Distance.transpose` function and then combined back with the volume. This allows the drag fill feature of Excel to be employed by the user for transposing sections or to define melodic lines using the interval between notes as shown in figure 1.2.

1.5 MIDI Converter

The following section documents how the Python converter from MIDI to CSV suitable for Excello playback works. A MIDI file is divided into up to 16 parallel tracks [1]. Each track contains a series of messages defined using predefined status and data bytes. I used the Mido library[7] to read MIDI files and abstract away from the underlying byte representations and view the messages. Note onsets and offsets are two separate events with two separate messages [1]. A note onset or offset message includes the note pitch and velocity, channel (not relevant) and time in ticks since the last message [2]. The times for messages defining information not relevant for the conversion (e.g. piano pedalling, meta messages) must still be taken into account.

First, the list of messages is converted to a list of notes defined by onset and offset time, pitch and velocity. For each track, the messages are iterated through, using the time value in every message to update a variable tracking time. If the message defines a note onset, this is added to a dictionary mapping pitches to a list of currently active notes. Lists are used because a pitch can be active multiple times at once. For note offset messages, or onset messages with zero velocity, the note popped from the active notes at that pitch, its end time added, and then it is added to the list of all notes defined in the file.

As each turtle can only define one note at a time, the notes are split into lists so no list contains two notes which are playing at the same time. Provided the main list of notes is non-empty, a new list is created. The first remaining note is moved to the new list. The next remaining note starting after the previous note ends is moved to this new list. All remaining notes are iterated over. The number of iterations required is the number of turtles required, n.

If every tick corresponded to a cell, any combination of note onsets and offsets in a MIDI file could be accurately represented in Excello. To achieve smaller representations, the start and end times are converted to the cell number within the path of the turtle. For many MIDI files, the duration of a note, is different to the time it is notated to occupy. For example, a note immediately followed by another note in notation may have an end time significantly less than the start time of the next note in MIDI. A method is required to account for this. For all notes, before separation into the streams for different turtles, the length of the notes in ticks and differences between consecutive start times are found.

[7] https://mido.readthedocs.io/en/latest/index.html

The minimum or modal values for these times are calculated depending on the level of compression giving the $lengthStat$ and $differenceStat$.

$$ratio_{int} = \lfloor \max(lengthStat, differenceStat) / \min(lengthStat, differenceStat) \rfloor$$

For each note, the times are adjusted as follows:

$$length \leftarrow (start - end)/lengthStat \text{ (rounded to the nearest 0.1)}$$

$$start \leftarrow start/differenceStat \times ratio_{int} \text{ (rounded to the nearest 0.1)}$$

$$end \leftarrow start + length$$

Next the streams, with note start and end times corresponding to cells, are converted to a CSV file to be run with Excello. The path for each turtle is initialised as an array of empty strings. The length of these arrays is the maximum end time for a note in any turtle, L. Each note the turtle plays is entered into the array. MIDI defines pitch using the integers. I used the library audiolazy[8] to convert MIDI number to SPN. If the note velocity is different to the previous note played by the turtle (or the note is the first note played), the eight-bit velocity as defined by MIDI is mapped to the range [0,1] as used by Excello. If the note length is greater than one, sustains are placed in the following cells. These paths go right starting in column A, with the first in row 2.

Finally the definition of the turtle must be placed in the spreadsheet. The start cell range is 'A2:A($n + 1$)'. The movement instruction is "r mL". The MIDI file contains meta data for the `tempo` (milliseconds per beat) and `ticks_per_beat`. Cells per minute is calculated as follows:

$$cells\ per\ tick \times ticks\ per\ beat \times beat\ per\ minute$$

$$= \frac{ratio_{int}}{mode\ difference} \times \texttt{ticks\_per\_beat} \times \frac{60 \times 10^6}{\texttt{tempo}}$$

Using one as the number of repeats, the turtle definition is placed in cell A1 and the CSV exported.

1.6 Repository Overview

TODO

[8] https://pythonhosted.org/audiolazy/

Bibliography

[1] MIDI Manufacturers Association. *The complete MIDI 1.0 detailed specification*, 1997.

[2] Ole Martin Bjrndalen. Midi files. `https://mido.readthedocs.io/en/latest/midi_files.html`, 2016. Accessed: 2019-04-18).

[3] Chris Nash. Manhattan: End-user programming for music. In *NIME*, 2014.

[4] Erik Sandberg, Examensarbete Nv, Reviewer Arne Andersson, and Examiner Anders Jansson. Separating input language and formatter in gnu lilypond, 2006.

Appendix A

Excello Implementation

A.1 bracketsParse.ts

```
// inspiration taken from:
// https://github.com/dy/parenthesis/blob/master/index.js

/**
 * Given a turtle instruction sequence this unwraps any brackets to create exact instrutions
 * @param str Turtle movement instructions e.g. "(r m3)4"
 * @return explicit unwrapped instructions e.g. "r m3 r m3 r m3 r m3"
 */
export function parseBrackets(str: string) {

var unnestedStr = ['will become highest level'];
var idPadding = '__';

var deepestLevelBracketsRE = new RegExp('\\([^\\(\\)]*\\)'); // finds bracket with no brackets inside

// store contents of bracket it unnestedStr and replace contents in str with ID
while (deepestLevelBracketsRE.test(str)) {
str = str.replace(deepestLevelBracketsRE,function(x) {
unnestedStr.push(x.substring(1, x.length-1)); // add the token without the brackets
return idPadding + (unnestedStr.length - 1) + idPadding;
});
}
unnestedStr[0] = str; // make first element in array the highest level of the string

var replacementIDRE = new RegExp('\\' + idPadding + '([0-9]+)' + idPadding);

// transform references to tree
function reNest (outestStr: string) {
var renestingStr = [];
var match;

while (match = replacementIDRE.exec(outestStr)) {

var matchIndex = match.index;
var firstMatchID = match[1];
var fullStringMatched = match[0];

// push what was before
if (matchIndex > 0) {
renestingStr.push(outestStr.substring(0, matchIndex))
}
//perform recursively
renestingStr.push(reNest(unnestedStr[firstMatchID]))
// remove the string that has been processed
outestStr = outestStr.substring(matchIndex + fullStringMatched.length)
```

```
}
renestingStr.push(outestStr)
return renestingStr
}

return reNest(unnestedStr[0])
}

export function processParsedBrackets(arr) {
var s = "";
var wasPrevArray = false;
var prevArray = "";
for (let val of arr) {
if (val.constructor === Array) {
prevArray = processParsedBrackets(val)
wasPrevArray = true;
}
else {
var singleInstructions = val.trim().split(" ");
if (wasPrevArray) {
s = s + prevArray;
if (!isNaN(singleInstructions[0])) {
for (var i=1; i<singleInstructions[0]; i++) {
s = s + prevArray;
}
singleInstructions = singleInstructions.slice(1);
}
}
for (let instruction of singleInstructions) {
s = s + instruction + " ";
}
wasPrevArray = false;
}
}
if (wasPrevArray) {
s = s + prevArray;
}
return s;
}
```