15-150 Fall 2015
Homework 11

Out: Wednesday, 18 November 2015
Due: Tuesday, 1 December 2015 at 23:59 EDT

1 Introduction

In this homework you will write a game and a game player using the techniques you have been learning in class. There are two major parts: writing the representation of the game, and writing sequential and parallel versions of the popular alpha-beta pruning algorithm. This will also continue to test your ability to use and understand the module system.

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository, as usual.

1.2 Submitting The Homework Assignment

Submissions will be handled through Autolab, at `https://autolab.cs.cmu.edu`

In preparation for submission, your `hw/11` directory should contain a file named exactly `hw11.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/11` directory (that contains a `code` folder and a file `hw11.pdf`). This should produce a file `hw11.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw11.tar` file via the "Handin your work" link.

The Autolab handin script does some basic checks on your submission: making sure that the file names are correct; making sure that no files are missing; making sure that your code compiles cleanly. Note that the handin script is *not* a grading script—a timely submission that passes the handin script will be graded, but will not necessarily receive full credit. You can view the results of the handin script by clicking the number corresponding to the "check" section of your latest handin on the "Handin History" page. **If this number is** 0.0**, your submission failed the check script; if it is** 1.0**, it passed.**

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

All the code that you want to have graded for this assignment should be contained in `game/ants.sml`, `lib/game/runants.sml`, `lib/game/boards.sml`, `game/alphabeta.sml`, and `game/jamboree.sml` , and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Due Date

This assignment is due on Tuesday, 1 December 2015 at 23:59 EDT. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester.

1.4 Methodology

You must use the five step methodology discussed in class for writing functions, for **every** function you write in this assignment. Recall the five step methodology:

1. In the first line of comments, specify the type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments to be passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns when applied to an argument that satisfies the assumptions in `REQUIRES`).
4. Implement the function.
5. For this assignment you do not need to test each function individually. See the specific sections for testing guidelines.

For example, for the factorial function presented in lecture:

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n! *)
fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)
val 1 = fact 0
val 6 = fact 3
val 720 = fact 6
```

2 Better Sequences

For the purpose of this assignment, it is useful to extend the sequence signature to include some more involved functions. In `game/betterseq.sml`, implement the following functions:

Task 2.1 (10 pts). Sometimes it is useful to be able to update values in a sequence. Implement the function

```
update : 'a -> 'a seq -> int -> 'a seq
```

Such that `update x S i` evaluates to a sequence identical to S, except at index i, where its value is x.

Task 2.2 (10 pts). When performing maps on sequences, we only have access to the individual elements that we are mapping. This restricts some of our functionality. It would be useful if we could have a map function that allows us to access both the element and its index. Implement the function

```
mapIdx : ('a * int -> 'b) -> 'a seq -> 'b seq
```

Such that `mapIdx f S` evaluates to a sequence `S'` which is the result of applying `f` to every (element, index) pair in `S`.

For example if `S = <"150", "is", "cool">`,
`mapIdx f S` would evaluate to `<f("150",0), f("is", 1), f("cool", 2)>`

Task 2.3 (10 pts). When given an option sequence, it is useful to be able to filter out the `NONE`'s as well as strip away the `SOME`'s from the actual values. Implement the function

```
extractSomes : 'a option seq -> 'a seq
```

Such that `extractSomes S` removes all of the `NONE` values from `S`, and strips off the `SOME` values from the remaining elements.

For example if `S = <SOME(5), NONE, NONE, SOME(3), SOME(4), NONE>`
`extractSomes S` would evaluate to `<5,3,4>`

3 Records

3.1 Overview

A record is a keyed data structure built into SML that allows us to create unordered tuples with named fields. Often times using a normal tuple can be confusing when it has many members. It can be hard to keep track of which member represents which value. Records solve this problem by adding names to each of their members. Importantly, each field has a set type which the value at the field must have.

3.2 They're Still Functional

Although records may sound much like a dictionary in Python, or an object in Java, they have one very important difference. Unlike dictionaries and objects, records are immutable data structures. This means that once a record has been created, its values stay as they are forever. We must create a completely new record everytime we would like to change the value of a field. Since records are immutable, they are okay to use in a functional setting as we always know the type of each field, and the value of any field cannot change due to side effects.

3.3 Creating a Record and Accessing Fields

To create a record we use a special syntax we have not used before. The following will create a record with two fields, cat and dog, with the values "meow" and "woof" at each field respectively: `{cat = "meow", dog = "woof"}`. Suppose we have bound that record to the variable R. We can get the value at the dog field by doing the following: `#dog(R)`. Likewise, the value at the cat field can be retrieved with: `#cat(R)`.

3.4 Pattern Matching with Records

Like most other data structures, we can pattern match on Records. The syntax is the same as that to create a record, except in place of a value for each field you put a pattern: `case R of {cat = x, dog = y}` binds the value of the cat field to x and the value of the dog field to y. Therefore x has the value "meow" and y has the value "woof". Since records are unordered, it is important that we include the field names when creating or pattern matching a record.

4 Description of Game

4.1 Introduction

Pokemon: Lambda Version is a functional approach to the classic Nintendo game, Pokemon. You will be able to choose your pokemon, choose their attacks, and battle against an opponent, all with more functions than ever before.

4.2 Game Constants

When creating games it can be useful to parameterize certain constants about the game environment. We have achieved this by defining a signature, which we have called `CONSTS`, that contains all of the relevant but mutable game constants. These include the level of all pokemon, relative damage between types, and the number of pokemon. You can view the full signature in `consts.sig`.

4.3 Battle

Each trainer will have one pokemon out at a time. They take turns making moves until the opponent has been defeated. A move can either be an attack or the trainer can choose to switch out their current pokemon for a different one that has not yet fainted. A pokemon has fainted when its health drops to 0. If the trainer chooses to attack, they must choose one of their current pokemon's attacks.

4.4 Victory

A trainer wins when all their opponent's pokemon have 0 health.

4.5 Types

Every pokemon and every attack has a type. They are one of Normal, Fire, Water, Electric, Grass, Ice, Fighting, Poison, Ground, Flying, Psychic, Bug, Rock, Ghost, or Dragon. Certain types are stronger against certain other types (Water against Fire, or Electric against Flying). There are four levels of effectiveness when it comes to attacking a pokemon with a move of a certain type: Super Effective, Effective, Not Very Effective, and No Effect. When calculating the modifier that applies to an attack, the relevant types are the type of the attack itself and the type of the defending pokemon. In addition there is a STAB (Same Type Attack Bonus) modifier applied when an attacker uses an attack with the same type as its own. For example, if a fire type pokemon uses a fire type move, it will do bonus damage.

4.6 Attacking

Attacks have four attributes: `abiltype` - the type of the move, `dmg` - the base damage done by the attack, `self_effect` - the stats effects the attack has on the attacker, `opp_effect` -

the stats effects the attack has on the defender. Attacks are defined with the 'ability' type as records with the aforementioned fields. Using the information of the two pokemon involved in the attack (the attacker and the defender) and the attack being used, damage is calculated and applied to the defending pokemon.

4.7 Switching

Another valid move is to switch out to a different pokemon. A player can only switch to a pokemon that has not fainted. If the current pokemon has fainted, the player can only perform a switch move, they will be unable to attack. When switching pokemon, all stats are reset to their original values. Also, since *Pokemon : Lambda Version* is a highly competitive game, there are permanent exit hazards on the field. This means that anytime you switch out a pokemon, the pokemon that you switched out takes a small amount of damage.

5 Implementation of Types

5.1 State

For our implementation, we represent the state of the game as the parties of the two trainers and the trainer whose turn it is:

```
type state = (party * party * trainer)
(* Maxie's party, Minnie's party, the current trainer *)
```

5.2 Party

The parties are defined by a sequence of pokemon and an index, corresponding to the current pokemon in play:

```
type party = (pokemon seq * int)
```

5.3 Trainer

The trainer is just the current player:

```
datatype player = Minnie | Maxie
type trainer = player
```

In `game/pokedex.sml` we define some of the record types:

5.4 Stats

The stats is a record containing the attack and defense of a pokemon.

```
type stats = {atk: int, def: int}
```

5.5 Poketype

The poketype is an integer corresponding to what type the pokemon is.

```
type poketype = int
   Types: 0 - normal
          1 - fire
          2 - water
          3 - electric
          4 - grass
          5 - ice
```

```
6 - fighting
7 - poison
8 - ground
9 - flying
10 - psychic
11 - bug
12 - rock
13 - ghost
14 - dragon
```

5.6 Ability

An ability is a record containing the type of the move (abiltype), how much damage it does (dmg), the stat change it should perform on the attacking pokemon (self_effect, stored in a stats record) and the stat change it should perform on the defending pokemon (opp_effect, again stored in a stats record).
The abilities type is just a sequence of ability.

```
type ability = {abiltype: poketype, dmg: int, self_effect: stats, opp_effect: stats}
type abilities = ability Seq.seq
```

5.7 Pokemon

The pokemon type is a record containing the name, the health (hp) of the pokemon, the effects, which is a stats record corresponding to the increase/decrease in stats that the pokemon is currently under, the base_stats, which is a stats record containing the pokemons original stats, the poketype, the type of the pokemon, and finally the moves, which is the abilities of the pokemon.

```
type pokemon = {name: string, hp: int, effects: stats, base_stats: stats,
                  poketype: poketype, moves: abilities}
```

5.8 Move

We define a move as either an Attack of an ability, or a Switch of an int where the int is the index of the pokemon to switch to in the hand.

```
datatype move = Attack of ability | Switch of int
```

5.9 Constants

Lastly, our pokemon functor takes in a `Options : CONSTS` structure as input, which holds some of the tweakable constants of our game:

```
signature CONSTS =
sig
(* The damage multiplier for normally
 * effective attacks
 *)
val effective : real
(* The damage multiplier for
 * not very effective attacks
 *)
val not_very_effective : real
(* The damage multiplier for super
 * effective attacks
 *)
val super_effective : real
(* The damage multiplier for
 * attacks with no effect
 *)
val no_effect : real
(* The damage multiplier for
 * using an attack with the
 * same type as the attacking
 * pokemon
 *)
val stab_modifier : real
(* The level of all the pokemon
 * which will effect damage calculation
 *)
val level : int
(* The amount of damage done to a pokemon
 * when it gets switched out
 *)
val switch_damage : int
end
```

6 Implementing the Game

We have already implemented a substantial part of the game for you, your job is to finish the rest.

6.1 Attacking

We have implemented the following functions for you:

- `get_type_modifier : poketype -> poketype -> real`

  Which takes an attacking type and a defending type and calculates the damage multiplier.

- `modifier : bool -> poketype -> poketype -> real`

  Which takes a boolean indicating whether the attack is STAB or not, an attacking type and defending type and calculates the damage multiplier.

- `damage : int -> int -> int -> real -> real`

  Which takes the attack stat of the attacking pokemon, the defense stat of the defending pokemon, the base damage of the move, the damage multiplier, and computes how much damage the move does.

Task 6.1 (5 pts). Implement the function

`apply_effect : pokemon * stats -> pokemon`

Which takes a pokemon and a new set of effects, and changes the pokemon's effect field to be the sum of the old effects and the new effects.

Task 6.2 (5 pts). Implement the function

`apply_damage : pokemon * int -> pokemon`

Which takes a pokemon and an int representing damage from a move, and updates the pokemon's hp field accordingly. Note that a pokemon can never have less than 0 hp.

Task 6.3 (6 pts). Implement the function

`apply_attack : pokemon -> pokemon -> ability -> (pokemon * pokemon)`

Which takes an attacking pokemon, a defending pokemon, and the ability that the attacking pokemon just used, and computes the resulting (attacking pokemon, defending pokemon) tuple. Don't forget to update both the attacking pokemon and defending pokemon's stats and effects accordingly.

6.2 Switching

We have implemented the function

```
calc_switch_dmg : pokemon -> pokemon
```

That takes the pokemon you switched out and applies the switch damage to it.

Task 6.4 (5 pts). Implement the function

```
reset_stats  : pokemon -> pokemon
```

That takes a pokemon and resets its stats by removing all effects.

6.3 Moves

Task 6.5 (12 pts). Implement the function

```
make_move : (state * move) -> state
```

Which takes a state and a move and computes the state as a result of applying that move. Make sure you consider both types of moves (Switch and Attack), as well as both possible states (Maxie's turn and Minnie's turn). Feel free to use any of the previous functions as well as any of the sequence functions you implemented in betterseq. (We have already opened the structure for you so you can just reference them as update, mapIdx and extractSomes).

Task 6.6 (9 pts). Implement the function

```
available_pokemon : pokemon seq -> int seq
```

Which takes a pokemon sequence and returns a sequence of indices to the pokemon that haven't fainted. You might find the `has_fainted` function useful.

Task 6.7 (12 pts). Implement the function

```
moves : state -> move seq
```

Which takes a state and generates a sequence of all possible moves at that state.

6.4 End of game

We represent the status of the game using the following datatype:

```
datatype outcome = Winner of player
datatype status = Over of outcome | In_play
```

Task 6.8 (5 pts). Implement the function

`status : state -> status`

Which takes the game state and returns the status.

6.5 Building the AI

Task 6.9 (6 pts). Implement `estimate : state -> Est.est`.

`estimate` returns an estimated score for a given state. You will receive full credit as long as `estimate` performs no worse than an estimator that takes into account only the total health of the pokemon for each player. Of course, a more refined evaluation algorithm will yield a more competent and challenging computer opponent! Your implementation of `estimate` should work on all valid game states, whether it is in play or it is a terminal state (leaf in the search tree). Remember that relatively lower scores should indicate Minnie is winning and relatively higher scores should indicate Maxie is winning. Your estimate should reflect this.

Please describe your implementation and the strategies you considered in a few sentences in a comment.

BONUS: If you would like to be entered in a potential tournament, copy and paste your estimator function (and just the estimator function) into the provided `estimator.txt` file. If you are not interested then just leave that file blank.

6.6 Playing the Game

In `runpokemon.sml` we have defined a bunch of structures using the Referee functor. The Referee functor takes two players and returns a structure that can officiate a game between the two players. Each player takes a specific game instance which we define at the top of `runpokemon.sml`. In order to play against one of your game algorithms we can simply call that referee's `go` function. For example, to play against the MiniMax player we open the REPL, load `sources.cm` and then run `Pokemon_HvMM.go()`.

If you choose to play human against one of the algorithms you will be able to make moves using the following syntax:

```
-S <pokemon_#>
-A <move_#>
```

Entering the string: `S i` where i is a number will switch to the ith pokemon. Similarly entering the string: `A i` where i is a number will use the current pokemon's i'th move.

Testing: You do not need to test your individual functions. Playing the game counts as testing.

7 Sequence Thingies

Assume given a function

```
split : 'a seq -> 'a seq * 'a seq
```

such that whenever `length s > 1`, `split s` evaluates to a pair of sequences `(s1, s2)` with `append(s1, s2) = s`, and the lengths of `s1` and `s2` differ by at most 1.

Consider the `reduce` function given by:

```
fun reduce g z s =
   case (length s) of
      0 => z
    | 1 => g(nth 0 s, z)
    | _ => let
               val (s1, s2) = split s
           in
               g(reduce g z s1, reduce g z s2)
           end
```

Task 7.1 (10 pts). What is the value of `reduce (op +) 5 s` when the value of `s` is the all-zero integer sequence $\langle 0, 0, \ldots, 0 \rangle$ of length 2^n? Prove it.

Task 7.2 (10 pts). Let $W(n)$ be the work, and $S(n)$ be the span, for the expression `reduce g z s`, when `g` is a constant-time function value, `z` is a value, and `s` is a sequence value of length 2^n. Give recurrence relations for $W(n)$ and $S(n)$, and derive asymptotic bounds. (You can appeal to the table of standard recurrences that was discussed in class and notes.) You can assume that the work and span for `split s`, `nth 0 s` and `length s` are $O(1)$ (so, constant time) when `s` is a sequence value.

Now consider the `mapreduce` function given by:

```
fun mapreduce f g z s =
   case (length s) of
      0 => z
    | 1 => g(f(nth 0 s), z)
    | _ => let
               val (s1, s2) = split s
           in
               g(mapreduce f g z s1, mapreduce f g z s2)
           end
```

Task 7.3 (15 pts). Prove by induction on the length of `s` that:

```
mapreduce f g z s = reduce g z (map f s)
```

State clearly any basic properties of the `map` and `reduce` functions on sequences that are needed to justify your proof.

NOTE: This proof needs to be done by induction on sequence length! Not by structure induction or any other method! In fact you shouldn't expect to be able to use structural induction here because we don't give any information about the implementation of sequences — they could be lists, or binary trees, or shrubs — and we need a proof that makes sense no matter how sequence values are represented.

8 Alphabeta Pruning

In class and in lab we gave an explanation of the general ideas behind $\alpha\beta$ pruning, based on computing an iterative sequence of approximations until reaching a threshold. Implement $\alpha\beta$ in ML (along the lines of our explanation) by filling in the missing code in the template that we supply.

Task 8.1 (35 pts). Define the following functions:

```
F2 : int -> alphabeta -> state -> (bound * move option)
G2 : int -> alphabeta -> state -> (bound * move option)
next_move : state -> move
```

where we use the type abbreviation

```
type bound = NEGINF | est | POSINF
type alphabeta = bound * bound
```

`next_move` is responsible for taking a state which you can assume is `In_play` and gets the best possible move for the current player.

`F2` and `G2` are then just helpers for the `next_move` function.

HINT 1: `F2, G2` should be mutually recursive, but unlike `F, G` they should operate sequentially on the states reachable in one move.

HINT 2: It might be helper to define a helper function within F2 and G2 that do most of the 'hard' work.

You can use the function `toList : 'a seq -> 'a list` to convert a sequence value into a list value. The function `compareAB` will probably also be extremely helpful in implementing the two alphabeta functions.

9 Jamboree

Yet another variant on $\alpha\beta$ and minimax is the so-called Jamboree algorithm. The idea behind jamboree is that we can combine both the pruning of alphabeta, and the parallelism of minimax. To do this we take some state `s` for which there are at least 2 possible moves, and we split the move sequence into two subsequences `(M1, M2)`, where the ratio of their lengths is some parameterized value, `prune_percentage`. Given initial alphabeta bounds of (α, β) we apply the alphabeta algorithm to the first sequence of moves, `M1` and get a new alphabeta bound of (α', β'). We then run jamboree in parallel on the second sequence of moves, `M2`, using (α', β') as the initial alphabeta bounds for each subsequent jamboree call. We then take the result of each recursive jamboree call and as Maxie we take the max of them and as Minnie we take the min of them.

Task 9.1 (35 pts). Define the following functions:

```
splitMoves : state -> (move seq * move seq)
Fjam : int -> alphabeta -> state -> (bound * move option)
Gjam : int -> alphabeta -> state -> (bound * move option)
next_move : state -> move
```

which use the jamboree algorithm to determine the best possible move from a state.

HINT 1: `Fjam` and `Gjam` should look fairly similar to `F2` and `G2`. Notice that there's no constraint on what the type of the inner function should be. It might be helpful to adjust the type of that function rather than the type of `Fjam` and `Gjam`.

For this one we also provide you with a helper function named `compareBounds` which will probably also be very helpful.