CS 3110 Fall 2014 **Due at 11:59 PM, 12/04/14, no late passes**
Problem Set 6
Version 2 (last modified November 30, 2014)

Revision log

Revisions in the document below are marked in blue to help you quickly identify them.

- [11/30/14] Clarified Confused status. Corrected usage of "attack" vs. "move." Revised description of steps of Battle Phase. Clarified handling of fainted Steammon. Stated that bot implementations must be independent of game implementations. This is version 2; there was no version 1.

Objectives

- Synthesize all you have learned in 3110 in building a complex piece of software.
- Practice reading a large requirements document and producing a piece of software that satisfies those requirements.
- Practice working in a software development team.

What we supply

We provide an archive `ps6.zip` that you should download from CMS.

Teams and source control

You are required to work with a small team of two to four students for this problem set. Each team member is responsible for understanding all parts of the assignment. You need not use the same team members as in previous problem sets.

You are required to use `git` to work with your team. Your repository must be private. We expect the `git` log you submit to show evidence of work over a period of time, not just a single commit at the end.

Specification and Testing

You should continue to document specifications for functions and modules you submit as part of your solutions. And you should continue to test your code as well as possible. We will not ask you, however, to turn in your test cases.

In this problem set, you will develop a game called *PokéJouki Emerald* and a bot (i.e., an AI player) that competes against other bots in the game.[1] This is a large project and an opportunity to demonstrate what you have learned about abstraction, style and modularity in the creation of elegant code.

Timeline

Friday, 11/14/14	PS6 released
Thursday, 11/20/14	Prelim 2
Wednesday, 11/26/14	Thanksgiving Break begins
Thursday, 12/04/14, 11:59 pm	Final submission due; **no late passes**
Saturday, 12/06/14, 6:00 pm	*PokéJouki Emerald* Tournament

Note that Cornell's calendar causes both Prelim 2 and Thanksgiving Break to interrupt your time for working on PS6, so careful planning on your part will be essential. (Historically, PS6 has featured design review meetings; however, since those meetings would either have to be the week of a prelim or the week of a break, we will not hold them this semester.)

On the Saturday after PS6 is due, there will be a *PokéJouki Emerald* tournament to which you are encouraged but not required to submit your bot. If you want to participate in the tournament, you must submit your bot both to the PS6 assignment on CMS as well as the tournament assignment on CMS. There will be lots of free food, and the chance to watch your bot perform live. The winners get bragging rights and will have their name posted on the 3110 Tournament hall of fame.

On this problem set, **no late passes will be accepted, and no extensions will be granted.** You must submit everything by 11:59 pm on Thursday, December 4. This constraint is to ensure that the course staff can be prepared to run the tournament on Saturday, and also to ensure that no course work is due during the Study Days for final exams.

[1] *Jouki* is a steampunk alternative reality to Japan.

Contents

1 Your tasks

There are two parts to the implementation of this project. Make sure you spend time thinking about each part before starting.

1.1 Implement the game (125 points)

Your first task is to implement the *PokéJouki Emerald* game in the file `game/game.ml`, and any files you choose to add. You may add files only to the `game` and `team` directories. You can use the sample team program we provide to help test your game, but for full testing coverage you will need to write your own unit tests as well as test bots.

When we grade your game implementation, we will primarily be concerned with running some sample match-ups whose outcomes are predictable. Your implementation needs to ensure that the progress of the game is rendered in the GUI, so that we can see those outcomes.

1.2 Implement a bot (75 points)

Your second task is to implement a bot to play the game. Your bot should be located a file named `team/bot.ml`. A very weak bot called `babybot` is provided; you can use it as a basis for your bot code. There are many different strategies for building a good bot. This is your chance to be creative and have fun creating a good AI. Your bot will have the opportunity to compete against other teams' bots in the Tournament.

When we grade your bot, we will primarily be concerned with how well it performs versus some staff-created bots of varying difficulty. Your bot should at a minimum be able to reliably beat `babybot` (provided in the release code).

1.3 Submit your solution

You will submit a zip of all files in your `ps6/game` directory, including even those you did not edit, and a zip of all files in your `ps6/team` directory. (Remember to delete any compiled `.cmi` or `.cmo` files, so that your zip file is not artificially large.) The other directories in `ps6/` should remain unchanged.

A grader should be able to unzip your submission, put the two directories in the right place, run `build_game.sh` to compile your game code, and run `build_team.sh` to compile your bot code. If you add new `.ml` or `.mli` files, you should add them to the build scripts. The scripts should compile your code without errors or warnings. **Submissions that do not meet this criterion will be heavily penalized.**

1.4 Advice

- **Design first.** This project is large and complicated. Without spending time on creating a design that is both solid and complete, you will quickly get bogged down in implementation. Before writing any code, you should have a clear idea of all of the following:

  - The information necessary to represent the state of the game.
  - How that information will be stored and accessed efficiently.
  - How you will factor your code into modules.
  - What the interfaces of your modules will be.
  - What invariants will hold in your modules.

- **Test incrementally.** Don't try to implement all of the game and test it with a single bot. Start with easy, individual pieces and work up to harder pieces.
- **Stay in synch.** Make sure that what is going on in the game state matches what is going on in the GUI. Updating one does not automatically update the other. If you are watching the game and something seems to go wrong, remember, it could just be the code controlling the output to the screen. Moreover, just because the graphics look correct doesn't mean the game is acting properly. It would behoove you to maintain some sort of invariant between the status of the game and the status of the graphics.
- **Be suspicious.** Bugs that appear to originate from your game implementation might actually be bugs with your bots, and vice versa.
- **Love your bot.** The bot part of the project is worth a substantial amount, so don't put off the bot part until the end. And your bot could win you fortune and glory as this semester's winner of the 3110 Tournament!

2 The Game

2.1 Players

Two bots, called *players*, battle each other. Each player controls a team of *Steammon*, which are the *PokéJouki Emerald* equivalent of Pokémon.[2] One player is designated the Red player and the other is designated Blue.

2.2 Steammon

A Steammon is a fantastical species. Each Steammon has certain *moves* it can make during a game, as well as various *stats*, which represent its current state. One important stat is *hit points* (henceforth, HP), which represents how much damage it can take during a battle. When a Steammon reaches zero HP, it *faints* and is removed from play.

[2] *Steam* is a traditional meme in 3110 games.

2.3 Draft

The game begins in the draft phase. A random player is chosen to have the first draft pick. That player chooses a single Steammon from a set of available Steammon; the chosen Steammon is removed from the set. The other player chooses two Steammon from the remaining set, which are removed from the set. The players continue to alternate their draft picks until the first player's team is *full*. The second player finishes by choosing a single Steammon. Both players now have full teams of equal size.

2.4 Inventory

The game enters the inventory phase. Each player starts with a certain amount of *cash*. The players simultaneously purchase *items* using that cash.

2.5 Battle

Each player selects a Steammon to be the *active* Steammon for that team. The battle phase commences. The battle phase is divided into *turns*, which alternate between players. During a turn, a player chooses an *action*, which can be one of the following:

- Cause its active Steammon to make a *move*.
- Switch its active Steammon to another Steammon on the team.
- Use an item.

After both players have chosen their action for the current turn, those actions are resolved. The team whose currently active Steammon has the higher Speed performs its action first. If the Speeds tie, then the order of resolution is chosen uniformly at random.

2.6 Object of the game

A player wins when all the Steammon on the opposing player's team have fainted.

2.7 Attributes of Steammon

A Steammon's *attributes* include

- up to two Steamtypes,
- up to four moves,
- stats,
- status effects, and

Charizard Fire, Flying

HP 78
Atk 84
Def 78
SpA 109
SpD 85
Spe 100

Move 1: RockSlide
Move 2: Earthquake
Move 3: FireBlast
Move 4: Surf

Figure 1: An example Steammon: Charizard

- modifiers.

See Figure 1 for an example Steammon. In the rest of this section, we describe each of these attributes.

2.7.1 Steamtypes

Steamtypes describe what a Steammon is strong at, and what a Steammon is weak against. The Steamtypes are Normal, Fire, Fighting, Water, Flying, Grass, Poison, Electric, Ground, Psychic, Rock, Ice, Bug, Dragon, Ghost, Dark, Steel, and Typeless. Much like a game of Rock–Paper–Scissors, certain Steamtypes are strong or weak against others.

2.7.2 Moves

Moves are the primary instrument of battle. A move has several characteristics, as described below. Moves that cause damage are also referred to as *attacks*. (Don't confuse attack moves with the attack stat, explained in the Stats section.)

- **Steamtype:** Each move has a Steamtype, called its *element*. An attack whose element is Electric, Fire, Water, Grass, Ice, Dragon, Psychic, or Dark Steamtype is considered a *special* attack, whereas an attack of any other element is considered a *physical* attack. For non-attack moves, the element is usually irrelevant.
- **Target:** Each move has a primary target, which is either the *user* (the Steammon making the move) or its *opponent*. Some attacks are actually self-directed, meaning the target of the attack is the user.
- **Power Points (PP):** Each move is associated with its own pool of PPs. The pool is unique to a given Steammon and a given move. Each time the move is used, it costs 1 PP from that pool. When the pool reaches 0 PP, the move may no longer be used.

- **Power:** This number is used to determine how much damage an attack will do.
- **Accuracy:** This number is used to determine how often a move will hit. The accuracy represents the percent chance of hitting the opponent out of 100.
- **Effects:** Some moves have a chance of causing an effect. Effects are described below.

2.7.3 Stats

A Steammon has the following stats:

- **hit points (HP):** The health of a Steammon. When HP reaches zero, the Steammon faints and becomes unable to battle.
- **attack (Atk):** Determines damage dealt by physical attacks.
- **special attack (SpA):** Determines damage dealt by special attacks.
- **defense (Def):** Determines damage taken from physical attacks.
- **special defense (SpD):** Determines damage taken from special attacks.
- **speed (Spe):** Determines how quickly a Steammon acts.

2.7.4 Modifiers

Steammon stats are influenced by *modifiers*, which result from items and effects. The current value of a modifier is called its *stage*. Modifiers range from stage -6 to stage 6, with a default of stage 0. When a Steammon is switched out, all its modifiers are reset to stage 0. The current stage is used to lookup a *multiplier* for the stat.

2.7.5 Effects

A move that hits a Steammon might cause an *effect*, which causes some beneficial or harmful change to the target of the effect (which might be either the Steammon who used the move, or the Steammon who was hit by the move):

- **Stat modifier:** One of the target's stat modifiers changes to a new stage.
- **Recover HP:** The target recovers a percentage of its maximum HP.
- **Take recoil damage:** The target takes a percentage of the damage dealt by the move.
- **Take HP damage:** The target takes a percentage of its maximum HP as damage.
- **Heal status effects:** The target is healed from some status effects.
- **Restore PP:** All the target's moves regain some PP.
- **Status effect:** Inflicts a status effect, as described below.

2.7.6 Status

Status effects are lasting ailments with which a Steammon can be inflicted. A Steammon can be afflicted only with one effect at a time, and a Steammon that is currently afflicted is immune to being afflicted by another effect until the current one wears off. Fainted Steammon have all effects removed when they are revived.

- **Confused:** A Confused Steammon might attack itself instead of using the move chosen by the player. At the beginning of each turn, the Steammon has a chance of snapping out of confusion.
- **Paralyzed:** A Paralyzed Steammon has its speed lowered, and might fail to move during its turn.[3]
- **Poisoned:** A Poisoned Steammon takes extra damage at the end of each turn.
- **Burned:** A Burned Steammon takes extra damage at the end of each turn, and the damage of its physical attacks is reduced.
- **Asleep:** A Steammon that is Asleep is unable to move. At the beginning of each turn, the Steammon has a chance to wake up and move.
- **Frozen:** A Frozen Steammon is unable to move. At the beginning of each turn, it has a chance to defrost.

2.8 Items

The following items can be held in a team's inventory:

- `Ether`: Restores 5 PP to each attack of the target Steammon, but cannot cause PP to exceed their maximum.
- `MaxPotion`: Restores a Steammon back to full HP. This item cannot revive a fainted Steammon.
- `Revive`: Restores a fainted Steammon back to half of its maximum HP. Revive can be used only on fainted Steammon and is also the only item that may be used on fainted Steammon.
- `FullHeal`: Removes all status effects from the target Steammon.
- `XAttack`: Increases the active Steammon's Attack modifier (not Special Attack) by 1.
- `XDefense`: Increases the active Steammon's Defense modifier (not Special Defense) by 1.
- `XSpeed`: Increases the active Steammon's Speed modifier by 1.

[3]A better name for this effect might be Slowed, but Paralysis is the traditional name.

If a player wants to use an item, they must have at least one of that item in their inventory, and after use, they lose one copy of the item. If a player tries to use an item in a way that contradicts any of the restrictions above, it has no effect and the player still loses a copy of the item.

2.9 Additional Rules

For a more balanced game, the following rules are also in place:

- **Sleep Clause:** A team with a sleeping Steammon cannot have another of its Steammon be put to sleep by the opposing team. However, self-directed sleep moves (such as Rest) can still put a Steammon on the team to sleep.
- **Freeze Clause:** Only one Steammon on each team may be frozen at a time.

3 Provided Source Code

Many files are provided for this assignment, as listed in Figure 2. You may edit and/or create new files only in the `game` and `team` directories. You may also make edits to the compilation scripts in those two directories based on the new files you add.

You will find it helpful to read `shared/definitions.ml` and `shared/util.ml` now, before reading the rest of this writeup.

3.1 Randomness

The standard library provides `Random.int n`, which generates a random integer between `0` (inclusive) and `n` (exclusive).

3.2 Constants

Constants are defined in `shared/constants.ml`. Constant names begin with a lowercase `c`, followed by a descriptive name in all caps. For example, `cNUM_PICKS` specifies the number of picks that can be made during the draft phase of the game. (The reason for the `c` in front of all constants is that OCaml doesn't allow value names to begin an uppercase letter—only type constructors may begin with capitals.) For all constants involving chance, the value of the constant is the percent chance that the relevant event occurs. For instance, if `cWAKE_UP_CHANCE` is 10, then there is a 10% chance to wake up.

Your bot may assume that these constants will be the same during the tournament as during your own development and testing. But during grading of your game implementation, the course staff reserves the right to change the constants to determine whether your implementation behaves appropriately.

`game/initialization.mli`	Signature file for initializing moves and Steammon
`game/initialization.ml`	Implementation of initializing moves and Steammon
`game/game.mli`	Signature file for handling actions, time, rules, and the game state
`game/game.ml`	Stub file for actions, rules, and time
`game/netgraphics.mli`	Signature file for sending updates to the GUI client
`game/netgraphics.ml`	Implementation of sending updates to the GUI
`game/server.ml`	Starts the game server and deals with communication
`shared/connection.mli`	Signature file for connection helper module
`shared/connection.ml`	Implementation of connection helper module
`shared/constants.ml`	Definitions of game constants
`shared/definitions.ml`	Definitions of game datatypes
`shared/util.mli`	Signature file for calculations and general use helper functions
`shared/util.ml`	Calculations and helper functions
`shared/thread_pool.mli`	Signature file for thread pool helper module
`shared/thread_pool.ml`	Implementation of thread pool helper module
`team/team.ml`	Basic framework for a client to interact with a game server
`team/babybot.ml`	Stub for a team AI
`game/steammon.csv`	Steammon for the game
`game/moves.csv`	Moves for the Steammon

Figure 2: Provided source files

3.3 Architecture

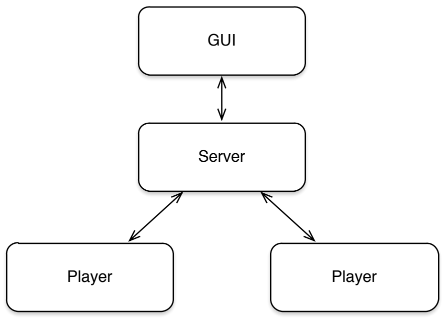

Figure 3: Architecture of *PokéJouki Emerald* implementation

The implementation is divided into a game *server*, a *GUI*, and two *players*. The game server is responsible for keeping track of the game state, applying the game rules, and so on. Players run as an entirely separate processes. Players can track whatever information they want. The GUI is responsible for displaying the game graphically.

3.3.1 Server

The `Server` module deals primarily with receiving network connections from the teams. It calls the `Game` module for all issues related to the game rules. You should avoid modifying the `Server` module, because that could make your bot incompatible with the server that will be used during the tournament.

At a high level, `server.ml` works as follows:

1. Calls `Netgraphics.init`, which accepts connects from GUI clients.
2. Waits until enough teams join the game.
3. Repeatedly:
   (a) Sends requests to players.
   (b) Receives actions from players.
   (c) Calls `Game.handle_step` to update the game state.
   (d) Sends graphics updates to the GUI.

You will need to think carefully about how you design and implement the game to meet the `Server` module's expectations.

The rate at which the server operates is controlled by constants documented in `constants.ml`. You might find it useful to tweak these constants while you are testing.

Update	Arguments	Meaning
`InitGraphics`	`red_team, blue_team`	Tell the GUI client to initialize the graphics in preparation for a new game starting and provides the names of both teams.
`UpdateSteammon`	`species, hp, maxhp, team`	Update (or add) Steammon by species to given team
`SetChosenSteammon`	`species`	Set the currently active Steammon (team looked up by species name)
`Move`	`move_result`	Display all the results of performing a move. Non-damaging moves (move with power = 0), as well as moves that missed or failed, should always be assigned `Regular` effectiveness.
`Item`	`item_name, effect_result, color, mon_name`	Display result of using an item on the Steammon with the name `mon_name` in the team specified by `color`.
`AdditionalEffects`	`effect_results`	Display results of additional effects that occur separately from moves. Examples of additional effects include poison damage and recovering from sleep.
`Message`	`msg`	Display a status message `msg` to the GUI console. This is not necessary for the game implementation, but you may find it useful for testing.
`SetFirstAttacker`	`team`	Sets which team moves first in the round. Must be called before the `Move` and `AdditionalEffects` messages are queued for the round.

Figure 4: Update messages that can be sent to the GUI

3.3.2 GUI

The GUI is provided in the `gui` directory. You should avoid modifying the GUI, because that could make your bot incompatible with the server that will be used during the tournament.

The GUI renders the game using Java and Swing. The game server (which you will implement) is responsible for sending updates to the GUI. The `Netgraphics` module, which is provided, will help with that. It contains functions to send graphical updates. The functions are specified in `netgraphics.ml`. To begin, the `init` function should called by the `Server` module. The `InitGraphics` update must be sent by itself, so the `Game` module should send this update directly by calling `Netgraphics.send_update`. After that, the `Game` module should call `Netgraphics.add_update`, which batches updates. A batch is sent when the `Server` module calls `sendUpdates`. Figure 4 summarizes the update messages that can be sent to the GUI client.

Your implementation should update the GUI as the game progresses. You may notice that the GUI refreshes only once every 1.5 seconds, independently of what has happened with the game state. This delay gives you time to observe messages in the GUI.

3.3.3 Player

A basic implementation of a player is provided in `team/babybot.ml`. That implementation relies upon `team/team.ml`, which contains code for the bot to connect to the server. You should avoid modifying `team/team.ml`, because that could make your bot incompatible with the server that will be used during the tournament. The player AI located in `babybot` is rudimentary at best—it chooses Steammon randomly, and it uses the first move it finds that has PP remaining.

3.3.4 Other Modules

Game Your design might require you to add or modify the type declarations or functions we have provided in the `Game` module. Such changes are allowed. When implementing `Game.handle_step`, please split the game logic into various helper functions for good style and readability.

State We strongly suggest that you have a `State` module in your final design, as the distinction between game rules and game state is significant. In other words, you should not put all of your code in the `Game` module.[4]

3.4 Communication

The server and players communicate by sending information over channels. The protocol for messages is defined by the type `command` in `definitions.ml`. There are five types of messages defined in the protocol:

- Control messages, which deal with starting and ending the game.
- Error messages, which are sent by the server when something goes wrong.
- Action messages, which players uses to indicate their intentions to the server.
- Request messages, which the server uses to query players for information.
- A `DoNothing` message.

3.5 Running the Game

You will need four command prompts to run the game:

1. **Game Server:** From `game/`, run `./build_game.sh` to build the game server. In this command prompt, run `./game`.

[4]In yet other words, we encourage use of the Model–View–Controller design pattern. `State` would be the Model, the GUI would be the View, and `Game` is the Controller. If you'd like to read more about this pattern, see the CS 2112 and CS 3152 course materials. As used here, the Model does not communicate with the View, because the View does not need to update every time the Model changes.

2. **GUI:** The GUI is already written for you in Java. It is distributed in a jar file called `gui/gui_client.jar`. The GUI needs the `data` folder for game assets. To run the GUI, just run the jar file (from the command line, `java -jar gui_client.jar`). Enter your Game Server connection information (by default it is localhost at port 10501), and press the `Connect` button. Always be sure to **start the GUI and press `Connect` before you start the teams.**

3. **Team 1:** From `team/` run `build_team.sh teamname` to build the bot in the file `bot/teamname.ml`. Next, run `./teamname localhost 10500`. This bot will become the Red player.

4. **Team 2:** Repeat the instructions above for Team 1 in a separate command prompt to cause a second team to join the game. This bot will become the Blue player.

Out of the box, you should be able to get the game server running and connect to the server with the GUI and `babybot`, but the server will throw an exception immediately after both bots connect.

4 Game Implementation

4.1 Initialization Phase

Already implemented for you: The file `initialization.ml` contains code for parsing of `moves.csv` and `steammon.csv`. That code creates hash tables for moves and for Steammon.

What you need to implement: After both players connect to the server, the server calls `init_game`, which you should implement. Your initialization code will need to

- use the code in `initialization.ml` to read files `moves.csv` and `steammon.csv` and obtain the information about the Steammon that will be in play for this game,
- send out a `TeamNameRequest` to each player and update the GUI with the team's names,
- select the player that will get first pick, and
- do any other initialization necessary for your design.

After initialization, the server enters its main loop as described in section 3.3.1. So the remaining implementation you do will mostly be related to `handle_step`. And if your design necessitates more initialization in `handle_step`, that's fine.

4.2 Draft Phase

Already implemented for you: Nothing.

What you need to implement: Each player should be given `cSTEAMMON_CREDITS` worth of credits with which to draft Steammon. At each round of drafting, a `PickRequest` should be sent to the player that is to pick, and nothing to the player that is to wait. Each player should choose a total of `cNUM_PICKS` Steammon. If a player tries to pick a Steammon whose cost exceeds the player's remaining credits, the game should give them any Steammon with the lowest cost. If that cost exceeds the player's remaining credits, set that player's credits to 0.

4.3 Inventory Phase

Already implemented for you: Nothing.

What you need to implement: The game should send `PickInventoryRequest` to both players. The players should respond with `PickInventory`. If the total cost of the inventory chosen by the player exceeds `cINITIAL_CASH`, then the player should instead be given the default inventory specified in `constants.ml`.

4.4 Battle Phase

Already implemented for you: Many functions in `util.ml`, which you would likely benefit from reading now.

What you need to implement: A `StarterRequest` should be sent to both teams, and the Battle Phase begins. During battle, each step of the game should proceed as follows:

1. Status effects that occur at the beginning of a turn should be processed.
2. The most recent `action` from each player should be resolved.
3. Status effects that occur at the end of a turn should be processed.
4. A `request` should be prepared for each player.
5. The condition of each team should be checked to determine whether there is a winner.

4.4.1 Implementing Status Effects

- **Asleep:** At the beginning of each turn, a sleeping Steammon has `cWAKE_UP_CHANCE` to wake up. A sleeping Steammon may not use moves.
- **Frozen:** At the beginning of each turn, a frozen Steammon has `cDEFROST_CHANCE` to thaw. A frozen Steammon may not use moves.

- **Confused:** A Confused Steammon has `cSELF_ATTACK_CHANCE` of attacking itself instead using the move chosen by the player. A Steammon has `cSNAP_OUT_OF_CONFUSION` chance to recover from confusion at the beginning of each turn. If a Steammon does attack itself, the move it chose as its action is ignored, and a move called AttackSelf is used instead. AttackSelf is a move that all Steammon share, hence it is not included in any Steammon's list of four moves. AttackSelf can always be used, regardless of its remaining PP.
- **Paralyzed:** When a Steammon is paralyzed, there is a `cPARALYSIS_CHANCE` chance that the Steammon will not move that turn. The speed of a paralyzed Steammon is divided by `cPARALYSIS_SLOW`.
- **Poisoned:** When a Steammon is poisoned, it takes `max_hp*cPOISON_DAMAGE` extra damage at the end of each turn, where `max_hp` is the Steammon's maximum HP.
- **Burned:** When a Steammon is burned, the damage of its physical attacks is multiplied by `cBURN_WEAKNESS`. It also takes `max_hp*cBURN_DAMAGE` extra damage at the end of each turn.

4.4.2 Implementing Actions

In response to an `ActionRequest`, the player should send either `SwitchSteammon`, `UseItem`, or `UseMove`.

- `SwitchSteammon`: The player should choose a non-fainted Steammon to become active. That Steammon should be moved to the front of the list within `team_data`. A Steammon that is switched out loses all modifiers.
- `UseItem`: A copy of the item should be removed from the player's inventory, and any effects from the item should be applied.
- `UseMove`: Resolving a move is a four-step process, described below:
  1. **Determine whether the move occurs.** If a Steammon attempts to use a move that has no PP remaining, the game should have the Steammon use a move called Struggle instead. All Steammon may Struggle, so that move is not included as one of a Steammon's four moves. There is no need to decrement the PP for this shared move.

     If a Steammon is frozen or asleep, the move fails. If a Steammon is confused or paralyzed, there is a chance that the move may fail. If a move does fail, no PP should be decremented.
  2. **Determine whether the move hits.** A move either *hits* or *misses*. The accuracy of a move is the chance that it hits. One PP should be decremented from the move regardless of whether the move hits or misses.

     If the power of a move is 0, then the move is *non-damaging*. Self-targeted non-damaging moves should never miss, but opponent-targeted non-damaging moves may hit or miss.

3. **Determine the damage done by the move.**

   Non-damaging moves should always do 0 damage. Non-damaging moves should ignore Steamtypes completely. Otherwise, the amount of damage done by a move is calculated by `calculate_damage` in `util.ml`. Appendix A.1 describes how that formula works.

4. **Determine any effects that result from the move.**

   If the move hits, determine whether its effects take place. If so, each effects should be processed in order.

4.4.3 Implementing Requests

Normally, the game should send an `ActionRequest` to each player. But when a Steammon faints, the game should send a `StarterRequest` to the player whose Steammon fainted, and no request to the other player. This gives the former player an opportunity to chose a new active Steammon before the next round begins.

4.4.4 Handling Fainted Steammon

There are two special cases that you will need to consider if a Steammon faints before all actions are resolved:

- **A Steammon faints before its team's action was resolved.** This could occur if the opposing Steammon was faster and caused the Steammon to faint before it had time to act. In this case, the action should be ignored.
- **A Steammon faints before the opposing team's action was resolved.** This could occur if the Steammon is faster and fainted due to a self-damaging or recoil move. In this case, the opponent's action should be ignored only if it is a move that was directed at the now-fainted Steammon; otherwise, it should be resolved normally.

4.5 Winner

A team wins when all the Steammon in the opposing team have fainted. If all the Steammon in both teams have fainted, the outcome is a tie.

4.6 Error Handling

We leave error handling unspecified, except for errors documented above, and except for missing messages and invalid messages, which we now describe.

If a player sends no message within `cUPDATE_TIME`, the server will treat the player's message as *missing*. The action passed into `Game.handle_step` will be `DoNothing`. The game logic should continue as follows:

- For a `TeamNameRequest`, the game should deem the player's team name to be "Red" or "Blue", depending on its team color.
- For a `PickRequest`, the game should pick the least expensive Steammon and deduct credits appropriately, setting credits to 0 if the Steammon costs more than the player has.
- For a `PickInventoryRequest`, the game should pick a default inventory based on the default item counts in `constants.ml`.
- For a `StarterRequest`, the game should pick an arbitrary non-fainted Steammon for the player.
- for an `ActionRequest`, the game should not perform any action for the player, but the player can still be affected by the opposing player's actions.

If the game receives an *invalid* message from a player, the game should treat it as if the message were missing. Here are some examples of invalid messages:

- Attempting to use an item that is not in the team's inventory.
- Sending anything other than `PickSteammon` in response to a `PickRequest`.
- Sending anything other than `SelectStarter` in response to a `StarterRequest`.
- Using a `UseItem` action that has a target that is not a Steammon in the team.
- Using a `UseMove` action that is not a valid move.

In addition, the player should never send any message other than `Action` to the server.

5 Bot Implementation

Use `babybot` as a starting point for creating your own bot. The function `handle_request` takes in the color representing the bot's team and a request from the game, and it should return the action that the bot wishes to take.

Your bot should be able to play in a variety of matches that have different Steammon and moves. In other words, your bot cannot rely on specific Steammon being available, on those Steammon having particular moves, or on the moves having specific attributes. Instead, you bot should analyze what is available and make choices based on that analysis. When picking a team, one approach might be to fold over all the available Steammon, assign a rating to each based on its stats and the attributes of its moves, then pick the Steammon with the highest ratings. When choosing moves during battle, one approach might be to examine the current state of the game, fold over the moves currently available to the Steammon to see which would be most advantageous, then choose the best move.[5]

[5] The CS 3152 course materials describe a Sense–Think–Act approach to AI design that you might find inspiring.

During grading and during the Tournament, your bot will be compiled and run in environments with different game implementations than your own. To ensure that the course staff can compile your bot, **do not make your bot implementation dependent on any files outside your `team/` directory.** The one exception is that you may assume the files in the `shared/` directory will be available—though you may not make any changes to those files. Your bot implementation particularly should not assume anything about files in the `game/` directory.

When submitting, the filename of the bot you want to have graded must be `team/bot.ml`, so we can easily distinguish it from other bots that you may have written.

Comments

[written,ungraded] In your file of written solutions, please include any comments you have about the problem set or about your solutions. This would be a good place to list any known problems with your submission that you weren't able to fix, or to give us general feedback about how to improve the problem set.

Include a statement of what work in this problem set was done by which team member. The ideal case is that each of you contributed to every problem. But (especially since this problem is ungraded) please be honest about how you divided the work.

A Formulas

We have primarily removed the need for you to implement formulas such as damage calculation; you can find the implementations in `util.ml`. We leave them here for the curious.

A.1 Calculating Damage

The formula for attack damage is:

$$\left(\frac{84}{100} \times \frac{\text{Attack}}{\text{Defense}} \times \text{Power} + 2\right) \times \text{Multiplier}$$

- **Attack:** For physical attacks, the attacker's Attack stat. For special attacks, the attacker's Special Attack stat.
- **Defense:** For physical attacks, the defender's Defense stat. For special attacks, the defender's Special Defense stat.
- **Power:** The power rating for the attack
- **Multiplier:** Product of various multipliers that affect the damage of attacks, as described next.

The multiplier is calculated with this formula:

$$\text{STAB} \times \text{SteamtypeMultiplier} \times \text{BurnWeakness} \times \left(\frac{\text{Random}}{100}\right)$$

- **STAB:** If a Steammon uses an attack that has the same element as one of its Steamtypes, STAB (Same Type Attack Bonus) is `cSTAB_BONUS` otherwise, it is 1.
- **SteamtypeMultiplier:** This is determined by the attacking Steammon's attack type and the defending Steammon's types. Calculated by `calculate_type_matchup` in `util.ml`.
- **BurnWeakness:** This is `cBURN_WEAKNESS` if the attacking Steammon is Burned and the attack is a physical attack, 1 otherwise.
- **Random:** A random number in the range [`cMIN_DAMAGE_RANGE`, 100].

B Differences Between *PokéJouki Emerald* and Pokémon

Pokémon aficionados might notice that *PokéJouki Emerald* differs in various ways from the official game. Here is an incomplete list of differences:

Abilities do not exist. Therefore, picking Shedinja is not recommended.

Many moves are unavailable. When writing your bot, don't expect your favorite strategies from the Pokemon games to work.

Hold items do not exist. These restriction is for your sanity, as well as ours.

Critical hits do not occur. This is to help prevent sheer luck from changing the tide of battle.

Confusion works like other status effects. For the sake of simplicity, Confusion does not wear off after a Steammon is switched out, nor can a Steammon have another status when Confused.

Sleep mechanics are different. Instead of sleep lasting 1-3 turns, Steammon have probability `cWAKE_UP_CHANCE` to wake up every turn.

Struggle is triggered differently. Instead of being used when all of the moves of a Steammon run out of PP, Struggle is performed when any move with no PP left is chosen.

Some move effects have been simplified. For example, Rest inflicts regular sleep, and Heal Bell heals statuses for only the user.

Steammons' stats are hard-coded. We assume that, for each Steammon, its level is 100, IVs are all 31, and EVs are all 0.

There are no accuracy or evasion modifiers. The chance of a move hitting is based solely on its own accuracy value.