	Tuples			*Records*	*Lists*
element types	differing			differing	uniform
selected by	order			label	order
type constructors	⟨⟩ * ⟨⟩	⟨⟩ * ⟨⟩ * ⟨⟩	⋯	`{a :` ⟨⟩ `; b :` ⟨⟩ `; c :` ⟨⟩ `; ...}`	⟨⟩ `list`
value constructors	⟨⟩ `,` ⟨⟩	⟨⟩ `,` ⟨⟩ `,` ⟨⟩	⋯	`{a =` ⟨⟩ `; b =` ⟨⟩ `; c =` ⟨⟩ `; ...}`	`[]` ⟨⟩ `::` ⟨⟩

Table 7.1: Comparison of three structuring mechanisms: tuples, records, and lists.

7.6 Problem set: The prisoners' dilemma

7.6.1 Background

I'm an apple farmer who hates apples but loves broccoli. You're a broccoli farmer who hates broccoli but loves apples. The obvious solution to this sad state of affairs is for us to trade – I ship you a box of my apples and you ship me a box of your broccoli. Win-win.

But I might try to get clever by shipping an empty box. Instead of cooperating, I "defect". I still get my broccoli from you (assuming you don't defect) and get to keep my apples. You, thinking through this scenario, realize that you're better off defecting as well; at least you'll get to keep your broccoli. But then, nobody gets what we want; we're both worse off. The best thing to do in this DONATION GAME seems to be to defect.

It's a bit of a mystery, then, why people cooperate at all. The answer may lie in the fact that we engage in many rounds of the game. If you get a reputation for cooperating, others may be willing to cooperate as well, leading to overall better outcomes for all involved.

The donation game is an instance of a classic game-theory thought experiment called the PRISONER'S DILEMMA. A prisoner's dilemma is a type of game involving two players in which each player is individually incentivized to choose a particular action, even though it may not result in the best global outcome for both players. The outcomes are commonly specified through a payoff matrix, such as the one in Table 7.2.

		Player 2	
		Cooperate	*Defect*
Player 1	*Cooperate*	(3, 3)	(−2, 5)
	Defect	(5, −2)	(0, 0)

Table 7.2: Example payoff matrix for a prisoner's dilemma. This particular payoff matrix corresponds to a donation game in which providing the donation (of apples or broccoli, say) costs 2 unit and receiving the donation provides a benefit of 5 units.

To read the matrix, Player 1's actions are outlined at the left and Player 2's actions at the top. The entry in each box corresponds to a payoff to each player, depending on their respective actions. For instance, the top-right box indicates the payoff when Player 1 cooperates and Player 2 defects. Player 1 receives a payoff of −2 and Player 2 receives a payoff of 5 in that case.

To see why a dilemma arises, consider the possible actions taken by Player 1. If Player 2 cooperates, then Player 1 should defect rather than cooperating, since the payoff from defecting is higher ($5 > 3$). If Player 2 defects, then Player 1 should again defect since the payoff from defecting is higher ($5 > -2$). The same analysis applies to Player

2. Therefore, both players are incentivized to defect. However, the payoff from both players defecting (each getting 0) is objectively worse for both players than the payoff from both players cooperating (each getting 3).

An ITERATED PRISONER'S DILEMMA is a multi-round prisoner's dilemma, where the number of rounds is not known.[14] A STRATEGY specifies what action to take based on a history of past rounds of a game. We can (and will) represent a history as a list of pairs of actions (cooperate or defect) taken in the past, and a strategy as a function from histories to actions.

For example, a simple strategy is to ignore the histories and always defect. We call that the "nasty" strategy. More optimistically is the "patsy" strategy, which always cooperates.

Whereas the above analysis showed both players are incentivized to defect in a single-round prisoner's dilemma (leading to the nasty strategy), that is no longer necessarily the case if there are multiple rounds. Instead, more complicated strategies can emerge as players can take into account the history of their opponent's plays and their own. A particularly effective strategy – effective because it leads to cooperation, with its larger payoffs – is TIT-FOR-TAT. In the tit-for-tat strategy, the player starts off by cooperating in the first round, and then in later rounds chooses the action that the other player just played, rewarding the other player's cooperation by cooperating and punishing the other player's defection by defecting.

In this problem set, you'll complete a simulation of the iterated prisoner's dilemma that allows for testing different payoff matrices and strategies.

[14] If the number of rounds is known by the players ahead of time, players are again incentivized to defect for all rounds. We will not delve into the reasoning here, as that is outside the scope of this course, but it is an interesting result!

7.6.2 Some practice functions

To get started, you'll write a series of functions that perform simple manipulations over lists, strings, numbers, and booleans. (Some of these may be useful later in implementing the iterated prisoner's dilemma.) See the comments in `ps1.ml` for the specifications. Give the functions the names listed in the comments, as they must be named as specified in order to compile against our automated unit tests.

The best way to learn about the core language is to work directly with the core language features. Consequently, you should not use any library functions in implementing your solutions to this problem set.

7.6.3 *Unit testing*

Thorough testing is important in all your work. Testing will help you find bugs, avoid mistakes, and teach you the value of short, clear functions. In the file `ps1_tests.ml`, we've put some prewritten tests for one of the practice functions using the testing method of Section 6.5. Spend some time understanding how the testing function works and why these tests are comprehensive. Then, for each function in Problem 1, write tests that thoroughly test the functionality of each of the remaining sections of Problem 1, covering all code paths and corner cases.

To run your tests, run the shell command

```
% make ps1_tests.byte
```

in the directory where your `ps1.ml` is located and then run the compiled program by executing `ps1_tests.byte`:

```
% ./ps1_tests.byte
```

The program should then generate a full report with all of the unit tests for all of the functions in the problem set.

7.6.4 *The prisoner's dilemma*

Having implemented some practice functions, you will apply functional programming concepts to complete a model of the iterated prisoner's dilemma, including the implementation of various strategies, including one of your own devising.

Follow the comments in `ps1.ml` for the specifications. Feel free to use any of the functions that you implemented in the earlier parts of the problem set.

All of the programming concepts needed to do the problem set have already been introduced. There's no need to use later constructs, and you should refrain from doing so. In particular, you'll want to avoid imperative programming, and you should not use any library functions.

7.6.5 *Tournament*

To allow you to see how your custom strategy fares, we will run a course-wide round-robin tournament in which your strategy will be run against every other student's strategy for a random number of rounds. To keep things interesting, we will add a small amount of noise to each strategy: 5% of the time, we will use the opposite action of the one a strategy specifies.

The tournament is pseudonymized and optional. If you want to participate, you'll provide a pseudonym for your entrant. If you

specify an empty pseudonym, we won't enter you in the tournament, though we encourage you to come up with clever strategies to compete against your fellow classmates! We'll post and regularly update a leaderboard, displaying each student's pseudonym and their strategy's average payoff. The top score will win a rare and wonderful prize!

7.6.6 Testing

You should again provide unit tests in `ps1_tests.ml` for each function that you implement in Problem 2.

In addition, you can choose to uncomment the `main` function in `ps1.ml` and then recompile the file by running `make ps1.byte` in your shell. Then, run the command `./ps1.byte` and you should see via printed output that Nasty earns a payoff of 500 and Patsy earns a payoff of −200. You can change the strategies played to see how different strategies perform against each other.