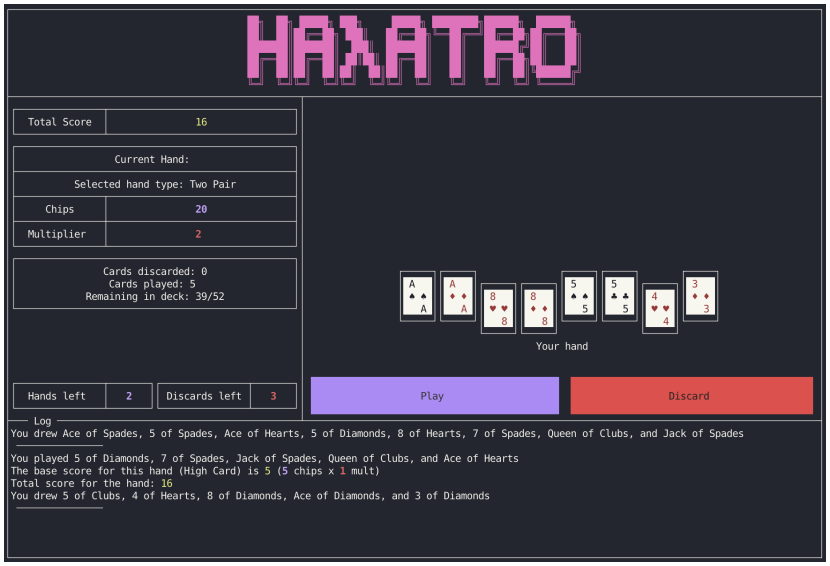
This document lays out the specification for the first coursework for CS141 Functional Programming. You should read this document **in its entirety** before starting on the coursework, even if you think you understand what is needed.



Balatro is video game released in late 2023 that uses card game mechanics. In the game, you play successive *hands* of cards, and the aim is to score as many points as possible—with more "powerful" hands scoring more points.

> Note to people who already know of Balatro: This coursework is a significantly simplified version which only considers Ante 1, Round 1, with an unmodified deck. Almost every other feature of the game is out of scope for the coursework.

In this coursework, you will implement a simple backend for a simplified version of the video game, called **Halatro**. The final task will be to implement an AI player which uses a clever strategy to maximise the score it gets in the game.

If you have Haskell set up on the DCS machines, you can try a working version of the game by running this command in a terminal:

```
halatro
```

Specification

This section contains everything you need to do in order to meet the requirements of the coursework.

To get started, you will need a copy of the coursework skeleton code, which you can clone:

```
git clone https://cs141.fun/halatro
```

There are five exercises to complete, just as you have been completing exercises on lab sheets.

All of your code should be written inside of the `CourseworkOne` module, which is located at `src/CourseworkOne.hs`. If you choose to edit any files outside of that, make it very clear at the top of `CourseworkOne.hs` that you have done so.

Navigate to the `halatro` directory and run the following command:

```
stack test
```

This will run the test suite. You should run this command regularly as you work on the exercises, to make sure that you are on the right track.

Submission

You should submit your work via Tabula.

To submit your work, perform **ALL OF THE FOLLOWING STEPS, IN ORDER:**

- Make a backup of your project, just in case.
- Delete the `.stack-work` directory, which contains build artifacts. (It is easy to do this directly from the file manager inside Visual Studio Code.)
- Compress the **whole project folder** into a .zip or .tar.gz archive.
- Upload your archive to the Tabula submission portal in the usual way.

It is your responsibility to ensure your submission is complete and contains the work that you intended to submit. Failure to follow the above steps will **not** be considered mitigating circumstances and delayed submissions as a result will earn the appropriate late penalty.

A Crash Course in Card Games

Here is a quick primer of card games, for those unfamiliar with how they work.

There are 52 cards in a deck of cards. Each card has a *rank* and a *suit*.

- The *rank* of a card is one of thirteen options: a numbered card with a number from 2 to 10; a face card (Jack, Queen, King); and Ace.
- The *suit* of a card is one of four options: Hearts ♡, Diamonds ♢, Clubs ♣, and Spades ♠. Traditionally, Hearts and Diamonds are red, and Clubs and Spades are black.

A *hand* is a set of cards. In Halatro, we will distinguish between a "hand", which is the set of cards the player has at one time, and a "played hand", which is the subset of UP TO FIVE of those cards that the player chooses to play on their turn.

Part One: Many Hands...

How good a played hand is depends on the combination of cards that you have in the played hand. There are ten possible hands of interest, listed in order from worst to best:

- **High Card:** The hand contains no better hand types and is nonempty - the highest ranking card in the hand is scored.
- **Pair:** The hand has two cards of the same rank.
- **Two Pair:** The hand has two sets of two cards of the same rank.
- **Three of a Kind:** The hand has three cards of the same rank.
- **Straight:** The hand contains five cards of sequential rank.

  NOTE: an Ace can also count as a 1, so A-2-3-4-5 is a valid straight and A-K-Q-J-10 is a valid straight. (but it can't be both at once; Q-K-A-2-3 is not a straight).
- **Flush:** The hand contains five cards of the same suit.
- **Full House:** The hand contains a three of a kind and a pair, which do not overlap.
- **Four of a Kind:** The hand contains four cards of the same rank.
- **Straight Flush:** The hand contains five cards of sequential rank, all of the same suit.
- **Royal Flush:** The hand contains the Ace, King, Queen, Jack, and 10 of the same suit.

An empty hand has a hand type of `None`.

A played hand may contain more than one hand type. For example, a played hand that contains a Royal Flush also contains a Straight Flush, and a Straight, and a Flush; the played hand A-A-A-2-3 of any suits contains both a Three of a Kind (AAA) and a Pair (AA).

> **Ex. 1.** It will be convenient to know what hand type(s) a played hand of cards contains. Implement the `contains` function, which takes a played `Hand` of cards and a `HandType` and returns whether the played hand contains that hand type.

The test suite contains, for each hand type, one or two hands that contain that hand type, and one or two that *do not* contain that hand type. You should use these to check that your implementation is correct.

Important: Instead of the above, you may prefer to align your implementation with the "real" rules of Balatro, which deviate slightly from the above. If you are doing this, please note it in your comments and I will use an alternative set of tests. (If in doubt, use the above definitions.)

Part Two: Simply the Best

We only consider the best possible hand type when computing the score of a played hand. The game will determine the best type that can be made, based on the ordering given in Part One.

> **Ex. 2.** Implement the function `bestHandType`, which takes a played `Hand` of cards and returns the **best** hand type that the played hand contains.
>
> If the played hand is empty, return `None` instead of a hand type.

Part Three: Score Draw

The aim of Halatro is to play hands that score as many points as possible. We will now implement the logic to work out how many points a played hand is worth.

Scoring a hand in Halatro is split into three stages:

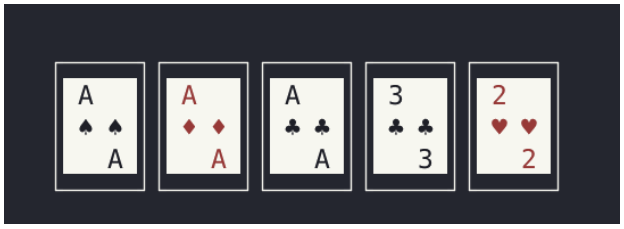
1. Figure out the best hand type a played hand contains, and get the base score for that hand type.
2. Add bonus chips for each card that contributed to the hand type.
3. Multiply the total by a multiplier.

Firstly, we should get the base score for a hand type. The base scores for each hand type are as follows:

Hand Type	Base Chips	Base Multiplier
None	0 chips	x0
High Card	5 chips	x1
Pair	10 chips	x2
Two Pair	20 chips	x2
Three of a Kind	30 chips	x3
Straight	30 chips	x4
Flush	35 chips	x4
Full House	40 chips	x4
Four of a Kind	60 chips	x7
Straight Flush	100 chips	x8
Royal Flush	100 chips	x8

You can find this data in a function in the `Halatro.Constants` module, and you should directly make use of that data in your implementation, rather than hardcoding the values.

Example. Consider the following played hand:



The best hand type is a Three of a Kind (AAA). The base score for a Three of a Kind is 30 chips, and the base multiplier is 3.

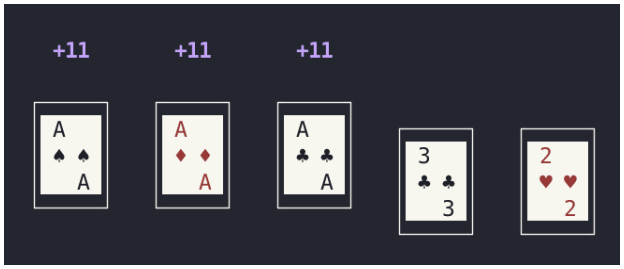
Secondly, we add **bonus chips** for every card that contributed to the hand type. The bonus chips for each card are as follows:

You can find a function to get the number of bonus chips from a card's rank, called `rankScore :: Rank -> Int`, in `src/Halatro/Constants.hs`.

Rank	Bonus Chips
2-10	equal to card rank
J,Q,K	10 chips
A	11 chips

IMPORTANT: Cards that aren't part of the best hand type are not scored.

Consider our example hand again.



Since only the Aces in the hand contribute to the Three of a Kind, we add 11 chips for each Ace, for a total of 33 bonus chips. The 2 and the 3 are not scored.

Ex. 3. Implement the function `whichCardsScore`, which takes a played `Hand` of cards and returns only those cards which score (i.e. those that contribute to the best hand type).

To get the total hand score, we multiply the total chips (base chips plus any bonus chips) by the **multiplier**.

Example. The final score for the hand A-A-A-3-2 is $(30 + 33) \times 3 = 189$ chips.

Ex. 4. Implement the function `scoreHand`, which takes a played `Hand` of cards and follows the above process to compute the number of chips that the hand is worth.

If your answers to Ex. 1-4 are correct, you have now fully implemented the backend logic for Halatro! You can play your own implementation directly with the following command:

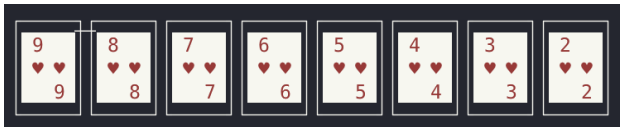
```
stack run
```

Part Four: Choices

In Halatro, the player has more than 5 cards, and they will need to choose up to 5 cards to form the hand that they will play. In order to do that, it will be useful to know what the best hand type they can play is.

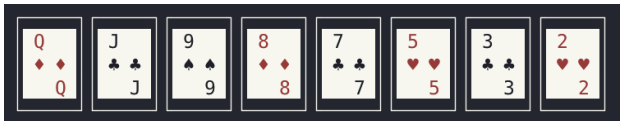
> **Ex. 5.** Implement the function **highestScoringHand**, which takes a list of cards and finds the best set of cards that can be played (according to the score).

Example. Consider the following hand:



Here there are four different straight flushes we could play, but we want to play 5-6-7-8-9 as it scores the most bonus chips.

Example. Consider the following hand:



Here there is no combination of cards that can form a hand type, so the best option is to play the Jack and/or Queen. We can play any other cards alongside the Jack or Queen, but none of them will score.

Note: The test case corresponding to this example is overly restrictive and it expects you to play the Queen, even though playing the Jack but not the Queen would score the same number of points. If your method of solving this test case would not naturally include the Queen in its set of played cards, then you are very welcome to ignore, fix, or remove

the failing test corresponding to this example.

Part Five: Artificially Intelligent

For the final exercise, you will create an AI to play Halatro as well as possible.

In addition to scoring hands, the player has the option to **discard** up to five cards and replace them with newly drawn cards. The AI will need to decide which cards to discard or play in order to maximise the score in the game. They have up three hands to play and can use up to three discards. The final score at the end of the game is the total score across the three played hands.

Your AI will have access to the following information: a list of the cards in their hand, and the moves that have been made so far in the game. The result should be a `Move`—a datatype comprising a list of one or more cards that are in hand, and either `Play` or `Discard`.

To test your AI that you have written, you can use the following command:

```
stack run ai
```

and you can watch it play the game.

The test suite also includes a test for the AI, but if you want to run many more than 100 tests, you can use a command like

```
stack run total 1000
```

and it will run the game 1000 times and print out the average score that your AI achieved.

The function called `myAI` is the one that will be used in both of the above commands. If you want to test the behaviour of AIs other than the final one, you can change `myAI` to be equal to `simpleAI` or `sensibleAI` in `CourseworkOne`.

> **Ex. 6.** Implement the function `simpleAI`, which always plays the five highest-ranked cards in hand.

Ex. 7. Implement the function `sensibleAI`, which plays the highest-scoring hand that can be made from the cards in hand, but doesn't use any discards. (Note: there are sometimes multiple possible highest-scoring hands; it is fine to play any of them.)

This is the AI that is included with the Halatro binary on the DCS machines.

Ex. 8. Implement the function `myAI` which should play Halatro as well as possible, using hands and discards where appropriate. Your AI can use any strategy you like, and you are encouraged to experiment with different strategies to see what works best.

The average score of your AI over many games will be used to assess your solution.

That's it! Once you are happy that your AI scores as high of a score as you can reach, you are done—submit to Tabula according to the process on Page 2 of this document.

Assessment

The marks for the coursework are broken down as follows:

Marking Point	Criterion	Marks
Ex. 1 (contains)	Correctness	5
Ex. 2 (bestHandType)	Correctness	5
Ex. 3 (whichCardsScore)	Correctness	5
Ex. 4 (scoreHand)	Correctness	4
Ex. 5 (highestScoringHand)	Correctness	5
Ex. 6 (simpleAI)	Correctness	3
Ex. 7 (sensibleAI)	Correctness	3
Ex. 8 (myAI)	Optimisation	10
Overall	Code quality	10
Total		**50**

Tests and Correctness

If your solution meets the specification, you will receive full marks for correctness for that exercise. If you are unable to complete an exercise, you should still submit your work, as you will still receive credit for the exercises you were able to complete.

The tests include all the examples given in this document, as well as some additional tests to help ensure that your solution is correct. However, they **do not and cannot** cover every possible edge case. Therefore, you should make sure that you understand what is expected, and that you are confident your code will work on any input that meets the spec, not just the examples given.

When we check that your code is correct, we will be comparing against a reference implementation of the coursework over a very large number of randomly generated test cases. For obvious reasons, that implementation cannot be included with the skeleton code.

Final Exercise

Exercise 8 will be marked on **optimisation** - after a large number of runs, what is the average score that it achieves?

There is no firm guidance on what constitutes an "excellent" score. However, as a guideline, the "sensible" AI scores around 400 points without any discards, so you should aim to beat that comfortably. Any solution that gets at least 400 points on average will receive a passing grade for Ex.10.

There are well known strategies in Balatro that can score significantly over 400 points on the vast majority of games—feel free to use those as inspiration.

You are encouraged to compare your average scores to other people's (but NOT code or specific approaches, of course) to see how well you are getting on.

Beyond the Specification

If you want to, you are very welcome to go beyond the standard specification. You are welcome to extend the skeleton code to implement additional features, or make any other changes you like, **so long as the tests pass**.

Elegance

Elegance will be marked out of 10. This is a subjective measure of how idiomatic, readable, and well-structured your code is, based on how easy it is to read for an experienced practitioner of Haskell. In the interests of keeping the feedback return speedy, there will not be detailed individual feedback on code quality - but please reach out to the lab tutors, before the coursework submission or after, if you would like to discuss the quality of your code and how to make it more idiomatic.

There are many stylistic choices to make writing a Haskell program, from the smallest uses of syntactic sugar right up to the big questions about program architecture. Making the choices that result in elegant, readable, and reusable code is a skill that must be learned and improved over time.

- Is the code easy to read?
- Is the code idiomatic (playing into the strengths of Haskell)?
- Does the code make good use of library functions where relevant?
- Does the submission show a good separation of concerns?
- Is syntactic sugar used appropriately?

Sometimes you may make a choice which you think is controversial or makes your code less elegant, but you want to keep it for other reasons. If so, justify it in your documentation!

Code comments

While there are no marking points for the quality of comments, it is still a very good idea to include them - a code segment without proper documentation is no better than a mathematical answer without justification. If you do not document your code well and thoroughly, there's no reason to believe that the solution will be good, nor that you understand it.

Here are some questions you can ask yourself to help you write good documentation:

- Does the documentation explain what the code does?
- Does it show that you understand the work that has been done?
- Is it clear and easy to read?

Documentation will also be used if there is suspicion of plagiarism, to help us to judge whether the code is your own work.

Additional remarks on...

Importing useful functions

It may often occur to you to make use of some pre-existing functions. For example, the `base` library contains a lot of useful modules with functions which can simplify code and make it more elegant. This is encouraged! **You may make use of any functions from any modules in libraries included with the skeleton code (most usefully `base` and `containers`).**

Using more exotic libraries requires additional evidence from you that it's worth the extra effort of importing the package. This might be the case, but you will need to argue so in your justification. **Libraries which are *designed* to solve the coursework, or to solve very similar problems, may not be used**. If in doubt, speak with your lab tutor or Alex.

Plagiarism & Cooperation

Plagiarism is a serious academic offense which affects the integrity of the teaching experience. It will not be treated lightly. You are expected to complete the coursework as a **solo effort**. To that end:

- **You must not copy code or text written by any person other than yourself, or any tool, into your implementation, even if you change it.**
- **You must not share your own implementation, publicly or privately.**

If you have made use of outside resources (books, websites, other people's unit tests, etc.), you **must** name those resources in the head of your coursework file; explain how you used them; and give a link for where to access them.

While you may discuss general principles and overarching approaches, you **should not** ask other members of the CS141 course to help you directly implement your code. If

someone asks for assistance of this type, **you should not give it**. If you need a prod in the right direction, ask one of your your lab tutors in the labs, or speak with Alex.

For more information about cheating and plagiarism, see the relevant section in the DCS student handbook. If you would like to report any potential cases of cheating or you have a question about the requirements laid out here, I will be very happy to talk to you.

Frequently Asked Questions

Do I need to know Poker/Balatro/etc to do well? No. Everything you need to understand the coursework is in this document. However, there are plenty of resources online that relate to Balatro in particular that may help you to understand how Halatro works, and you are encouraged to use them.

Why does it take so long to compile on the lab machines? Unfortunately, `stack` is not able to safely cache the compiled libraries, so it recompiles them the first time you use them on a machine. Unfortunately, Brick (on which Halatro is based) is a big library, so it takes a while to recompile.

I keep failing the tests, but my code looks correct? Double and triple check your code. Also double check that you are fully meeting the specification—there may be important details that are not immediately obvious on a first read.

Bear in mind that some test cases depend on earlier functions being implemented correctly. If you are still having trouble after checking all of these things, contact your lab tutor.

Can I use ChatGPT/Copilot/Claude/etc to... No.

In this coursework, the use of generative tools is **explicitly forbidden**, for all use cases. That includes, but is not limited to:

- Generating ideas for how to approach exercises
- Explaining exercises to you
- Reporting on how an attempted solution could be improved
- Generating code
- Generating documentation

You are allowed to use the tools built into the Haskell Language Server in VS Code, including `hlint` (the blue squiggles), which will sometimes suggest meaningful simplifications or improvements to your code.

Strong positive evidence of AI tool usage (for example, if you are seen using it to solve the coursework) then it will be treated as a form of academic misconduct and with the same set of penalties.

It is fine to use AI tools while working on the lab exercises though, and that might help you to understand the coursework better. But remember, Alex and the lab tutors are always available to answer any questions you have, and to point you in the right direction. You are not alone!