CS131, Spring19 – Discussion 1B

Week 2 (04/12/19)

Administrations

1. TA: Wenhao Zhang (wenhaoz@cs.ucla.edu)
2. Office hour:
   - Time: 9:30am ~ 11:30am, Monday
   - Location: 3rd floor, common area in Eng VI
3. Class Announcements
   - All TAs slides are under **resources tab** on Piazza
   - HW1 dued on 04/09
   - HW2 deadline extended to **04/21** (next Sunday)
   - Please dont copy & paste code from stackoverflow or other sources.
4. Today's agenda
   - Recap of last week
   - Currying
   - More on type inferences
   - Mutual recursion
   - Type definition
     - Trees
   - List module
   - Homework 2
     - Context free grammars
     - Hints

Quick recap from last week

- Writing recursive functions
  - Anonymous functions (lambda func)
  - `rec` keyword
- Pattern matching
  - `match with` syntax
  - `_` placeholder
  - `f::r` for list
  - `function` keyword
- List manipulation
  - `::`, `@`
  - List module

Currying

Recall that functions in OCaml can only take **one** argument. Then how come we can define a function with two arguments,

```
let mul x y = x*y;;
```

Questions:

1. What is `mul`, and what's its type?
2. What is `mul 2 1`, and what's its type?
3. What is `mul 2`, and what's its type?

Q1. `mul` is a function, it takes two arguments which are integers and it returns an integer as their product.

```
mul : int -> int -> int = <fun>
```

Q2: `mul 2 1` is a integer, the product of 1 and 2.

```
2 : int
```

Q3: `mul 2` is **NOT** a bug. It's a function which takes one argument which is integer and doubles it and return the value, which also has integer type.

```
mul 2 : int -> int = <fun>
```

Let's put all things together,

```
mul :   int -> int -> int = <fun>
mul 2   :   int -> int = <fun>
mul 2 1 :   int
```

You can observe we can just apply one argument to `mul` and have a partially applied function. This process is called, **currying**. The formal definition of currying is as follows:

> "currying is the technique of translating the evaluation of **a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument"** – Wikipedia

```
(* a function that takes multiple arguments *)
let mul a b = a*b;;
(* rhs: a sequence of lambda functions, each with a single argument *)
let mul = (fun a -> (fun b -> a*b));;
```

Recall that `fun b -> a*b` is an **anonymous (lambda) function** from last week. This practice of applying some of the arguments of a curried function to get a new function is called **partial application**. Now the `->` symbol should make more sense to you.

Why currying?

Short answer: code reusing.

More on type inferences.

As we just studied the concept of currying, and we're comfortable with `->` symbol. Let's switch gear and do more type inferences, shall we?

Let's say we have two functions, `mul` func we've already seen previously, and `somefunction` as defined below,

```
let rec somefunction f x = match x with
    | [] -> []
    | hd::tl -> (f hd)::(somefunction f tl);;
```

First of all, what's the type of `somefunction`?

```
val somefunction : (`a -> `b) -> `a list -> `b list

(* This func works exactly like List.map *)
```

This is what we called, **polymorphic function**. Polymorphic functions can have parameters of different types. OCaml's type inference determines that an expression is valid for any type, it is automatically made polymorphic, parameterized by type variables. Type variables are lowercase identifiers preceded by a single quote `'`, normally `'a`, `'b`, `'c` and so on

Now the real question, what's this `anotherfunc`, and its type?

```
let anotherfunc xs = somefunction (mul 2) xs;;

(* what's the result of this expr*)
anotherfunc [1;2;3;4;5];;

(*answer*)
int list : [2;4;6;8;10]
```

`somefunction` is known as a **higher-order function (HOF)**. Higher-order functions are just a fancy way of saying that the function takes a function as one of its arguments.

Another useful higher-order function is `List.fold_left`, which has the following type.

```
(*how fold_left works?*)
(*List.fold_left is the same function as defined here*)
let rec fold_left (f: 'a->'b->'a) (acc : 'a) (l: 'b list) : 'a =
match l with
```

```
| [] -> acc
| h::tl -> fold_left f (f acc h) tl

(* type of fold_left*)
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Note that `acc` is an *accumulator* we'll see in `sum1` function later. This accumulator saves the result of current recursive calls.

Let's say we'd like to define a function `sum` to add up all integers in a list. One way to implement it is as follows,

```
(*v1*)
let rec sum lst = match lst with
    | [] -> 0
    | f::r -> f + (sum r)

(* v2: Use accumulator, tail recursion *)
let rec sum1 acc lst = match lst with
    | [] -> acc
    | f::r -> sum (acc+f) r
```

There is nothing much to say of *v1* implementation. *v2* used an **accumulator** to save the current results for the current recursive calls. The *v2* implementation is called, **tail recursion**, in which **NO** extra computation is performed when subroutine returns. Tail recursion has the benefits of being memory efficient (space complexity is constant) when you have a lot of layers of recursive calls. The space complexity of **non-tail recursion** *O(n)*. When the number of call stackframes go beyond the allocated stack space, we'll run into `Error: Stack overflow during evaluation (loop recursion?)`. *If you want to know more details about tail recursion, please ask your best friend, Google* ;).

Tail recursion is always preferred than non-tail recursion for being memory efficient

Now let's see how we can use the higher-order function `fold_left` to write a `sum` function,

```
(*v3: sum function written using fold_left*)
let sum2 lst = fold_left (fun acc x -> acc + x) 0 lst;;

(*v4: simplify v3 using curring*)
let sum3 = fold_left (fun acc x -> acc + x) 0;;

(*v5: simplify v4 by using `(+)` function instead of the lengthy lambda func*)
let sum4 = fold_left (+) 0;;

(*v6: Use List.fold_left*)
let sum5 = List.fold_left (+) 0;;
```

```
(*Use sum5*)
sum5 [1;2;3;4;5];; (* => 15*)

(*Do you see the true power of functional programming with currying, HOF*)
(*If you're still not convinced, try to write it in C or Java with 1 line code*)
```

Now we can write a function that concate a list of strings like this by following the same approach.

```
(*the initial value of accumulator is empty string here as our return value is a string*)
(*remeber accumulator has the final result when the recursion finishes*)
(* `(^)` function takes in two strings and returns a concatenated string of them *)
let concate = List.fold_left (^) "";;

(*use concate*)
c ["c";"s";"1";"3";"1"];; (* => "cs131" *)
```

It's also easy to check that `sum5` and `concate` functions are **tail-recursive** as `List.fodl_left` is tail recursive. However, `List.fold_right` is **NOT** tail-recursive. Hence we use the former one here.

Brain teaser:

```
(* what's the type of this one? *)
let f f = f 1 1;;
(*Please figure out why without using your ocaml interpreter*)
(*hints: are the two 'f' on the lhs refering to the same thing?*)
```

Multual recursion

If you want to define two functions calling each other, you can follow this example,

```
(*mutual recursion, use `and` to concatenate two funcs that call each other*)
let rec even n =
    match n with
    | 0 -> true
    | x -> odd (x-1)
  and odd n =
    match n with
    | 0 -> false
    | x -> even (x-1);;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
```

Type definition – Variants

Variant types are one of the most useful features of OCaml and also one of the most unusual. They let you represent data that may take on multiple different forms, where each form is marked by an explicit tag. As we'll see, when combined with pattern matching, variants give you a powerful way of representing complex data and of organizing the case-analysis on that information.

```
(*Basic syntax of variants*)
type <variant> =
  | <Tag> [ of <type> [* <type>]... ]
  | <Tag> [ of <type> [* <type>]... ]
  | ...
```

```
type optionalInt =
| Something of int
| None

Something 3;;
val: int optionalInt = Something 3

None;;

(*Parameterized variants*)
type ('a) myOption =
| Mysome of 'a
| None

Mysome 1;;
int myOption = Mysome 1
```

Let's see how to define a binary tree. Each node either has two branches and a value `a`, or a leaf node.

```
type ('a) btree =
| Node of 'a * 'a btree * 'a btree
| Leaf;;

(*How to write a pre-order traversal func*)
let rec preorder bt = match bt with
| Node (v, left, right) -> v::((preorder left)@(preorder right))
| Leaf -> [];;
```

Question:

How to implement dictionary/hashmap (i.e. a collection of key-value pairs) data structure in OCaml?

```
(*dictionary*)
```

```
type ('k, 'v) myDictionary = ('k * 'v) list;;
```

How to write a getter func and setter func for myDictionary?

```
(*getter*)
let rec get1 key dict = match dict with
| [] -> None
| (k, v)::tl -> if k=key then Some v else get1 key tl;;

(*setter*)
let set1 key value dict = (key, value)::dict;;
```

HW2: Naive parsing of context free grammars

Start early on hw2 as it's much more difficult than hw1

Please check the hint code.

The main task in hw2 is to create a parse tree for a fragment (e.g. [“$”; “1”; “++”; “-”; “2”]) using the given grammar.

Quick recap:

- Symbols
  - Terminal symbol
  - Nonterminal symbol
- Rules
  - derivation rules
- Grammar
  - start point, and a set of rules.

Notice that the grammar in hw2 has a different representation.

```
(*grammar in hw1*)
let awksub_rules =
   [Expr, [T"("; N Expr; T")"];
    Expr, [N Num];
    Expr, [N Expr; N Binop; N Expr];
    Expr, [N Lvalue];
    Expr, [N Incrop; N Lvalue];
    Expr, [N Lvalue; N Incrop];
    Lvalue, [T"$"; N Expr];
    Incrop, [T"++"];
    Incrop, [T"--"];
    Binop, [T"+"];
    Binop, [T"-"];
    Num, [T"0"];
    Num, [T"1"];
    Num, [T"2"];
```

```
    Num, [T"3"];
    Num, [T"4"];
    Num, [T"5"];
    Num, [T"6"];
    Num, [T"7"];
    Num, [T"8"];
    Num, [T"9"]]

let awksub_grammar = Expr, awksub_rules

(*grammar in hw2*)
let awkish_grammar =
  (Expr,
   function
     | Expr ->
         [[N Term; N Binop; N Expr];
          [N Term]]
     | Term ->
     [[N Num];
      [N Lvalue];
      [N Incrop; N Lvalue];
      [N Lvalue; N Incrop];
      [T"("; N Expr; T")"]]
     | Lvalue ->
     [[T"$"; N Expr]]
     | Incrop ->
     [[T"++"];
      [T"--"]]
     | Binop ->
     [[T"+"];
      [T"-"]]
     | Num ->
     [[T"0"]; [T"1"]; [T"2"]; [T"3"]; [T"4"];
      [T"5"]; [T"6"]; [T"7"]; [T"8"]; [T"9"]])
```

Hw2-style grammar contains a **start symbol** and a **production function**. The **start symbol** is a nonterminal value. **Production function** is a function whose argument is a nonterminal value. It returns a grammar's alternative list for that nonterminal. **Alternative list** is a list of right hand sides. It corresponds to all of a grammar's rules for a given nonterminal symbol. By convention, an empty alternative list [] is treated as if it were a singleton list [[]] containing the empty symbol string.

Your 1st task is to write a function `convert_grammar g1` which converts hw1-style grammar to hw2-style grammar.

The second task is to write `parse_tree_leaves tree` func which traverses the

parse tree tree left to right and yields a list of the leaves encountered. *I believe we've seen one example in this discussion*

The third task is to write `make_matcher gram2` which returns a matcher for the grammar `gram2`.

More definitions:

1. fragment: a list of terminal symbols, e.g., [“3”; “+”; “4”; “xyzzy”].
2. acceptor: a function whose argument is a fragment frag. If the fragment is not acceptable, it returns None; otherwise it returns Some x for some value x.
3. matcher: a curried function with two arguments, 1) an acceptor accept and 2) a fragment frag. A matcher matches a prefix p of frag such that accept (when passed the corresponding suffix) accepts the corresponding suffix (i.e., the suffix of frag that remains after p is removed). If there is such a match, the matcher returns whatever accept returns; otherwise it returns None.

```
(*one example of acceptor*)
let accept_all derivation suffix = Some (derivation suffix);;
```

How does matcher work exactly?

1. find the next matching prefix using grammar derivation
2. if no prefix is found: return None.
3. else call acceptor on suffix if acceptor returns None return to step 1. else return whatever acceptor returns.

For example, let's say we have a fragment `["3", "+", "4", "-"]`, and you want to parse it using the given grammar in hw2 specs. Your matcher finds two possible **prefixes**, [“3”, “+”, “4”] and [“3”]. This can be easily checked using `Expr->[N Term; N Binop; N Expr] or Expr -> [N Term]` in `awkish_grammar`. The corresponding **suffixes** of `["3", "+", "4"] and ["3"]` are `["-"] and`[“+”, “4”, “-”]‘:

1) If the acceptor only accepts suffix `["-"]`, then your matcher found a good prefix (i.e. `["3", "+", "4"]`), and will **return whatever the acceptor returns**.

2) If the acceptor only accepts **empty suffix**, then you run out of options because the acceptor rejects the two suffixes metioned above. **In this case, your matcher fails to find a matching prefix, and returns None**.

The next task for you is to write a curried func `make_parser gram` which returns a parser for the grammar gram. When applied to a fragment frag, the parser returns an optional parse tree. If frag cannot be parsed entirely (that is, from beginning to end), the parser returns None.

Important: Please read the hint code before you start the last 2 tasks, `make_matcher` and `make_parser`. The hint code is extremely helpful, and click here to see more.