Week 2 Discussion

CS 131 Section 1B
8 April 2022
Danning Yu

Announcements

- HW1 due tonight (4/8)
- HW2 released, due 4/21
  - 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

Option

- Used to express possibly “invalid” values, such as null pointers, divide-by-zero
- Force proper handling of invalid case when one might exist

```
type 'a option =
  | Some of 'a
  | None
```

- Example

```
let divide a b = match b with
  |0.0 -> None
  | _ -> Some (a /. b)
```

Recursive Types

- `type` keyword can also define recursive data structures
  - Helpful in defining tree-like data structures

```
type tree =
   | Leaf of int
   | Node of tree * tree;;
let my_tree = Node (Node (Leaf 1, Leaf 2), Node (Leaf 3, Leaf
4))
```

- How to find leftmost leaf in the tree?

OCaml Functions

```
let sum a b = a + b;;
sum 1 2;;
  -: int: = 3
```

- No need for parentheses unless you want to change order of evaluation
  - `sum 1 2*3;; returns 9`
  - `sum 1 (2 * 3);; returns 7`
- Lambda function: `(fun x -> x + 1) 5;;`
- Equivalent forms
  - `let add_one x = x + 1;;`
  - `let add_one = fun x -> x + 1;;`

Currying and Partial Application (1 of 2)

- Internally, all functions in OCaml only have one argument
- Multi-argument functions
  - When supplied with one argument, it returns a new function that expects remaining arguments
- The “real behavior” of `add` with 2 arguments:
  - `let add = fun a -> (fun b -> a + b)`
- This is called currying: expressing a function with multiple arguments as a series of functions that takes one argument each
  - The nature of functions in OCaml (and many other functional languages)
- Normal way of expressing and using multi-argument functions is provided for our convenience

Currying and Partial Application (2 of 2)

- Implement `add_one` with `add`:

```
let add a b = a + b
let add_one x = add 1 x
```

- The `x` argument can be omitted

```
let add_one = add 1
```

- This is called partial application
  - Supply the function with less argument it takes, and get a new function that expects remaining arguments
- Function return types
  - `add_one` has type `int -> int`
  - `add` takes an `int (1)`, and returns `add_one` (of type `int -> int`)
- Type of `add`: `int -> (int -> int)`
- `->` is right-associative, yielding `int -> int -> int`, OCaml's output

`List` Module Functions

- `filter_map`: A combination of filter and map
- `fold_left`/`fold_right`: Summarize a list into a single value
- `flatten`: Concatenate a list of lists into a single list
- `split`/`combine`: Conversion between list of tuples and tuple of lists

`List.filter_map`

- `List.filter_map f l` applies `f` to every element of `l`, filters out the `None` elements and returns the list of `Some` elements.

```
let square_of_even l = List.filter_map (fun x ->
   if x mod 2 = 0 then Some (x * x) else None) l;;
val square_of_even : int list -> int list = <fun>


square_of_even [1;2;3;4;5];;

- : int list = [4; 16]
```

`List.fold_left`

- `List.fold_left f a [b1; ...; bn]`
  - Equivalent to `f (... (f (f a b1) b2) ...) bn.`
- Useful to accumulate over a list
  - `let list_sum l = List.fold_left (fun x y -> x + y) 0 l`
- More concise version
  - `let list_sum = List.fold_left (+) 0`
- Similarly, `fold_right` exists
  - Different associativity
- These two operations are called "reduce" in some other languages

List.flatten, List.split, List.combine

```
List.flatten [["a";"b"];["c";"d"];["e";"f"]];;
- : string list = ["a"; "b"; "c"; "d"; "e"; "f"]

List.split [("A", 1); ("B", 2); ("C", 3)];;
- : string list * int list = (["A"; "B"; "C"], [1; 2; 3])

List.combine ["A"; "B"; "C"] [1; 2; 3];;
- : (string * int) list = [("A", 1); ("B", 2); ("C", 3)]
```

Pattern Matching with `function`

- `function` keyword can be used to simplify pattern matching on functions with single parameter

```
let rec list_size ls = match ls with
  | [] -> 0
  | _ :: tl -> 1 + list_size tl
```

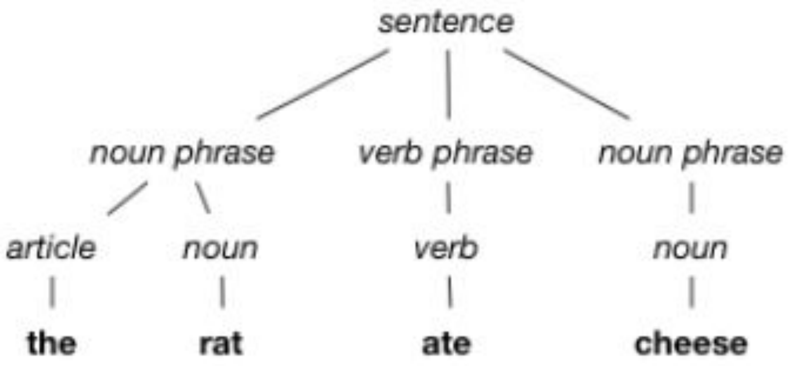
```
let rec list_size = function
  | [] -> 0
  | _ :: tl -> 1 + list_size tl
```

- Further simplification if argument is a tuple
  - `let my_fst t = match t with (a, _) -> a`
  - `let my_fst (a, _) = a`

Homework 2

Homework 2

- Problem to solve: how to find the parse tree for a given sentence?
  - Given a context-free grammar
- Finding a parse tree is equivalent to finding the derivation
- Old homework is provided as a hint
  - http://web.cs.ucla.edu/classes/spring20/cs131/hw/hw2-2006-4.html
  - Includes sample code for a similar problem, can be used as a starting point



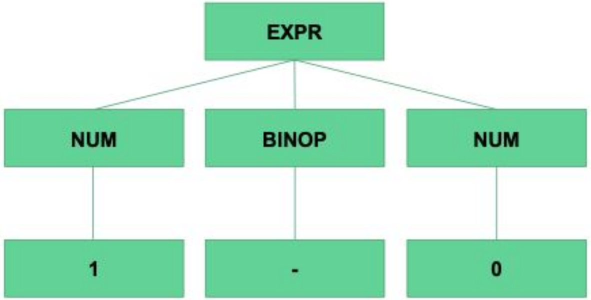
HW2 Problem 1

- Grammars are now represented as functions that take a non-terminal and return a list of rules
  - Allows us to access all the rules associated with a particular nonterminal
  - Write a function to convert an old grammar to the new style
- Note that your solution does not need to use pattern matching even though the example does! You need to write a function that takes a non-terminal symbol and returns a list of right-hand sides of rules

HW2 Problem 2

- Write a function `parse_tree_leaves tree` that returns a list of leaves in the tree from left to right
  - In the example below, return `[“1”; “-”; “0”]`
- Basically, traverse all the leaves in a parse tree
- Tree data structure

```
type ('nonterminal, 'terminal) parse_tree =
  | Node of 'nonterminal * ('nonterminal, 'terminal) parse_tree list
  | Leaf of 'terminal
```



HW2 Problem 3 - Matcher

- Write a function `make_matcher gram` that returns a matcher for the given grammar
- Matcher tells us if some prefix of the input can be generated using our grammar
  - Using our earlier grammar and input `["1"; "+"; "1"; "-"]`, we can match `["1"; "+"; "1"]`
- We are not looking for the longest match - just the first one that we find using our top-down parsing rules

HW2 Problem 3 - Acceptor Function

- Given as an argument to the matcher
- Acceptor function takes a suffix and tells us whether it is “acceptable”
  - `Some x` if acceptable, `None` if not, where `x` is usually the suffix itself
- Lets us have additional conditions for matches, such as only accepting full matches
- If acceptor returns `None`, backtrack
  - If the last rule we tried didn’t work

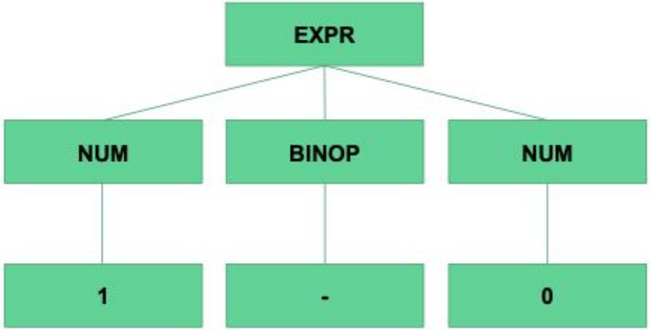
```
let accept_all string = Some string
let accept_empty_suffix = function
  | _::_ -> None
  | x -> Some x
matcher accept_all ["9"; "+"; "$"]
```

HW2 Problem 3 - Suggested Approach

- Expand parse tree using the top-down derivation rules from the earlier slide
- Once you have derived some prefix of the input string, call the acceptor with the unmatched input symbols (suffix)
  - If the acceptor returns `Some value`, return that same `value`
  - If the acceptor returns `None`, backtrack and try the next rule

Example:

1. Input is `[“1”; “-”; “0”;”+”]`
2. Use grammar rules until there are no non-terminals and all the terminals match with our input
3. Give unmatched suffix `[“+”]` to acceptor
4. Return the acceptor’s return value if the suffix was accepted; otherwise, try other derivations



HW2 Problem 4 - Parser

- Write a function `make_parser gram` that returns a parser for the grammar gram.
- Parser differs from a matcher in two ways:
  - It does not take an acceptor as an input - it will only return full matches
  - It returns a parse tree, not a suffix
- Example: parsing `[“1”; “-”; “0”; “+”]` must return `None` (non-empty suffix)

HW2 Problems 5 - 7

- 5. Write a non-trivial test case for `make_matcher`; this should include writing your own grammar
- 6. Using the same grammar, write a test case for `make_parser`
- 7. Write a report
  - Did you use `make_matcher` to write `make_parser` or vice versa or neither, and why?
  - Explain the weaknesses of your solution, provide test cases if possible
    - There will be some weaknesses, your parser/matcher might fail with a specific type of grammar for example

Thank You