FIT2102 Programming Paradigms 2024

Assignment 1: Functional Reactive Programming

Due Date: Friday, 30 August 2024, 11:55 PM
Weighting: 30% of your final mark for the unit
Interview: During Week 7
Overview: Students will work **independently** to create a game using Functional Reactive Programming (FRP) techniques. Programs will be implemented in TypeScript and use RxJS Observable streams to handle animation, user interaction, and other similar stream behaviours. **The goal is to demonstrate a good understanding of functional programming techniques as explored in the first five weeks of the unit**, including written documentation of the design decisions and features.

Submission instructions

Submit a zipped file named <studentNo>_<name>.zip which extracts to a folder named <studentNo>_<name>

- **It must contain all the code for your program** along with all the supporting files as well as the **report**.
- It should include sufficient **documentation** that we can appreciate everything you have done.
- You also need to include a report describing your design decisions.
- The only external library should be RxJS libraries supplied with the starter code.
- **Make sure the code you submit executes properly.**
- **Do not submit the node_modules or dist folders.**

The marking process will look something like this:

1. Extract **<studentNo>_<name>.zip**
2. Navigate into the folder named **<studentNo>_<name>**
3. Execute `npm install` and `npm run dev`
4. Open `http://localhost:5173` in a browser

Please ensure that you test this process before submitting. Any issues during this process will make your marker unhappy, and may result in a deduction in marks.

Late submissions will be penalised at 5% per calendar day, rounded up. Late submissions more than seven days will receive zero marks and no feedback.

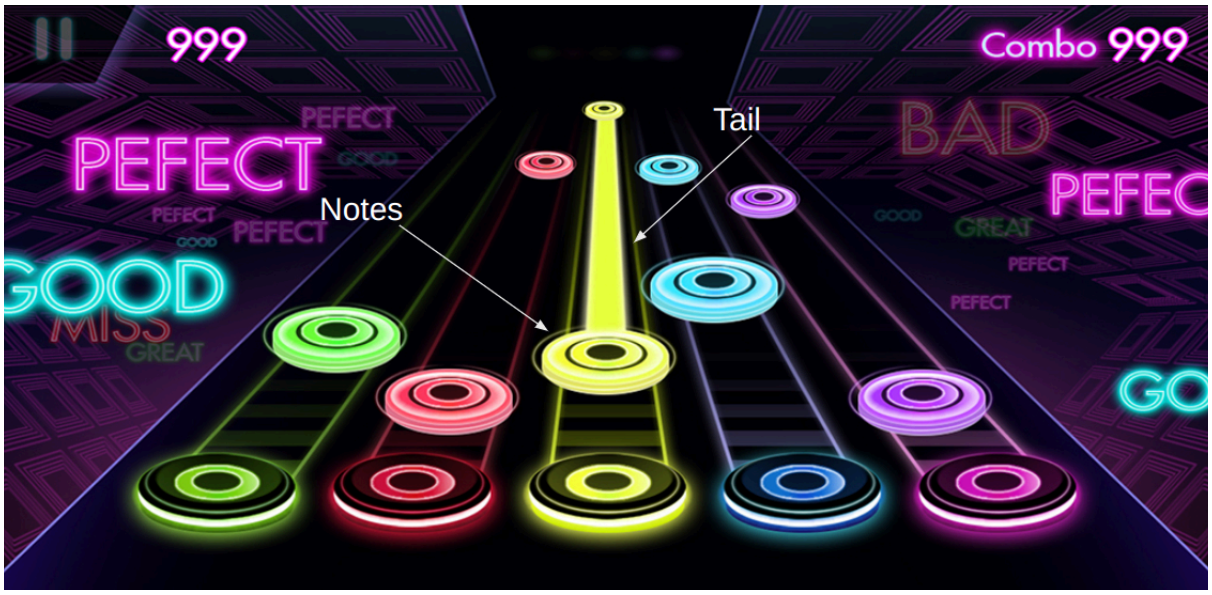
Git Instructions

We will be using Git for the assignment, however, this will mostly be self directed. There are no requirements on how many commits you need to the repo. However, we do recommend following good practices, and having frequent commits with meaningful commit messages. If any issues arise with academic integrity or submission, this will be used as evidence if you have completed your own work on time, if you have no commits, this will likely make it harder for you to clear yourself of any possible academic integrity issues, so we **highly recommend** you follow good practices.

The assignment uploaded to moodle, will be used for marking, unless there are exceptional circumstances which prevented you from uploading to moodle, at which point, we will be marking the last version committed to Git before the due date.

The instructions for the setup are posted on Ed, and follow them to setup the repo and access the skeleton code.

Table of Contents



Task description

In this assignment, we will use the RxJS Observable stream explored from Week 3 to create the classic Guitar Hero game in an SVG canvas. You will be provided with a starter code bundle similar to the applied sessions, including instructions on usage.

The image above and the Wikipedia page are meant to give you an idea of the gameplay, but yours needn't look the same or work in precisely the same way, especially with regard to graphics. **Note that only a subset of the features discussed in the link will be part of the requirements.**

You will also need to write a report, as described below.

Requirements

The game must be implemented in a good functional reactive programming style to get marks. A subset of the game's features will be required to get a passing grade. A greater subset of features will be required to get a higher grade. To achieve the maximum marks for this assignment, you will have to use a little creativity and add some non-trivial functionality of your own choice.

Minimum requirements

All of these requirements must be reasonably executed to achieve a passing grade

- A game board with four columns
- Circles appear from the **top** of the board and move **down** in a continuous manner, where each circle aligns with a music note.
- Notes can be played by using keys for each note when the circles align with the bottom row.
  - You can use any keys you want to, but this must be documented somewhere easy to find for your marker.
- Notes are read from the file provided as input to the main function (see **Note Specification** for specification)
- The timing of the notes must align with the given CSV file.
- The notes **not** for the current instrument, will need to be played in the background (i.e., without being displayed on the gameboard) at the correct time.
- Notes demonstrate reasonable behaviour
  - Appear heuristically (a simple heuristic will suffice) across all four columns
  - Notes disappear when they have been played
- Each note is played for the correct duration in which they are played.
  - If the key press, does not correctly align with a note, it will be played for a **random** duration between 0 and 0.5s
- Each time a key is pressed
  - The correct note must be played if the circles align with the bottom row
  - Otherwise, a **random** note is played.
- Scores must be kept track during the game, for both hitting and missing notes.
- The game should end when the song finishes playing.
- A short 1-2 page PDF report detailing your design decisions and use of functional programming techniques discussed in the course notes

Full Game requirements

Meets minimum requirements and has additional features

- If the note is longer than one second, the notes must have tails, where the tail represents the length of the note.

- The user must hold down the correct key for the length of the tail to ensure it is ‘correctly’ played
    - The score will update, iff the note is played for the correct duration
    - If the player lets go of the key too early, the note stops playing
- A score multiplier must be included, starting at 1x and increasing by 0.2 for every 10 consecutive notes hit (e.g., 10 notes = 1.2x, 20 notes = 1.4x), and resetting to 1x when a note is missed.
- Smooth and usable gameplay.
- See video for an idea of appropriate gameplay. *Note: This is not a full implementation but is meant to showcase what a game might look like.*

Additional requirements

See the Additional Information and How to get a High HD sections.

Report

Your report should be 300–600 words in length, plus up to 200 words for each **significant** additional feature, where you should:

- Include basic report formatting headings/paragraphs and diagrams **as necessary**
- **Summarise** the workings of the code and **highlight** the interesting parts (don’t just describe what the code does, we can read the source code!)
- Give a high level overview of your **design decisions** and **justification**
- Explain how the code follows FRP style and interesting usage of Observable
- How state is managed throughout the game while **maintaining purity** and **why**
- Describe the usage of Observable beyond simple input and **why**
- **Important**: Need to explain **why** you did things
- **Do not include screenshots of code unless you have an exceptional reason**
- This should be concise and straightforward, you may use dot points

Your marker will be instructed to stop reading if your report is too long, and only mark the first 600 (+200 per feature) words.

Plagiarism

We will be checking your code against the rest of the class and the internet using a plagiarism checker. Monash applies strict penalties to students who are found to have committed plagiarism. Additionally, we will be conducting an interview, which gives you a chance to explain your code and help us understand your code better. As long as you wrote your own code, there is nothing to worry about during the interview process.

AI statement

As per the AI statement on Moodle, use of generative AI in this unit is unrestricted. However, all code generated with AI **must** be properly **cited** in the form of code comments stating what has been generated and the scope of its use. You must be able to demonstrate understanding of all code submitted as part of your assignment, inability to explain any submitted code may result in an academic integrity case.

Definition of a Note

There are two types of notes:

1. `user_played == True`: This is the note which will be played by the user during the game.
    - Visual Representation:
        - Appearance: In the game, notes are represented as coloured circles that travel down the screen. Each note will be in a column, however, the column does not need to have one-to-one mapping with an individual pitch value.
        - Columns: The game screen is divided into four columns, each associated with a different button or key.
    - Musical Correspondence:
        - Sound Trigger: When a note reaches the designated row at the bottom of the screen, the player must press the corresponding button or key. If done correctly and in time, this action "plays" the note, meaning it triggers the associated musical sound.
        - Timing: The timing of pressing the button or key is crucial. The game typically rates the accuracy of the player's timing, which affects the score and the quality of the performance.
2. `user_played == False`: The note will be played by your code, but will not be shown to the user in the game, but played using the music library.

Note Specification CSV

Each note is specified by five columns: `user_played`, `instrument_name`, `velocity`, `pitch`, `start (s)`, `end (s)`.

- If the `user_played` column is `True`, this note should appear in the game, otherwise, the note should be played in the background.
- The `instrument_name` will be the instrument for which the note should be played in from the sample library
- The `velocity` of the note represents the **volume**, between 0 and 127. **Note**: the API requires this to be in the range [0,1]
- The `pitch` of the note.
- `start (s)`: start time in seconds, which is when the note should be played
- `end (s)`: end time in seconds, which is when the note should be stopped.

How to play a note:

To play a note, you can use the **given** `samples` dictionary. This is already set up in the skeleton code and is unlikely to be changed. Each instrument from the CSV, can be used as the key in the dictionary. You can use the `triggerAttackRelease` function. This takes four arguments:

1. The tone to play, which can be specified using the `pitch` from the CSV
2. The duration of the note in seconds
3. The start time, set to `undefined`, to start playing the note instantly.
4. The volume of the note, which corresponds to the velocity, in the range [0,1]

An example of playing notes is shown in the given `midi_example.ts` file, which you can refer to, but is also provided below.

```
 samples[instrument_name].triggerAttackRelease(
   Tone.Frequency(pitch, "midi").toNote(), // Convert MIDI note to
frequency
   1, // Duration of the note in seconds
   undefined, // Use default time for note onset
   velocity, // Set volume
 );
```

Additional Information: Marking Criteria and Suggestions

This section is not essential for completing the assignment, and is provided purely for context and additional information to answer common questions students may have.

Marking (30 marks total)

The goal of this assignment is to assess your understanding of FRP and Functional Programming. The marking has three broad sections:

1. Implementation of game features
2. Usage and understanding of proper functional programming style
3. Usage and understanding of RxJS and Observable

It is important to realise that:

- To receive a **Pass** grade by implementing the **Minimum requirements**, demonstrating application of functional programming ideas from our lectures and applied sessions.
  - You can receive up to a **Distinction** for perfectly implementing the **Minimum requirements** and demonstrating an excellent understanding of how to use Observable to write clean, clear functional UI code.
- To achieve a **High Distinction**, you will need to implement the **Full game requirements**
- To achieve the **maximum possible marks**, you will need to implement the full game requirements plus some aspect of **additional functionality,** as described below.

Note that it is essential to follow the submission instructions, as deductions may be applied for failing to follow the submission instructions.

We will mark 5 sections – Report, Functional Programming style, Code Quality, Observable and RxJS usage, and Game Features (including advanced features) – that are individually weighted.

Code that does not use Observable will **not** get a passing grade; games that use imperative, impure, or mutable code will be heavily penalised.

The rubric and marking guide are provided here.

Report (4 marks)

The report is intended to demonstrate your theoretical understanding of functional reactive programming, highlight design decisions, and help your marker appreciate the work that you have put into this assignment.

Important considerations for the report:

- Design decisions need to be correct
- Need to display understanding of course material
- Reports must demonstrate knowledge of FRP to achieve a passing mark
- **Marks can be awarded for students identifying issues with the code and how they can be addressed**
- Avoid filler in the report, but include enough information to show your marker that you have understood the core concepts

Functional Programming style (8 marks)

This section is about using what we have covered in lectures and tutorials. This involves concepts like:

- Small, granular functions
- Reusable functions, avoiding duplicate code
- Purity / referential transparency
- Fluent interfaces and fluent coding style
- Manipulation of different complex types and generic types
- HOF, curried functions
- Function composition/chaining

To achieve the maximum available marks, it is important to not only use advanced functional programming concepts, but do so in a useful way – for example, improving the readability of the code or following a declarative programming style. For example, simply currying all your functions will not receive marks unless they are partially applied somewhere and used appropriately

You may also attempt to use Lambda Calculus concepts in your code; however, be careful as they can often just make things hard to understand – it will be important to explain their usage in your report, so your marker can better appreciate your work.

Deductions will be applied for improper usage of types, including unjustified “any” types.

Code Quality (8 marks)

This section loosely covers anything to do with how readable and understandable your code is. Applying a good functional programming style tends to increase the readability of your code. **It is important that your code can be easily understood to help your marker appreciate your work.**

Some examples of what we look at are

- Appropriate line lengths (<80 characters)
- Documentation and commenting (should explain why the code is the way it is)
- Logical structuring of functions and variables, including overall flow of program logic
- Appropriate variable naming
- Consistent and understandable formatting

Using a linter and formatter may help greatly with this section. See below for tips and suggestions.

Observable and RxJS usage (8 marks)

This section covers usage of FRP – did you use Observable well?

Some important considerations:

- **Must manage game state in Observable**, and use the **scan** and **merge** operators to get a **passing** mark (please refer to the Asteroids example)

- Must handle creation of a stream of notes using Observables and an appropriate set of operations

- Usage of Observable as per discussed in the lectures, applied sessions, workshop, and in the Asteroids example, while maintaining purity, is sufficient for a **high** mark in this section if implemented very well and without issues

- **To achieve the maximum marks available, we want to see interesting and creative uses for Observable and RxJS operators (original work)**
  - This can involve implementing custom Observables and research into the RxJS operators documentation
  - Refer to the marking guide for a breakdown of what is required.

Other considerations:

- Side effects should be contained as much as possible
- Using additional RxJS operators that are not covered in class, or using the ones we introduce in interesting and novel ways, will be awarded additional marks (given that they are appropriate and useful)

Game Features (2 marks)

This section is about whether your game fulfils the requirements, and the overall complexity of your game (and thus the implementation).

Adding features should not come at the expense of the other criteria – a well implemented game with fewer features may and, often will, achieve a higher mark than a less well implemented game with more features.

Important: You will receive marks for implementing game features, but **this mark will also cap your total mark**.

- The maximum mark possible for implementing **minimum game requirements** is 70 (Distinction)
- The maximum mark possible for implementing **full game requirements** is 90 (HD)
- To achieve the maximum available marks (90+), you must implement **advanced requirements**

Some marking considerations:

- Extra features must follow FRP
- Advanced requirements can be not just gameplay but extra FRP features too
- Tests: for full marks, tests need to be **comprehensive** and not just simple/random test cases – they should guide development
- Bugs and other gameplay related issues will **not** be deducted from this section and be deducted from the total mark
- The total mark cap will be increased when implementing additional features. It is *possible* to achieve an HD by implementing the minimum game requirements and *some* full game requirements

To achieve the maximum available marks, features should be significant and change how state is managed in interesting ways. Discussed further below.

Bonus marks are available for particularly novel, impressive, or advanced features. Note that marks cannot exceed 100% of the total available marks.

Rubric

The rubric consists of **Marking bands** that represent the possible grade values for implementing requirements. This will be a **cap** on your final mark.

The **Marking guide** is what TAs will be using to mark your assignment, and what will contribute to your final mark/grade for this assignment.

Marking bands (summary of marking guide)

Code/Report quality	**Implementation**		
	Minimum requirements	*Full game*	*Full game + extension(s)*
Any of the following are not acceptable: Use of imperative code, TypeScript compile errors, `any` types, Not using rx.js, No comments, Missing or unreadable report, Missing instructions for how to play the game	Not passing.	Not passing.	Not passing.
Pure functional code (except in `subscribe` handlers), no compile/runtime errors, basic comments, basic report covering the implemented features. Uses Observable for state management.	P	C	C
Effectively uses Observable for state management, has generic types, and side effects are identified; comments are brief, only describing the implementation. The report demonstrates basic understanding of FRP principles. Functions are used for broad high level behaviour.	C	D	D
Small pure functions, immutable data and reusable code exploiting parametric polymorphism, side effects are contained; complete comments explaining the rationale and choices made in code. Advanced usage of Observable, including custom implementations. Detailed report of implemented features that demonstrates strong understanding of Functional Programming and FRP.	D	HD	HD (90+)

Marking guide

	FP Style	**Code Quality**	**Observable**
0 - 1 mark	Code is written in an imperative style, use of for/while loops and mutable variables (let/var). Modifies mutable data structures that aren't declared as read only to handle state management. No use of FP	Code is completely unreadable. Contains very large code blocks with complex nested logic and long lines. Excessive use of single letter and/or vague function names.	No use of observables. Uses DOM to store state or does not use Observable to store state.
2 marks	Some use of FP but has not demonstrated good understanding. Many functions are impure and modify state.	Code is difficult to read and requires careful analysis to understand intent. Many poor choices for variable names and many examples of complex nested logic with lack of documentation.	Some use of Observables, but does not utilise RxJS operators such as scan to effectively handle state. Observable callbacks contain impure code outside subscribe.
3 marks	Demonstrates some understanding of FP. Code contains some impure code. Use of HOF, but not utilised effectively.	Reader is able to get a general idea of code, but is difficult to read. Contains long lines and large code chunks. Some attempt at using functions and splitting up complex logic.	Uses Observables to handle state management and user interaction. Some Observable methods are not used effectively or not as intended, which demonstrates a lack of understanding.
4 marks	Style and structure is adapted from Asteroids example, but is not adapted to fit Guitar Hero. Code is entirely pure.	Able to get the general idea of code. Contains many complex structures, and large chunks of code that require refactoring. Minimal documentation	Uses observables to handle state management and user interaction. Uses subscribe to handle stream logic; overuse of subscribe callback.
5 marks	Similar style to the asteroids example, effectively adapted to new context.	Can tell the purpose of each piece of code. Contains documentation, but	Good use of basic Observables from the unit. Some methods in the

	Code is entirely pure and utilises the state management system introduced in the Asteroids example.	some comments are redundant. Some long lines and large blocks, but generally minimised.	Observable stream are overly complex and can be broken down more appropriately.
6 marks	Improves the Asteroids example considerably for the new game context. Good use of small modular functions and HOF. Shows great understanding of course content.	Code quality is of similar level to the Asteroids example in the notes.	Utilises Observable structure covered in unit content effectively. Good use of using observables for state management.
7 - 8 marks	Applies FP concepts in original ways beyond the Asteroids example. Great use of HOF, modular functions and a custom type system. Demonstrates fantastic understanding of course content in novel and interesting ways.	Code is easy to read, intuitive and flows well. Self documenting (descriptive variable names, easy to follow code flow). Well documented and comments are provided when needed. No long lines, and code is broken into readable chunks.	Uses interesting Observable methods not covered in course content. Uses custom Observables/Subject.

	0 - 0.5 marks	1 mark	1.5 marks	2 marks
Report	Not written or does not correspond to submission. Provides a summary of the code. Contains some justification, but focuses too much on summarising code. Contains too many screenshots of code.	Provides a summary of code with reference to FRP principles followed. Demonstrates some understanding of FRP and how it was used to manage state. Some justification for design choices with some focus on why.	Clearly written and concise. Provides a good summary of code. Design choices are justified and considers tradeoffs. Relates design choices to FRP and course content. Good understanding of FRP and pure state management.	Clearly written and concise. Highlights only key aspects of the code. Strong understanding of FRP and how it is used to manage state. Design choices are well justified, and considers non-trivial alternatives and tradeoffs.

Features	**Marks**	**Running total**	**Classification**
Notes are read from the given CSV file	**0.25**	0.25	**Minimum (≤ 70)**
Circles corresponding to each note move down from the top continuously	**0.25**	0.5	
Notes not for the current instrument are played in the background	**0.25**	0.75	
Timing of the notes aligns with CSV file	**0.25**	1	
Score	**0.25**	1.25	
Game ends when the song ends	**0.25**	1.5	
Notes are played when a key is pressed (correct notes if pressed at the right time, otherwise a random note)	**0.5**	**2**	
Score multiplier	**0.25**	2.25	**Full (≤ 90)**
Note tails have correct length corresponding to length of note if note is	**0.25**	2.5	

longer than one second			
Note tails move down along with the circle continuously	**0.25**	2.75	
Notes with tail stop playing when key is let go	**0.25**	3	
Notes with tails must be held for correct duration to be played correctly	**0.5**	**3.5**	
Advanced feature	**0.5**	**4**	**Advanced (90+)**

How to get an HD or High HD

To achieve a mark in the HD range, you need to implement a complete game with good style. To get in the high HD range, you will also need to implement **advanced features**.

One or more of the following (or something of your own devising with a similar degree of complexity) done well (on top of the basic functionality described above) will earn you a high HD, provided it is implemented using the functional programming ideas we have covered in lectures and classes:

- Create unit tests and create a file **tests/main.test.js** which are **comprehensive** and **guided the development of the program**
- Ability to pause/restart a game
- The ability for users to choose a song for your game
- Power ups or Multipliers for some notes, which give bonus scores to the user.
- *Advanced (not recommended unless you already know how): Make a distributed multiplayer version, wrapping the comms in Observable (you'll have to provide your own server for this).*

In general, **additional features for achieving HD and high HD will have to non-trivially impact your state management and/or overall complexity of the game**. For example, a power-up that changes the speed of the notes does not require interesting usage of state on its own, but if power-ups decay over time, then that would be more interesting and non-trivial.

Note that adding features will grant you a higher grade **under the condition that it is done in proper Functional and FRP style**. For an example of the proper style, refer to the example Asteroids Game described in the Course Notes.

Tips and suggestions

These are not part of the explicit requirements, but are things we may look at as part of the marking criteria. For example, poor choices of variable names may not have an explicit deduction but may impact your code quality mark as it makes the code hard to read.

Tips for getting started.

- Complete the Week 3-5 RxJS exercises and begin studying Observable in the course notes.
- Once you have completed the above, work through the example Asteroids Game described in the Course Notes. Follow the same framework to begin adding functionality to **main.ts** as above.

More tips.

- Finish all the JavaScript and TypeScript exercises and the course notes FRP material first. They are designed to give you the skills you need to prepare for this assignment
- Come to the workshops and applied sessions for important tips and assistance
- **Attend consultations given by the teaching team.** They are often sparse or empty around the time assignments are released, so it can be a great opportunity to get more detailed guidance and feedback
- Any general questions should be directed to the Ed forums when possible. However, try to avoid posting potential solutions. If you cannot make the consultations, you may make a ***private*** post for the assignment with your code.
- Your code should include brief comments to explain logic and design choices where necessary, or to refer to detailed explanations in your report. Please do not add comments that are self-evident from the code, e.g.

  ```
  const x = 1; // variable x is set to 1.
  ```
- **Start as soon as possible**. Do not leave the assignment until it's too late.

Recommended coding practices

- Structure your program in a consistent and coherent manner (group relevant functions, declarations, and variables together)
- Use block/section comments to clearly lay out each part of your code
- **Use nice indenting and formatting**
  - By default, Prettier is given to you and is integrated with VSCode. This includes format on save by default.
  - Follow the instructions in the README to manually format your code.
  - If you choose to use a different IDE, it will be left up to you to set up the formatter to your own satisfaction.

- Use camelCase for names, UpperCamelCase for types, and UPPER_CASE for constants.

Changelog

- 06/08/2024: Updated a sentence from:
  - The notes not for the current instrument, will be played which contains the rest of the song → The notes **not** for the current instrument, will need to be played in the background (i.e., without being displayed on the gameboard) at the correct time.
- 06/08/2024: Update late penalty from 10% to 5% as per the current Monash policy
- 07/08/2024: Fix `midi_example.ts` in starter code to use the correct velocity as per the Tone.js library (`midiMaxValue / 4` was changed to `0.25`)
- 07/08/2024: Minor improvements to the starter code:
  - Updated the `main` function to take in a `samples: { [key: string]: Tone.Sampler }` parameter and pass the `samples` object to `main`
  - Renamed `start_game` to `startGame` and `csv_contents` to `csvContents`
  - Overall, this means the following lines were changed in `src/main.ts`:
    - Line 108:
      ```
      export function main(csv_contents: string) {
      ```
      to
      ```
      export function main(csvContents: string, samples: {
      [key: string]: Tone.Sampler }) {
      ```
    - Line 216:
      ```
      const start_game = (contents: string) => {
      ```
      to
      ```
      const startGame = (contents: string) => {
      ```
    - Line 220:
      ```
      main(contents);
      ```
      to
      ```
      main(contents, samples);
      ```
    - Line 235:
      ```
      .then((text) => start_game(text))
      ```
      to
      ```
      .then((text) => startGame(text))
      ```