CPSC 217 Assignment 4

Due: Friday, December 4, 2026, at 11:00pm
Weight: 7.25%
Sample Solution Length: 212 lines (including good comments, but not the bonus part)

Representing music is a complex task because pitch, rhythm, tempo, dynamics, articulation and phrasing (among other factors) must be represented over time. Modern sheet music captures this information in a manner that allows skilled players to play a song in the manner intended by the composer, though each artist typically brings their own style to the piece when it is performed.

While sheet music is appropriate for people reading music, it is not straightforward for a computer to process because of the complexity of the overlapping symbols. As a result, other formats have been developed for representing music that are more amenable to being processed by a computer. One such format is the completely text-based ABC notation. While everything you need to know about ABC notation for this assignment is described in this document, you can also read an overview of the format on Wikipedia (https://en.wikipedia.org/wiki/ABC_notation) if you are interested, and the full specification for the format can be found here: https://abcnotation.com/wiki/abc:standard.

Because ABC notation allows a wide variety of music to be represented, it is quite complex. As a result, we will only consider a limited subset of ABC notation in this assignment, but it is still possible to represent numerous pieces of music with this subset.

In this assignment you will write a program that reads a file in ABC notation and displays the equivalent sheet music for it. You will also extend your program so that it either plays the piece of music in question or highlights each note that should be played in sequence with appropriate timing and colors that help guide a novice player. This document begins with an overview of the subset of ABC notation that will be used in this assignment. That is followed by steps that guide you through rendering an ABC notation file so that the music is displayed in the conventional form.

ABC Notation

Each input file in this assignment will consist of two major parts: A header that contains information fields for the tune, followed by a body that contains the notes that represent the tune itself.

The first line in the header is always `%abc` (followed by a newline character). This makes it easy for software to quickly check if the file that has been opened is in the expected format. The second line in the header is always `X:1` (again, followed by a newline character). When implementing a complete parser for ABC notation files, the X information field allows multiple tunes to be stored in one file (but this isn't something that will be considered in this assignment). Your program will need to read the `%abc` and `X:1` lines, but it won't actually use the values – you just need to move past them to reach the remaining lines in the file. These first two lines will be followed by five information fields, the first four of which will always be:

T: The title of the song.

C: The composer of the song.

L: The duration of a note with length one (subsequently referred to as the unit-length note). Supported note lengths are 1/2, 1/4, and 1/8.

Q: The tempo of the tune in unit-length notes per minute.

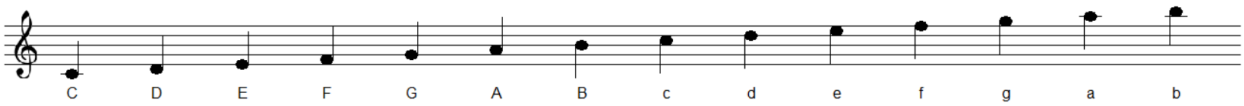
You should assume that all of these fields will always be present in this order as this will make reading the file's header relatively straightforward. (A complete parser for ABC notation files must allow the order of some of these fields to differ and assign default values to fields that are missing, but supporting such would add unnecessary complexity to this assignment).

Finally, the header ends with the required K information field which indicates the key in which the tune is written. We will only consider tunes in the key of C for this assignment.

An example header follows:

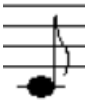
```
%abc
X:1
T:Twinkle, Twinkle, Little Star
C:Traditional
L:1/4
Q:100
K:C
```

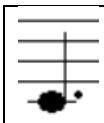
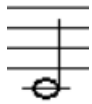
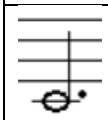
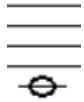
The tune's header is followed by a body which describes the pitches and durations of its notes. Each line in the body represents all of the notes and bar lines that will appear on one staff of the rendered music. Pitches are denoted by the following strings: "C", "D", "E", "F", "G", "A", "B", "c", "d", "e", "f", "g", "a" and "b", as illustrated below:



A pitch must be provided for every note that appears in a tune stored in an ABC notation file.

While the notes above all have the same duration, most music includes notes with different durations. The durations that your program must support are shown below:

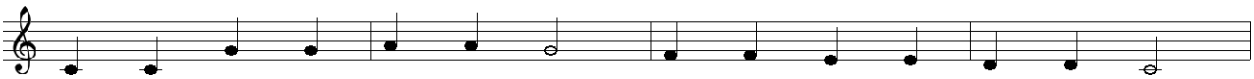
Image	Duration	Name
	1/8 = 0.125	Eighth note
	1/4 = 0.25	Quarter note

	3/8 = 0.375	Dotted quarter note
	1/2 = 0.5	Half note
	3/4 = 0.75	Dotted half note
	1 = 1.0	Whole note

When only a pitch is provided, the note has the unit-length indicated in the tune's header. For example, "c" and "G" are both strings that represent a note with unit length. Non-unit durations are specified as a multiplier immediately following the pitch. For example, while "C" represents a C pitch with unit length, "C2" represents a C pitch that is twice as long. The | character is used to separate bars. Each note (pitch together with optional duration) is separated from all other notes by a space. Similarly, there will also be a space between any note and an adjacent bar line. For example, the first line of Twinkle, Twinkle, Little Star is denoted by:

```
C C G G | A A G2 | F F E E | D D C2 |
```

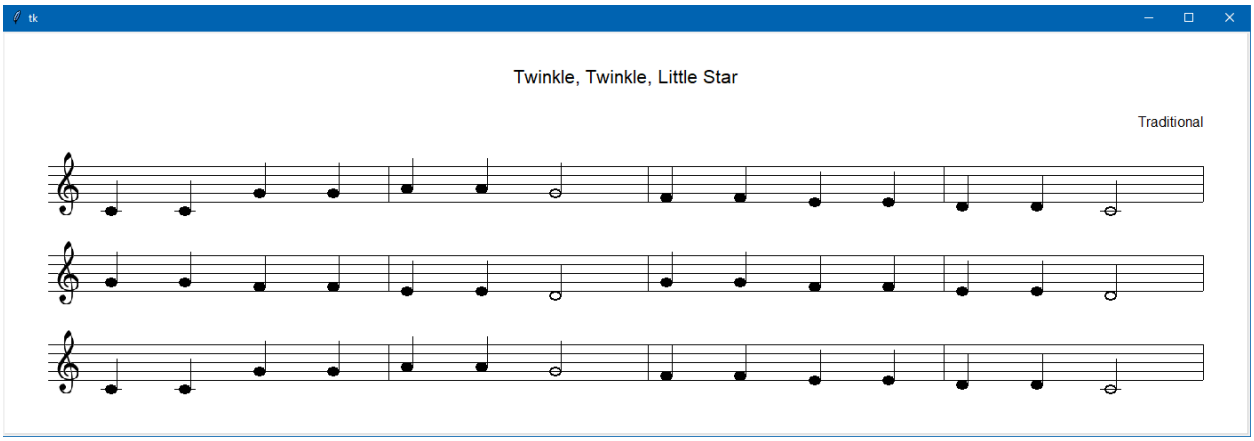
It is rendered as:



Notice that all of the unit-length notes are quarter notes because the L information field in the header had the value 1/4, and the notes with a duration of 2 are half notes because 2 * 1/4 = 0.5 = 1/2.

A complete ABC file and rendering of Twinkle, Twinkle Little Star is shown below:

```
%abc
X:1
T:Twinkle, Twinkle, Little Star
C:Traditional
L:1/4
Q:100
K:C
C C G G | A A G2 | F F E E | D D C2 |
G G F F | E E D2 | G G F F | E E D2 |
C C G G | A A G2 | F F E E | D D C2 |
```



Notice that the title is centered above the music, the composer is right justified and appears vertically between the title and the music, and one line of music is rendered for each line in the body of the ABC file. A treble clef appears at the beginning of each line. An image of this clef is available on the course website.

Part 1: Processing the Header

Begin by creating a program with a main function that reads the name of an ABC file from the user, opens the file, reads and processes the header information, and display a selection of values. Your program should report an appropriate error message and quit if the user enters the name of a file that doesn't exist. If the file is opened successfully, your program may assume that all of the values in it are present and correct. You don't need to consider cases such as a missing information field or an improperly formatted value.

Hint: You may find it helpful to use the strip method to remove the newline character from each line read from the file before processing it.

Once you have read all of the header values from the file, you should use the `print` function to display them so that you can verify that you have read and processed them correctly. (There is no graphical output for this part of the assignment). The output from my program for Twinkle, Twinkle, Little Star is shown below:

```
Title: Twinkle, Twinkle, Little Star
Composer: Traditional
Unit note length: 0.25
Tempo: 100
```

Notice that while the unit note length was stored in the file as the string 1/4, my program reports the unit note length as the floating-point number 0.25. As you progress through the assignment, you will find that it is necessary evaluate fractions. I have provided a function named `evaluateFraction` that will assist you with such. This function takes a string representing a fraction such as "1/4", "1/2", "3/8", etc. and returns the equivalent floating-point number. It will also accept strings that contain an integer such as "2", "3" and "4" and return the equivalent floating-point number. Use the `evaluateFraction` function to convert the unit note length to a floating-point number before displaying it.

Several other tunes are available on the course website along with their expected output. I strongly encourage you to verify that your program displays correct values for the tunes as well before moving on to the second part of this assignment.

Part 2: Processing the Body

The body of the file consists of one or more lines that describe the pitches and durations of the notes that make up the tune. Read all of these lines from the file and store them in a list. Once you have done that, display the number of lines in the song with an appropriate label.

As mentioned previously, each note in the tune consists of a pitch followed by an optional duration. This optional duration is a multiplier applied to the unit note length read from the tune's header. The multiplier can be greater than one, resulting in a longer note, or it can be less than 1, resulting in a shorter note. For example, when the unit note length is 1/4, "C" (which does not include an optional duration) represents a quarter note, while "C2" represents a half note (because 1/4 * 2 = 1/2) and "C4" represents a whole note (because 1/4 * 4 = 1). Similarly, "C1/2" represents an eighth note (because 1/4 * 1/2 = 1/8). Because the multiplier may be a fraction, you'll want to use the `evaluateFraction` function described previously to assist you with determining the duration of each note.

Use the ideas in the previous paragraph to write a function that splits a note into its pitch and its duration. The function will take a string representing a note (pitch, optionally followed by a duration) as its first parameter. It will take the length of the unit length note as its second parameter. The function will return the pitch (a one-character string containing a letter between A and G, either upper or lower case) as its first result, and it will return the note's duration as a floating-point number as its second result.

Use your function for splitting notes to write a function that prints out the pitch and duration for every note in the song, with each pitch and duration appearing on its own line. Your function will take two parameters which are the list that represents the tune and the duration of a unit-length note. For example, the first line of Twinkle, Twinkle, Little Star has the following sequence of pitches and durations:

```
C 0.25
C 0.25
G 0.25
G 0.25
A 0.25
A 0.25
G 0.5
F 0.25
F 0.25
E 0.25
E 0.25
D 0.25
D 0.25
C 0.5
```

Extend this function so that it identifies the length of the shortest note in the song (in addition to displaying all of the pitches and durations). The length of the shortest note will be returned as the function's only result. Once you have done this, add two lines of output to your main program: One that displays the number of lines in the song, and a second that reports the length of the shortest note as a floating-point value.

I strongly recommend that you verify that your complete list of pitches and durations matches the results shown on the course website for all of the tunes. Pay particular attention to the unit note lengths and shortest note durations when you are testing. Once you have verified that all of the durations match, you can turn the print statement in your shortest note function into a comment so that your program's output is a more manageable length. When this change is performed, your program should print the following output for Twinkle, Twinkle, Little Star:

```
Title: Twinkle, Twinkle, Little Star
Composer: Traditional
Unit note length: 0.25
Tempo: 100
Number of lines: 3
Shortest note: 0.25
```

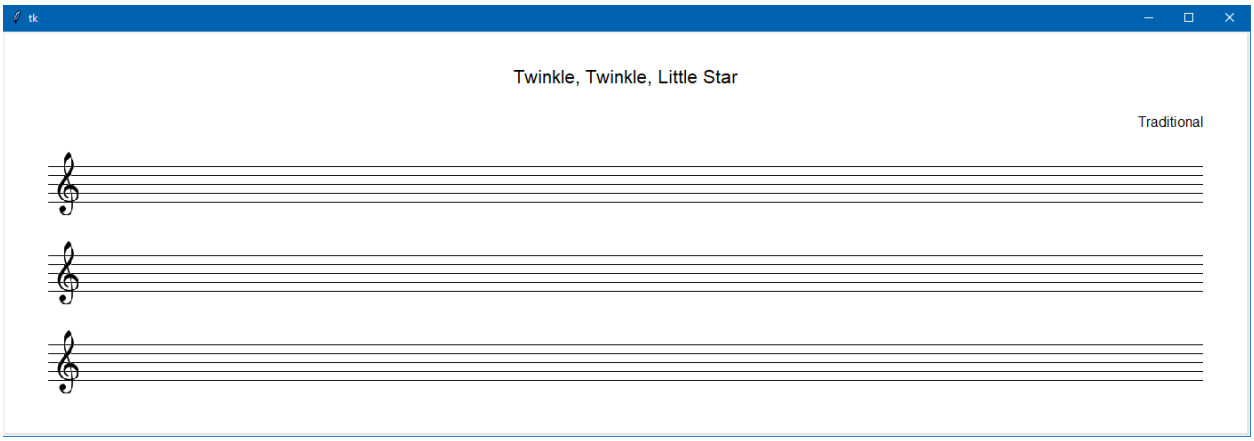
Part 3: Displaying the Tune

In the previous parts of the assignment, you have written functions and computed values that will help you to display the music for a tune. The default SimpleGraphics window size is rather small for displaying sheet music. As a result, you'll want to resize the window to a larger size. Furthermore, since you'll be resizing the window, you can resize it to the correct height to comfortably hold all of the lines for the tune. I chose a width of 1400 pixels, but my program is designed so that it scales nicely for windows that are narrower or wider. Your program should similarly handle the width of the window being changed (within reasonable values). The height of the window should be 150 pixels, plus 100 pixels for every line in the song.

Draw the title of the tune near the top of the window, centered horizontally, ideally in a larger font. Draw the composer below the title and right justify it so that the right edge of the composer is 50 pixels from the right edge of the window.

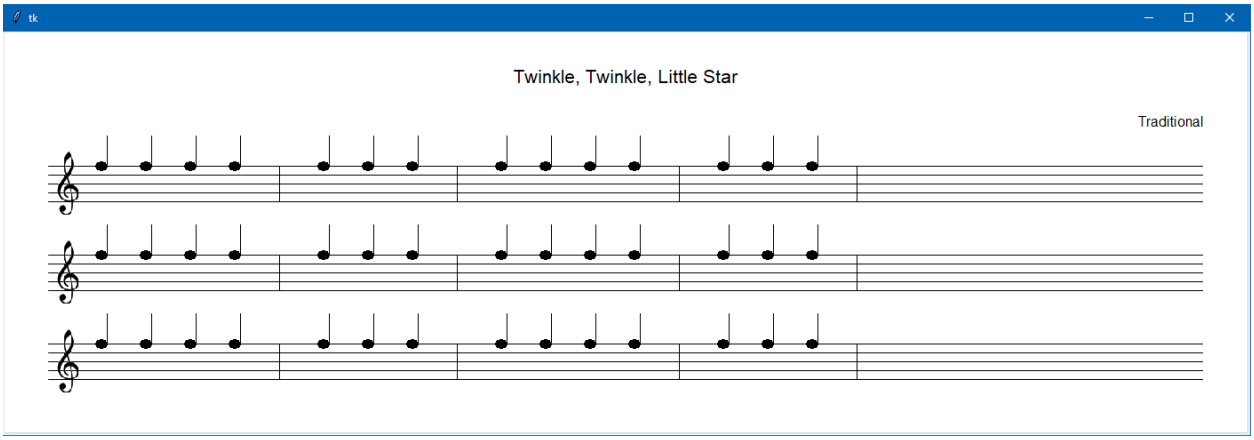
A function named `drawStaff` is available in the starter code on the course website. It draws a staff consisting of 5 horizontal lines and a treble clef. That function will draw the staff so that it fills the width of the window nicely, leaving a 50-pixel margin at its left and right edges, but you must specify the vertical location at which it is drawn by providing it as the only parameter to `drawStaff`. This parameter is the y position of the top line of the staff. The first staff that you draw should have y position 150. Each subsequent line of the tune should be 100 pixels lower down in the window.

Once you update your program to resize the window, draw the title and composer, and one staff for each line in the tune, your window should look something like this for Twinkle, Twinkle, Little Star:

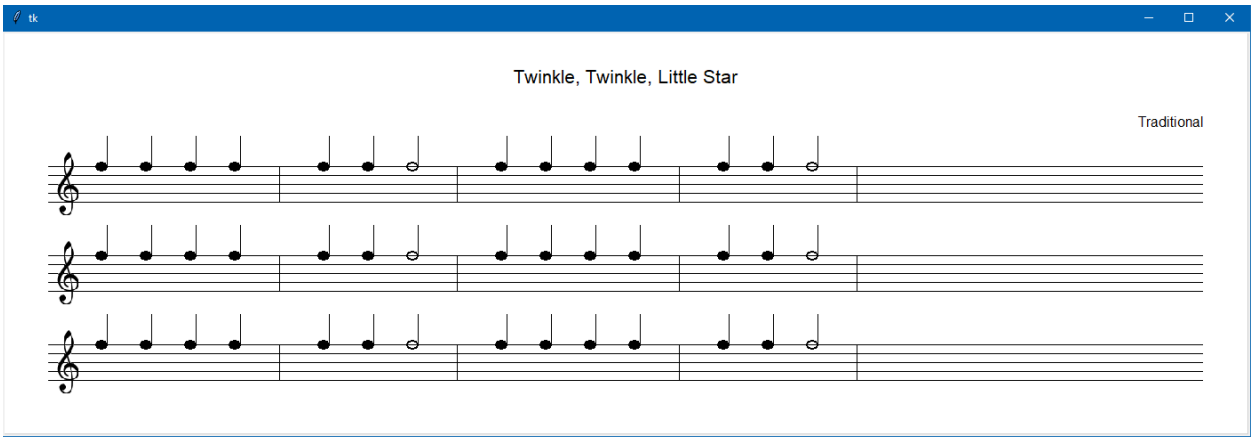


Once the staves have been drawn, you'll need to add notes and bar lines to them. A bar line is straightforward to draw; it's just a vertical line that extends from the top line of the staff to the bottom line.

A function named drawNote has been provided for you that is able to draw eighth notes, quarter notes, dotted quarter notes, half notes, dotted half notes and whole notes. Its first two parameters specify the location of the center of the head of the note, and its third parameter is the duration of the note as a floating-point number (one of 0.125, 0.25, 0.375, 0.5, 0.75, or 1.0). To get started, I'd suggest that you ignore both the note's duration and its pitch and concentrate only on drawing the bar lines and the correct number of notes between them. For example, if you start x at 110 and increase the x position by 50 pixels for each subsequent note or bar line and keep y constant, you'll end up with output that looks like this for Twinkle, Twinkle, Little Star if you draw all the notes as quarter notes on the top line of the staff:

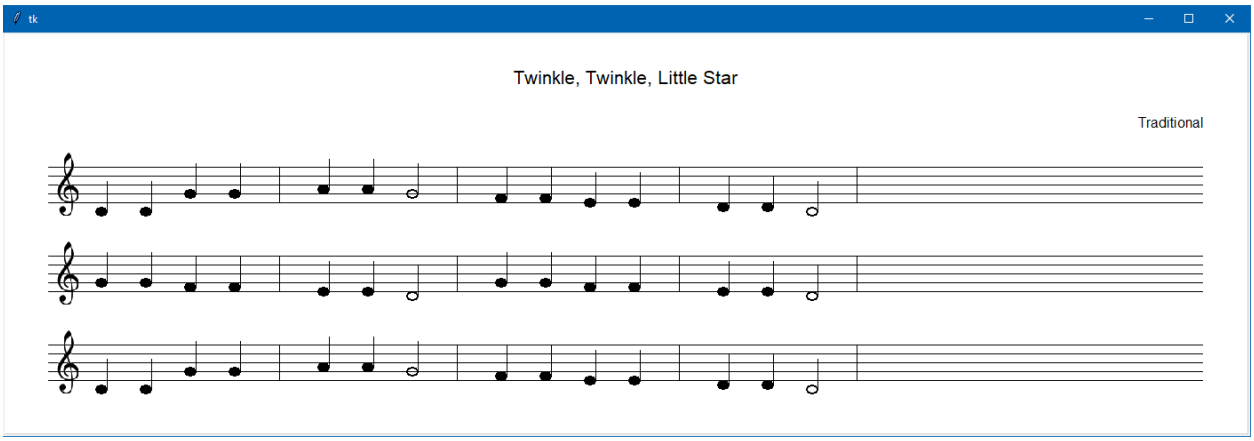


Of course, music that only uses quarter notes is relatively rare, so you are going to need to add support for the durations to your program. Begin by determining whether or not the note consists of only a pitch (it is a single character) or a pitch and a duration (it is more than one character). If the note is only a pitch, then its duration is the unit note length read from the tune's header. Otherwise, it's the unit note length read from the tune's header multiplied by the duration part of the current note. The duration part of the current note may be a fraction, so you'll want to use the fraction evaluation function described previously (that has been provided on the course website). Once the notes' durations are considered, your output for Twinkle, Twinkle, Little Star should look like this:



While the image above accurately captures the durations of the notes that need to be played for the tune, all of the notes are currently being displayed with the same pitch. To accurately represent the tune, the vertical positions of the notes will need to be changed so that each note's head is on the correct line or in the correct space for its pitch.

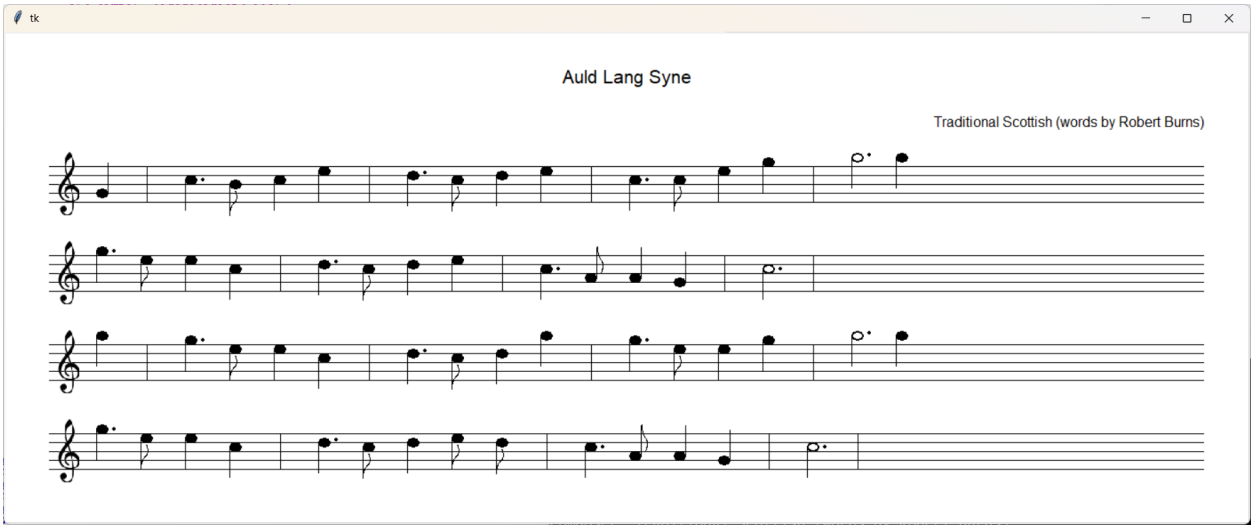
Build a dictionary that maps each of the 14 pitches ("C", "D", "E", "F", "G", "A", "B", "c", "d", "e", "f", "g", "a", and "b") to the vertical offset needed for that pitch from the top line of the staff. (This means, for example, that the offset for "f" will be 0 because it is supposed to reside on the top line). Use the values in this dictionary to correctly position each note without using if statements to differentiate between all of the cases. Using the dictionary to adjust the vertical positions of your notes should result in output similar to what is shown below:



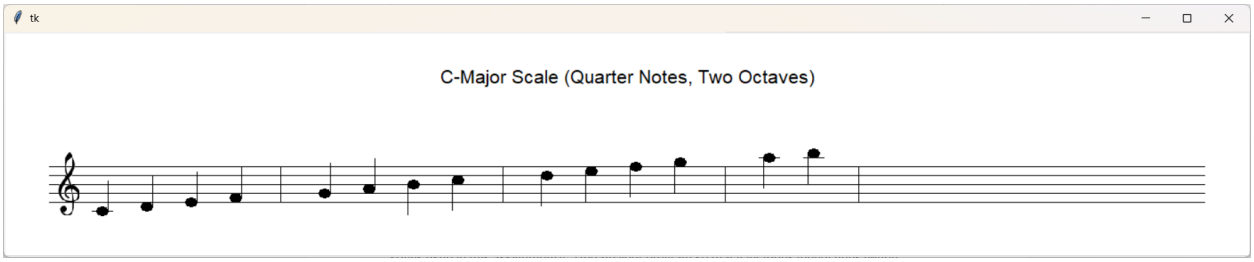
The drawNote function includes three optional parameters that follow the required parameters. These control the position of the dot for dotted quarter and dotted half notes, whether the stem points up or down, and the color of the note. If the pitch of the note is "D", "F", "A", "c", "e", "g", or "b", then the note resides in a space between the staff lines and the dot should be centered vertically on the note head by passing True to drawNote's fourth parameter (or not providing a value for it because its default value is True). Otherwise, the note's head is on a line, and the dot needs to be shifted up to prevent it from overlapping with the line. False must be passed for this parameter in this situation.

Right now, all of the notes that you draw have their stems pointing upwards. While this makes sense for Twinkle, Twinkle, Little Star because it only includes pitches on the lower half of the staff, it doesn't work well for tunes with higher notes. For the purposes of this assignment, notes that sit below the middle line ("C", "D", "E", "F", "G", and "A") must have up stems, while notes that sit on or above the middle line must have down stems. The direction of the stem is controlled by drawNote's fifth parameter.

Twinkle, Twinkle, Little Star doesn't include any dotted notes, nor notes that use a down stem. As a result, you should move to testing your program with a different tune. For example, Auld Lang Syne will allow you to test your most recently added functionality.



Most of the notes in the previous piece of music lie within or immediately adjacent to the staff, but the last two notes on the first line are above it (and not touching the top line), as are three notes on the third line. It is conventional to add ledger lines for such notes to make it clear how far above the staff the note head is located. Similar ledger lines are needed for notes below the bottom of the staff. (Ledger lines are important for the notes that appear in the preceding piece, and are critical for notes that are further below or above the staff (though such notes won't appear in any of the songs used in this assignment)). Update your program so that it includes ledger lines (when necessary) that are 25 pixels wide and centered horizontally on the note's head. One ledger line is drawn 10 pixels below the bottom line on the staff for "C", and one ledger line is drawn 10 pixels above the top of the staff for "a" and "b". When the ledger lines are included, the graphical output for the `scale.abc` should look like this:



Sheet music is normally rendered so that longer notes take up more space. One might be inclined to think that the amount of space should be directly proportional to the length of the note, but sheet music is rarely rendered in this manner because whole notes end up taking up too much space. Instead, a commonly used approach is to increase the space by 1.5 times each time the duration of the note doubles. This provides more space for longer notes without excessive space around longer notes.

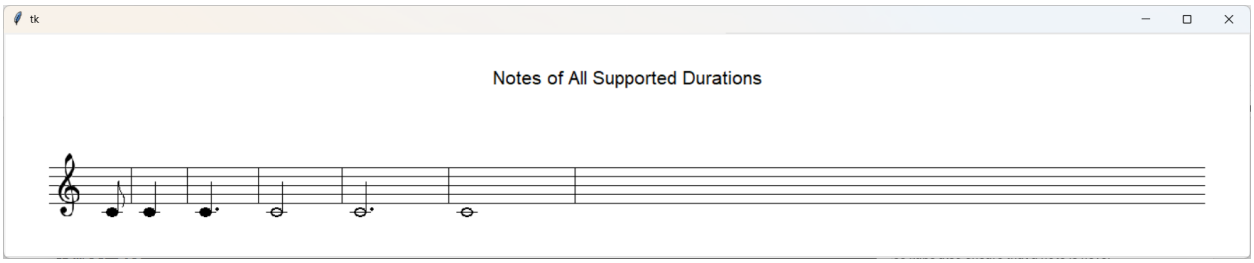
Each note head is 14 pixels wide. To avoid having any notes too close together, we will always reserve 14 pixels to the left of each note head and a minimum of 14 pixels to the right of each note head (with more than that provided for longer notes). These gaps also ensure that a note is never too close to a bar line that precedes or follows it.

You previously wrote a function that identified the length of the shortest note in a song. Using this function, update your program so that the space allocated to each note is 14 + 14 + 14 = 42 pixels multiplied by 1.5 raised to the base 2 logarithm of the length of the note divided by shortest note length, as shown below:

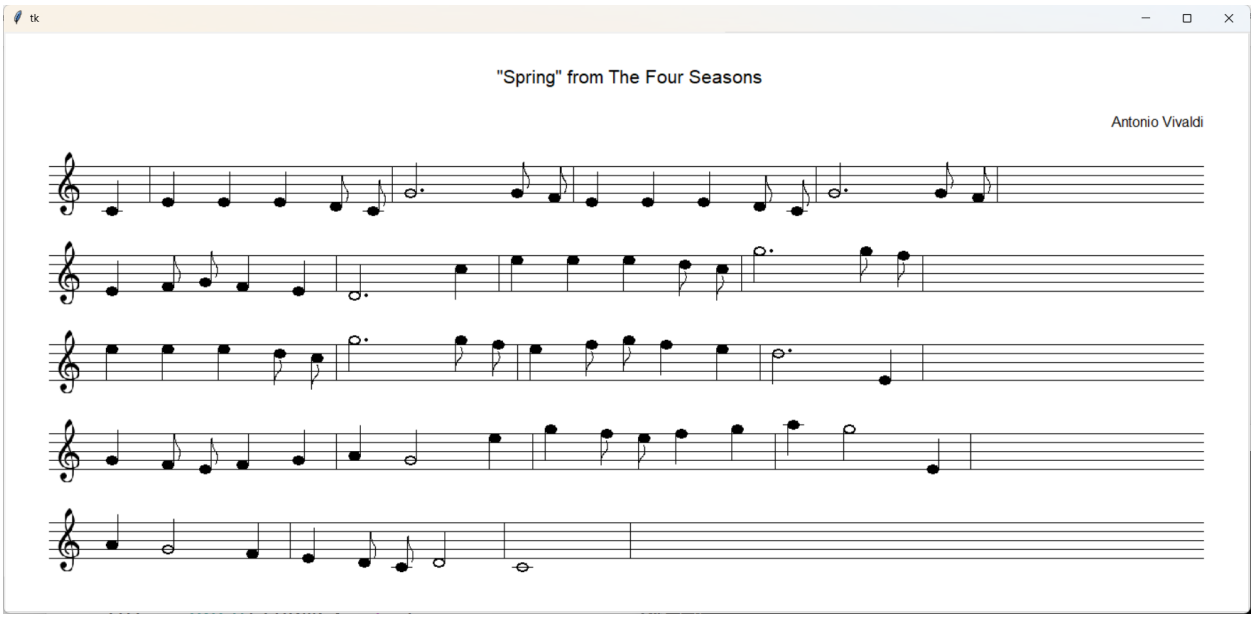
x_skip = 42 * 1.5 ** log2(note_length / shortest_note)

Note that you'll want to ensure that the note head has 14 pixels reserved to its left so that it's not right up against the bar lines. This means that you'll probably want your note's head drawn at (x + 21, y) if you were previously drawing it at (x, y). You will also need to update the x-coordinates for your ledger lines. Finally, you might want to start your x value at 100 instead of 110 because the changes made here ensure that there is space to the left of the first note on the line that wasn't there previously.

Once you make these updates, your output should look like this for durations.abc:

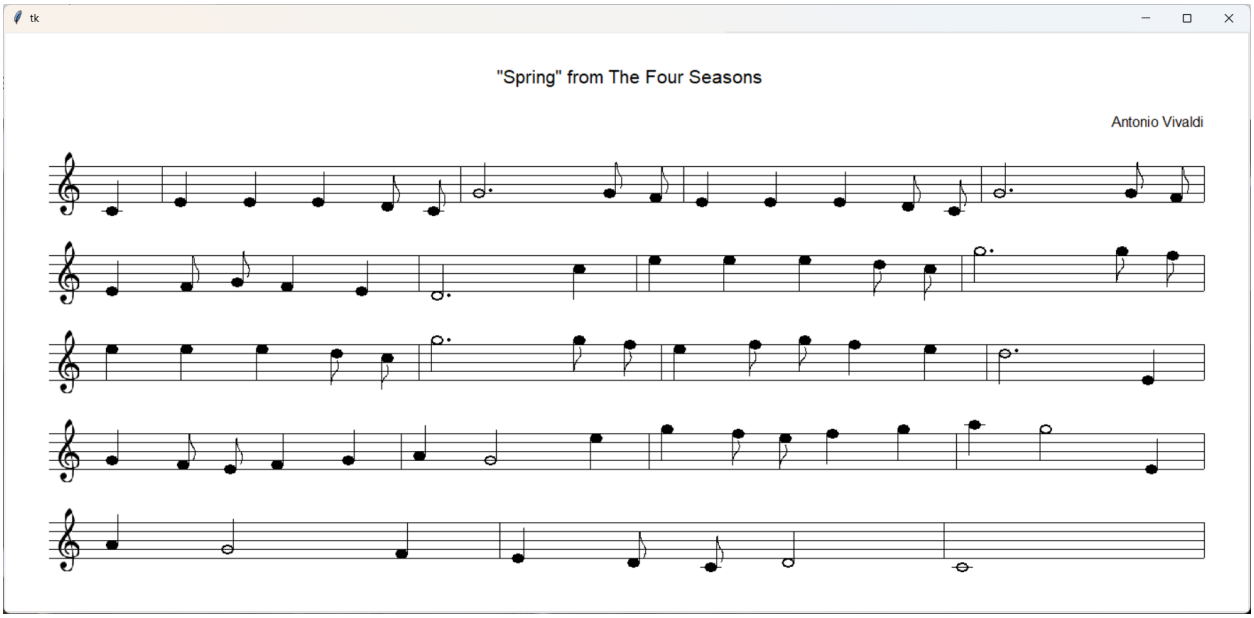


Similarly, the output for Spring is shown below:



Music is normally rendered so that each line is filled completely by adding additional space between each of the notes much like a word processor increases the amount of space between words when full justification is enabled. To accomplish this, compute the number of pixels required to render the line in its current form (without actually drawing it). Then compute a stretch factor, which is the width of the window minus 150 pixels[1], divided by the number of pixels used to display the line in its current form. Finally, render the line as you did previously, but multiply the x_skip value by the stretch factor before adding it to your current x position.

Once you make this final update, your output should look like this:



[1] The left and right margins are each 50 pixels, and an additional 50 pixels is needed to leave sufficient space around the treble clef.

Part 4: Playing the Tune

In this part of the assignment, you can choose to perform one of these two tasks. Either:

- Play all of the notes in sequence so that you can hear the tune; or
- Highlight each note in sequence (with correct timing) showing the notes that are to be played.

You are also welcome to complete both tasks, though there is no additional grade bonus for doing so. I encourage you to choose the first option (and it's no more difficult than the second) because I think it is more engaging. Select the second option if you struggle to get pygame or pygame-ce installed, or if you are working on a device that doesn't have any speakers (or if the sound will disturb people around you).

Part 4a: Playing the Notes

In this part of the assignment, you'll extend your program so that you can hear the song play. Note that you must render and display the complete song first and then play it. It is **not** acceptable to merely play the notes as they are displayed.

Python does not include a module for playing sound as part of its default installation. As a result, you'll need to install pygame or pygame-ce. In PyCharm, you do this by opening the terminal tab at the bottom of the screen and entering the command "pip install pygame" or "pip install pygame-ce" (without the double quotes). For other environments, you can open a command prompt or terminal window and enter one of the two commands noted previously.

Once pygame or pygame-ce is installed, you need to import it by adding the following statement near the top of your program:

```
import pygame
```

Then, in your main program or a function that plays the song, you need to initialize pygame and its audio mixer with the statements:

```
pygame.init()
pygame.mixer.init()
```

Wave files are available on the course website for notes C4 through B5 (with files C4 through B4 being the files for pitches "C" through "B", and files C5 through B5 being the files for pitches "c" through "b"). If these are stored in the same folder as your .py file, you can load them by passing the name of the sound file as the only parameter to pygame.mixer.Sound. A sound object will be returned that you'll want to store in a dictionary where the keys are pitches (strings like "c" and "A") and the values are the sound object that corresponds to each pitch. For example, if you have previously initialized `sounds` to the empty dictionary, you can load the sound for "C" with the statement:

```
sounds["C"] = pygame.mixer.Sound("Piano.mf.C4.wav")
```

You'll want 13 additional lines in your program to load the other 13 wave files.

To play a sound, you'll need to start it playing, then sleep for the duration of the note, and then stop the sound from playing before moving onto the next note. The tempo value that was read from the tune's header also needs to be considered when computing the delay. Use the following formula:

delay = note_duration / unit_length * 60 / tempo

This computes the delay needed in seconds (and will often be only part of a second). The 60 in the formula is needed because the tempo is specified as unit-length notes per minute, not unit-length notes per second. Then this delay value can be passed to the time module's sleep function (don't forget to import the time module). Finally, you'll need to stop the note from playing. For example, to play the file for "C", you'd use the following statements:

```
sounds["C"].play()
delay = note_duration / unit_length * 60 / tempo
time.sleep(delay)
sounds["C"].stop()
```

Of course, you'll want to use the pitch for the current note in the song instead of always playing Cs, (and you'll similarly need to ensure that you use the duration of the current note). This can be accomplished by using the current note's pitch as the key for the sounds dictionary instead of always using "C".

Part 4b: Highlighting Each Note in Sequence

Some children (and, perhaps, some adults) learn to play the piano with the assistance of color coding added to the notes. For example, C's might be highlighted in red, D's in orange, E's in yellow, etc. In this part of the assignment, you'll highlight each note in sequence (with the correct timing for the tune) using a unique color for each note.

Begin by building a dictionary that maps each pitch to a color, as described in the table below:

Pitch	SimpleGraphics Color
"C" or "c"	red
"D" or "d"	darkorange1
"E" or "e"	gold
"F" or "f"	green3
"G" or "g"	cyan
"A" or "a"	purple
"B" or "b"	hotpink1

The final parameter to drawNote controls the color of note (and has a default value of "black"). Once the whole song has been rendered in black, you'll need to draw each note again in sequence with the correct color, delay for the appropriate length of time based on the note's length, and then draw the note again in black to cover up the colorful version of it. The length of the delay will be computed from the note's duration and the tempo value that was read from the tune's header. Use the following formula:

delay = note_duration / unit_length * 60 / tempo

This computes the delay needed in seconds (and will often be only part of a second). The 60 in the formula is needed because the tempo is specified as unit-length notes per minute, not unit-length notes per second. Then this delay value can be passed to the sleep function which is in the time module (don't forget to import the time module before calling sleep). The following algorithm summarizes what you need to accomplish:

After the music for the tune has been displayed, for each note in the tune:
    Identify the color for its pitch
    Draw the note in the correct location with that color (overwriting the black version of the note that was already present)
    Sleep for an appropriate amount of time
    Draw the note again in black to cover up the colorful version of it

Note that you must display the colorful notes in sequence after the complete tune has been displayed. It is **not** acceptable to merely show the notes in color as the tune is first displayed.

Additional Requirements

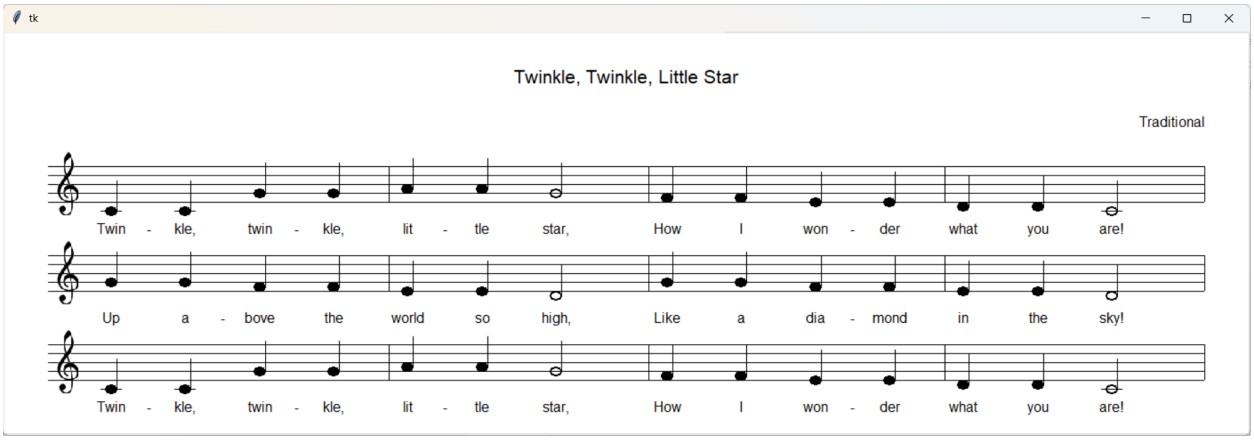
- Your program must verify that the file specified by the user exists. If an error is encountered while opening the file, then your program should display an appropriate error message and quit. Close the graphics window when the program quits by calling the `close()` function.
- You must create a dictionary that maps pitches to offsets from the top line of the staff. It is **not** acceptable to use if or if/elif statements to compute the offsets required for all of the pitches.
- Your program should only read the file once. Do **not** read the file a second time to complete Part 4 of the assignment.
- Close every file that you open.
- Your program must make appropriate use of functions. In particular, it must include a function that splits a note into its pitch and duration (as described in Part 2) and a function that finds the duration of the tune's shortest note (also described in Part 2).
- You may write additional functions if doing so helps you separate the problem into logical steps, reduces the amount of repeated code or otherwise results in a better solution to the problem.
- The only lines of code in your program that should be outside of a function definition are constants, import statements, and the call to the main function (which is normally the last line in the file).
- Do **not** define one function inside another function.
- Your program must **not** use global variables (except for constant values that are never changed). Constant values that you might want to include in your program include the minimum number of pixels allocated to a note (which is 42), and the dictionary that maps pitches to offset needed to correctly position a note with that pitch on the staff.
- You may assume that the data in the files you are working with is correct. Once you open the file successfully, you don't need to worry about problems such as reading a word where a word is expected, a missing colon, a blank line, etc.

- Your program must use good programming style. This includes things like appropriate variable names, good comments, minimizing the use of magic numbers, etc. Your program should begin with a comment that includes your name, student number, and a brief description of the program. **Each function should begin with a comment that describes its purpose, parameters (if any) and return values (if any).**
- Break and continue are generally considered bad form. As a result, you are **not** allowed to use them when creating your solution to this assignment. In general, their use can be avoided by using a combination of if statements and writing better conditions on your while loops.
- All graphical elements must be drawn by calling functions in the SimpleGraphics module. Do not call any of the graphics functions in the pygame module.
- Do not import any modules other than math, pygame, SimpleGraphics and time.

Looking for an A+? Allow songs to (optionally) include lyrics…

The lyrics for a song can be included in an ABC notation file by following a line of notes with a line that starts with "w:". The rest of the line has the word (or part of a word) that corresponds to each note on the line. The words may be separated by spaces or they may be separated by dashes.

Update your program so that it displays the lyrics when they are present (but continues to support files that do not include lyrics). You may assume that each word / part of a word that corresponds to each note will always be separated from the other words by either a space or a dash, and you may assume that the number of words will always match the number of notes. When displaying lyrics that are separated by dashes you should include the dash in what is displayed so that a singer knows that the word continues on the next note. Ideally, this dash is placed so that it is equally spaced between the two parts of the word that it is joining (unless it is for the last note on the line) though it is acceptable to display it immediately next to the earlier portion of the word for this assignment (which is slightly easier to implement). Twinkle, Twinkle, Little Star with the lyrics is shown below.



Grading

This assignment will be graded on a combination of functionality and style. A base grade will be determined from the general level of functionality of the program (Does it open the file successfully

(including error checking)? Are the header values printed out successfully? Is the shortest length identified correctly? Are the notes drawn with correct pitch and duration? Are the stems in the correct direction? Does the music fill each line? Is the tune played?). The base grade will be recorded as a mark out of 12.

Style will be graded on a subtractive scale from 0 to -3. For example, an assignment which receives a base grade of 12 (A), but has several stylistic problems resulting in a -2 adjustment will receive an overall grade of 10 (B+). Fractional grades will be rounded to the closest integer.

Total Score (Out of 12)	Letter Grade
12	A
11	A-
10	B+
9	B
8	B-
7	C+
6	C
5	C-
4	D+
3	D
0-2	F