Week 3 Discussion

CS 131 Section 1B
15 April 2022
Danning Yu

Announcements

- HW2 due 4/21 11:55pm
  - Start early! It's harder than HW1
- Homeworks should be submitted on BruinLearn, under Assignments
- Before submitting
  - Make sure your code compiles on SEASnet server
  - Make sure your function signatures are correct
  - Follow all instructions and specifications
  - Do not submit files in a .zip unless told to do so
- Help and starter code from past TAs
  - https://github.com/CS131-TA-team

OCaml

Function Types

- What are the types of the following functions?
  - `let f a b = a b`
    - `val f : ('a -> 'b) -> 'a -> 'b = <fun>`
  - `let g = f (fun x -> x + 1)`
    - `val g : int -> int = <fun>`

Tail Recursion

- A recursive function is tail recursive when the recursive call is the last thing executed
- Compiler optimization: if it's tail recursive, there’s no need to save the current stack frame
  - Compiler can simply optimize the recursive function into a loop

```
let rec factorial x =
    if x < 2 then 1
    else x * factorial (x - 1)
```

```
let factorial x =
    let rec fac y acc =
    if y < 2 then acc
    else fac (y - 1) (y * acc)
    in fac x 1
```

Tail Recursion

- Another example of non-tail recursive to tail recursive

```
let rec make_list = fun n ->
    if n < 1
    then []
    else List.append (make_list (n - 1)) [n]
```

```
let make_list = fun num ->
    let rec helper = fun n acc ->
        if n < 1
        then acc
        else helper (n - 1) (n :: acc)
    in
    helper num []
```

Homework 2

Homework 2 Description

- See week 2 discussion slides

Old HW2 Example

- http://web.cs.ucla.edu/classes/spring20/cs131/hw/hw2-2006-4.html
- Task: Build pattern matcher for genetic sequences
  - Genetic sequence consists of letters (nucleotides): A, G, C, T (adenine, guanine, cytosine, thymine), such as ATGCCAGATCATTGC...
- Patterns are similar to regular expressions
- `Frag [symbol list]`: exact match
  - e.g. `Frag [C; T; G]` matches input `[C; T; G]`
- `Or [pattern list]`: matches any rule within
  - `Or [Frag [C; T]; Frag [A; G]]` matches `[C; T]` and `[A; G]`
- `List [pattern list]`: concatenates multiple patterns
  - `List [Frag [A; G]; Or [Frag [C]; Frag [T]]]` matches input `[A; G; C]` and `[A; G; T]`

The `make_matcher` Function

- Hint code contains a `make_matcher` function with similar functionality to the assigned HW
- `make_matcher pattern` returns a matcher function for `pattern`
- Matcher function takes a fragment (input sequence) and an acceptor
- If the matcher finds a prefix of the fragment that matches the pattern, and the acceptor accepts the remaining suffix, it will return `Some suffix`
- Otherwise, the matcher returns `None`

The `make_matcher` Function

```
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
```

- Pattern matching is used to handle multiple type of patterns
  - `Frag` and `List` patterns are implemented with `make_appended_matchers`, since they both match something concatenated together
  - `Or` is handled by `make_or_matcher`
- Notice the extensive usage of higher order functions
  - Both `make_appended_matchers` and `make_or_matcher` transforms a matcher function to another matcher

`match_nucleotide` and Matching Fragments

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

- Basic building block for matchers
- `match_necleotide nt` creates a “matcher” that matches a single nucleotide `nt`
- To match a `Frag`, need to create a matcher that matches a list of nucleotides
- The function make_appended_matchers does this by appending multiple simple matchers together
  - `| Frag frag -> make_appended_matchers match_nucleotide frag`

Appending Matchers

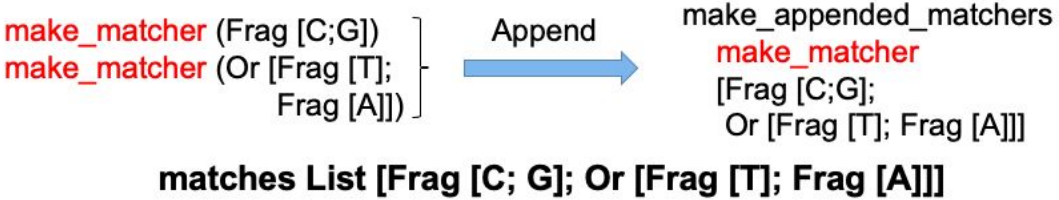
- Given 2 matchers `matcher1` and `matcher2`, return a matcher that matches `matcher1` and `matcher2` in a row
  - Key: Properly define a new acceptor for `matcher1`

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

- Appending multiple matchers with recursion

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

- Use this strategy to create the matcher for List
  - `| List pats -> make_appended_matchers make_matcher pats`



Matching `Or`

- Simpler than appending matchers: simply try the possibilities until a matching one is found, or all have been tried and none match

```
let rec make_or_matcher make_a_matcher = function
  | [] -> (fun accept frag -> None)
  | 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
```

HW2 Tips

- Consider the relationship between grammar and nucleotide pattern
  - In what situation are you matching `x and y` vs. `x or y`
- What's the difference between our homework and the old version?
- Besides matcher, we also need to implement a parser tree builder
  - How to do that? By keeping a list of how you got to each rule
- Try implementing the homework in Python while retaining a functional style, then port it to OCaml

Thank You