Homework 2. Eager fragment analyzer

Introduction

In this assignment you will write a simple pattern-matcher generator for DNA fragments, protein sequences, character strings, and the like. Your generator will take a pattern and produce a pattern-matcher. This is akin to the POSIX `regcomp` regular expression compiler for character strings. However, unlike `regcomp` your generator will work on arbitrary sequences (not just character strings), and it will produce a function that can be called directly, not data to be passed to a separate `regexec` interpreter.

It should be said right up front that this assignment is just a toy: real pattern matchers are more complicated than this, and don't rely solely on the simple kinds of backtracking used here.

The key notion of this assignment is that of a matcher. A *matcher* is a function that inspects a given fragment to find a match for a prefix that corresponds to a pattern, and then checks whether the match is acceptable by testing whether a given acceptor succeeds on the corresponding suffix. For example, a matcher for the pattern `Or [Frag [C;A]; Frag [A]]` will succeed only on a fragment whose first two symbols are `C` followed by `A` or whose first symbol is `A`, and whose remaining tail is acceptable to the acceptor.

An *acceptor* is a function that accepts a fragment as an argument by returning some value wrapped inside the `Some` constructor. The acceptor rejects the fragment by returning `None`. For example, the acceptor `(function | (G::tail) -> Some tail | _ -> None)` accepts only fragments beginning with the symbol `G`. Such an acceptor would cause the example matcher to succeed on `[A;G]` but fail on `[C;A;T]`.

As you can see by mentally executing the `[A;G]` example, matchers sometimes need to try multiple alternatives and to backtrack to a later alternative if an earlier one is a blind alley.

Definitions

nucleotide
: one of `A`, `C`, `G`, and `T`, which stand for adenine, thymine, cytosine, and guanine, respectively. Please see Richard B. Hallick's Introduction to DNA Structure for more about nucleotides in biochemistry.

symbol
: a value of an arbitrary type. This is a generalization of the notion of nucleotides. Most of the examples below use nucleotides, but your functions should work with fragments containing any sets of symbols.

fragment
: a list of symbols.

acceptor
: a function with one argument: a fragment *frag*. If the fragment is not acceptable, it returns `None`; otherwise it returns `Some` *x* for some value *x*. Typically *x* is some suffix of *frag* (possibly *frag* itself), but this is not required.

matcher
: a function with two arguments: a fragment *frag* and an acceptor *accept*. A matcher matches a prefix of *frag* such that *accept* accepts the corresponding suffix. If there is such a match, the matcher returns whatever *accept* returns; otherwise it returns `None`.

pattern
: an object that represents a set of fragments. It is said to *match* each such fragment. It takes one of the following forms:

  `Frag [`*symbol…*`]`

matches the list of symbols `[`*symbol*…`]`. For example, `Frag [A;C;T]` matches `[A;C;T]`, and `Frag []` matches `[]`.

`Junk` *k*
: matches up to *k* symbols. *k* must be a nonnegative integer. If there are several possible matches, it uses the shortest match by default, the longest match if nested somewhere inside an `Eager` pattern. For example, `Junk 0` matches only the empty fragment `[]`, and `Junk 2` matches `[]`, `[A]`, `[C]`, `[G]`, `[T]`, `[A;A]`, `[A;C]`, `[A;G]`, `[A;T]`, `[C;A]`, `[C;C]`, `[C;G]`, `[C;T]`, `[G;A]`, `[G;C]`, `[G;G]`, `[G;T]`, `[T;A]`, `[T;C]`, `[T;G]`, and `[T;T]`.

`Or [`*pat*…`]`
: matches any fragment that any pattern *pat* matches. If more than one *pat* matches, it uses the first *pat* that matches. For example, `Or []` does not match any fragment, and `Or [Frag [A]; Frag [G]; Frag [T]]` matches `[A]`, `[G]`, and `[T]`.

`List [`*pat*…`]`
: matches any concatenation of fragments that are matched by each *pat*, respectively. If there are several possible matches, `List [`pat_1`;` pat_2`; …]` uses the first match for pat_1 that is acceptable to `List [`pat_2`; …]`. For example, `List []`, which has zero patterns, matches only the empty fragment `[]`, and `List [Frag [A]; Junk 1; Or [Frag [A]; Frag [T]]]` matches `[A;A]`, `[A;T]`, `[A;A;A]`, `[A;A;T]`, `[A;C;A]`, `[A;C;T]`, `[A;G;A]`, `[A;G;T]`, `[A;T;A]`, and `[A;T;T]`.

`Closure` *pat*
: matches any concatenation of nonnull fragments that are each matched by *pat*. If the matched concatenation is of two or more fragments, this pattern is equivalent to `List [`*pat*`; Closure` *pat*`]`. If there are several possible concatenations, it uses the concatenation of the least number of lazily-matched fragments by default, or the concatenation of the most number of eagerly-matched fragments if nested somewhere inside an `Eager` pattern. For example, `Closure (Or [Frag [G]; [Frag [A;T]]]))` matches `[]`, `[G]`, `[A;T]`, `[G;G]`, `[G;A;T]`, `[A;T;G]`, `[A;T;A;T]`, …. Here another example, this one with duplicates: `Closure (Or [Frag [A]; Frag [A;A;A])` matches `[]`, `[A]`, `[A;A;A]`, `[A;A]`, `[A;A;A;A]`, `[A;A;A;A]`, `[A;A;A;A;A;A]`, `[A;A;A]`, `[A;A;A;A;A]`, ….

`Eager` *pat*
: matches anything that *pat* does, except it does so eagerly rather than using the default of lazy matching. This affects the behavior of any `Junk` and `Closure` subpatterns when they have multiple possible matches, as described above.

Assignment

Write a function `make_matcher` *pat* that returns a matcher for the pattern *pat*. When applied to a fragment *frag* and an acceptor *accept*, the matcher must return the first acceptable match of a prefix of *frag*, using the definition of "first" given by the pattern rules described above; this is not necessarily the shortest nor the longest acceptable match. A match is considered to be acceptable if *accept* succeeds when given the suffix fragment that immediately follows the matching prefix. When this happens, the matcher returns whatever the acceptor returned. If no acceptable match is found, the matcher returns `None`.

Also, write five good test cases for your `make_matcher` function. These test cases should all be in the style of the test cases given below, but should cover different problem areas. Your test cases should be named `test_1` through `test_5` (note the underscores; this distinguishes your test cases from the standard ones given below). Your test cases should test several patterns, each of which is compiled into its own matcher. At least three of your test cases should involve fragments that are not DNA fragments.

Your code should be free of side effects and should avoid using unnecessary storage. Also, the matchers that `make_matcher` returns should use only the following OCaml primitives:

- integer arithmetic
- variable references
- function calls

- constructors (e.g., `List`, `Or`)

along with the following kinds of expression:

- `::`
- `==`
- `=`
- `[]`
- `fun`
- `function`
- `if`
- `let`
- `match`

Your implementation of `make_matcher` *pat* should generate a matcher that does not do any pattern-matching on *pat* itself; all the pattern-matching on *pat* should be done by `make_matcher` before `make_matcher` returns. For example, `make_matcher (Frag [])` should return a matcher that acts like the `match_empty` function defined in the test cases below; the matcher itself should not test whether the `Frag` constructor's argument is empty.

Submit

Submit two files via CourseWeb. The file `hw2.ml` should define `make_matcher` along with any auxiliary types and functions needed to define `make_matcher`. The file `hw2test.ml` should contain your test cases. Please do not put your name, student ID, or other personally identifying information in your files.

Sample test cases

```
(* Fragments.  *)

let frag0 = []

let frag1 = [A;T;G;C;T;A]

(* OCaml does not care about the newlines in the definition of frag2.
   From OCaml's point of view, they are merely extra white space that
   is ignored.  The newlines are present only to help humans understand
   how the patterns defined below are matched against frag2.  *)
let frag2 = [C;C;C;G;A;T;A;A;A;A;A;A;G;T;G;T;C;G;T;
             A;
             A;G;T;A;T;A;T;G;G;A;T;A;
             T;A;
             A;G;T;A;T;A;T;G;G;A;T;A;
             C;G;A;T;C;C;C;T;C;G;A;T;C;T;A]

let frag3 = [A;G;A;G;A;G]

(* Patterns.  *)

let pat1 = Frag [A;T;G;C]
let pat2 = Or [Frag [A;G;T;A;T;A;T;G;G;A;T;A];
               Frag [G;T;A;G;G;C;C;G;T];
               Frag [C;C;C;G;A;T;A;A;A;A;
                     A;A;G;T;G;T;C;G;T];
               List [Frag [C;G;A;T;C;C;C];
                     Junk 1;
                     Frag [C;G;A;T;C;T;A]]]
let pat3 = List [pat2; Junk 2]
let pat4 = Closure pat3
let pat5 = Closure (Frag [A])
let pat6 = Closure (Frag [A;G])
let pat7 = Eager (List [pat5; pat6])
let pat8 = List [pat5; pat6]

(* Matchers.  *)
let matcher1 = make_matcher pat1
let matcher2 = make_matcher pat2
let matcher3 = make_matcher pat3
let matcher4 = make_matcher pat4
let matcher5 = make_matcher pat5
let matcher6 = make_matcher pat6
let matcher7 = make_matcher pat7
let matcher8 = make_matcher pat8

(* Acceptors.  *)

(* Always fail, i.e., never accept a match.  *)
let accept_none x = None

(* Always succeed.  This acceptor returns the suffix
   containing all the symbols that were not matched.
   It causes the matcher to return the unmatched suffix.  *)
let accept_all x = Some x

(* Accept only the empty fragment.  *)
let accept_empty = function
  | [] -> Some []
  | _ -> None


(* Test cases.  These should all be true.  *)

let test1 = matcher1 frag0 accept_all = None
let test2 = matcher1 frag1 accept_none = None

(* A match must always match an entire prefix of a fragment.
   So, even though matcher1 finds a match in frag1,
   it does not find the match in A::frag1.  *)
let test3 = matcher1 frag1 accept_all = Some [T;A]
let test4 = matcher1 (A::frag1) accept_all = None

let test4 = matcher2 frag1 accept_all = None
let test5 =
  (matcher2 frag2 accept_all
   = Some [A;
           A;G;T;A;T;A;T;G;G;A;T;A;
           T;A;
           A;G;T;A;T;A;T;G;G;A;T;A;
           C;G;A;T;C;C;C;T;C;G;A;T;C;T;A])

(* These matcher calls match the same prefix,
   so they return unmatched suffixes that are ==.  *)
let test6 =
  match (matcher2 frag2 accept_all,
         matcher3 frag2 accept_all)
  with
  | (Some fraga, Some fragb) -> fraga == fragb
  | _ -> false

(* matcher4 is lazy: it matches the empty fragment first,
   but you can force it to backtrack by insisting on progress.  *)
let test7 =
  match matcher4 frag2 accept_all with
  | Some frag -> frag == frag2
  | _ -> false
let test8 =
  match (matcher2 frag2 accept_all,
         matcher4 frag2
           (fun frag ->
              if frag == frag2 then None else Some frag))
  with
  | (Some fraga, Some fragb) -> fraga == fragb
  | _ -> false
let test9 = matcher4 frag2 accept_empty = Some []

let test10 = matcher5 frag3 accept_all = Some frag3
let test11 = matcher6 frag3 accept_all = Some frag3
let test12 = matcher7 frag3 accept_all = Some [G; A; G; A; G]
let test13 = matcher8 frag3 accept_all = Some frag3
```

Sample use of test cases

If you put the sample test cases into a file `hw2sample.ml`, you should be able to use it as follows to test your `hw2.ml` solution on the SEASnet implementation of OCaml. Similarly, the command `#use "hw2test.ml";;` should run your own test cases on your solution.

```
$ ocaml
        Objective Caml version 3.09.2

# #use "hw2.ml";;
…
type nucleotide = A | C | G | T
…
val make_matcher :
  '_a pattern -> '_a list -> ('_a list -> '_b option) -> '_b option = <fun>
…
# #use "hw2sample.ml";;
val frag0 : 'a list = []
val frag1 : nucleotide list = [A; T; G; C; T; A]
val frag2 : nucleotide list =
  [C; C; C; G; A; T; A; A; A; A; A; A; G; T; G; T; C; G; T; A; A; G; T; A; T;
   A; T; G; G; A; T; A; T; A; A; G; T; A; T; A; T; G; G; A; T; A; C; G; A; T;
   C; C; C; T; C; G; A; T; C; T; A]
val frag3 : nucleotide list = [A; G; A; G; A; G]
val pat1 : nucleotide pattern = Frag [A; T; G; C]
val pat2 : nucleotide pattern =
  Or
   [Frag [A; G; T; A; T; A; T; G; G; A; T; A];
    Frag [G; T; A; G; G; C; C; G; T];
    Frag [C; C; C; G; A; T; A; A; A; A; A; A; G; T; G; T; C; G; T];
    List [Frag [C; G; A; T; C; C; C]; Junk 1; Frag [C; G; A; T; C; T; A]]]
val pat3 : nucleotide pattern =
  List
   [Or
     [Frag [A; G; T; A; T; A; T; G; G; A; T; A];
      Frag [G; T; A; G; G; C; C; G; T];
      Frag [C; C; C; G; A; T; A; A; A; A; A; A; G; T; G; T; C; G; T];
      List [Frag [C; G; A; T; C; C; C]; Junk 1; Frag [C; G; A; T; C; T; A]]];
    Junk 2]
val pat4 : nucleotide pattern =
  Closure
   (List
     [Or
       [Frag [A; G; T; A; T; A; T; G; G; A; T; A];
        Frag [G; T; A; G; G; C; C; G; T];
        Frag [C; C; C; G; A; T; A; A; A; A; A; A; G; T; G; T; C; G; T];
        List [Frag [C; G; A; T; C; C; C]; Junk 1; Frag [C; G; A; T; C; T; A]]];
      Junk 2])
val pat5 : nucleotide pattern = Closure (Frag [A])
val pat6 : nucleotide pattern = Closure (Frag [A; G])
val pat7 : nucleotide pattern =
  Eager (List [Closure (Frag [A]); Closure (Frag [A; G])])
val pat8 : nucleotide pattern =
  List [Closure (Frag [A]); Closure (Frag [A; G])]
val matcher1 :
  nucleotide list -> (nucleotide list -> '_a option) -> '_a option = <fun>
val matcher2 :
  nucleotide list -> (nucleotide list -> '_a option) -> '_a option = <fun>
val matcher3 :
  nucleotide list -> (nucleotide list -> '_a option) -> '_a option = <fun>
val matcher4 :
  nucleotide list -> (nucleotide list -> '_a option) -> '_a option = <fun>
val matcher5 :
  nucleotide list -> (nucleotide list -> '_a option) -> '_a option = <fun>
val matcher6 :
  nucleotide list -> (nucleotide list -> '_a option) -> '_a option = <fun>
val matcher7 :
  nucleotide list -> (nucleotide list -> '_a option) -> '_a option = <fun>
val matcher8 :
  nucleotide list -> (nucleotide list -> '_a option) -> '_a option = <fun>
val accept_none : 'a -> 'b option = <fun>
val accept_all : 'a -> 'a option = <fun>
val accept_empty : 'a list -> 'b list option = <fun>
val test1 : bool = true
val test2 : bool = true
val test3 : bool = true
val test4 : bool = true
val test4 : bool = true
val test5 : bool = true
val test6 : bool = true
val test7 : bool = true
val test8 : bool = true
val test9 : bool = true
val test10 : bool = true
val test11 : bool = true
val test12 : bool = true
val test13 : bool = true
#
```

Hint

Start with the the following code. This code differs from the solution that you need, on two main grounds.

First, the following code defines a pattern-matcher generator that works only on DNA patterns and requires acceptors to return fragment options, which means its `make_matcher` is of type `pattern -> nucleotide list -> (nucleotide list -> 'a option) -> 'a option`. Your `make_matcher` should have the more-general type `'a pattern -> 'a list -> ('a list -> 'b option) -> 'b option`.

Second, the following code does not support the `Eager` pattern; you need to add support for that.

```
(* DNA fragment analyzer.  *)

type nucleotide = A | C | G | T
type fragment = nucleotide list
type acceptor = fragment -> fragment option
type matcher = fragment -> acceptor -> fragment option

type pattern =
  | Frag of fragment
```

```
   | List of pattern list
   | Or of pattern list
   | Junk of int
   | Closure of pattern

let match_empty frag accept = accept frag

let match_nothing frag accept = None

let rec match_junk k frag accept =
  match accept frag with
    | None ->
       (if k = 0
        then None
        else match frag with
               | [] -> None
               | _::tail -> match_junk (k - 1) tail accept)
    | ok -> ok

let rec match_star matcher frag accept =
  match accept frag with
    | None ->
       matcher frag
               (fun frag1 ->
                  if frag == frag1
                  then None
                  else match_star matcher frag1 accept)
    | ok -> ok

let match_nucleotide nt frag accept =
  match frag with
    | [] -> None
    | n::tail -> if n == nt then accept tail else None

let append_matchers matcher1 matcher2 frag accept =
  matcher1 frag (fun frag1 -> matcher2 frag1 accept)

let make_appended_matchers make_a_matcher ls =
  let rec mams = function
    | [] -> match_empty
    | head::tail -> append_matchers (make_a_matcher head) (mams tail)
  in mams ls

let rec make_or_matcher make_a_matcher = function
  | [] -> match_nothing
  | head::tail ->
      let head_matcher = make_a_matcher head
      and tail_matcher = make_or_matcher make_a_matcher tail
      in fun frag accept ->
           let ormatch = head_matcher frag accept
           in match ormatch with
                | None -> tail_matcher frag accept
                | _ -> ormatch

let rec make_matcher = function
  | Frag frag -> make_appended_matchers match_nucleotide frag
  | List pats -> make_appended_matchers make_matcher pats
  | Or pats -> make_or_matcher make_matcher pats
  | Junk k -> match_junk k
  | Closure pat -> match_star (make_matcher pat)
```
