1 Introduction and Resources

These problems were originally written for a one-week class on functional programming in OCaml at Canada/USA Mathcamp 2016. They're based on a similar class I taught in Haskell in 2013. If you'd like to see the notes for that class, you can find them at `http://nicf.net/static/haskellpacket.zip`.

Exercises marked with a star are especially challenging. Whether or not an exercise has a star on it, you should always feel free to ask for help. I have intentionally written more exercises than I think you will finish during class time, and they're not necessarily written to be done in order, so feel free to jump around a lot if you get stuck and/or bored. If you manage to finish everything, tell me and I'll come up with something for you to do.

We'll be exploring only a small subset of the language; the point is to familiarize you with the methods and tools of functional programming rather than to teach you how to write OCaml in a way that is useful in the real world. That said, I did write OCaml in the real world for a couple years, and if you're reading these notes at Mathcamp I'm happy to talk to you about it if you're curious!

In this class we'll be using "Core," an open-source replacement for the OCaml standard library. You should be aware that, as a replacement for the standard library, Core has a lot of modules that have the same names as ones in OCaml's ordinary standard library. (For example, both define modules called `List` and `String`.) When you're looking up documentation for one of these, make sure you're reading about Core's version.

The nature of this class is such that you will have to spend some time doing Internet research and learning topics on your own — I've intentionally included some problems that involve exploring things I won't be talking about directly in class. The following are good resources to get you started:

- OCaml.org (`https://ocaml.org/`). The official OCaml website, which includes lots of documentation and tutorials. (Beware: their tutorials aren't about Core!)
- Real World OCaml (`https://realworldocaml.org/`). This is a good, comprehensive book about OCaml and Core. It includes instructions for setting up an OCaml programming environment, which will be useful if you'd like to continue to work on your own computer.
- OCaml Cheat Sheets (`https://ocaml.org/docs/cheat_sheets.html`). A collection of short, quick references for OCaml syntax and conventions. Nice to have open on another tab while you're working.
- Core documentation. You can find this in several places, probably the best of which is `https://ocaml.janestreet.com/ocaml-core/latest/doc/core_kernel/`.
- Core source. This is available on GitHub at `https://github.com/janestreet/core_kernel/tree/master/src`.

2 Problems

1. Use `List.fold` to write the factorial function in one line in a way that doesn't have any explicit recursion. (That is, your function shouldn't call itself, but the fold function is of course itself recursive.)

2. Write a function `range` that takes three integers, `i`, `j`, and `step`, and returns a list containing the sequence of integers less than `j` starting at `i` and increasing by `step` each time. For example:

```
range 0 8 1 = [0;1;2;3;4;5;6;7]
range 3 15 3 = [3;6;9;12]
```

3. What are the types of the following expressions? (You can check your answers in `utop`, but do that only after you have a guess.)

   (a) `[[]; []]`

   (b) `[[]; [[]]]`

   (c) `fun x y z -> x = 7 || y = z`

   (d) `List.filter [1; 2]`

   (e) `(+) 3`

   (f) `List.map ~f:(List.map ~f:(/.))`

   (g) `fun x y -> y x x`

   (h) `fun x y -> y (y x x) (y x x)`

4. (a) Write a function that sorts a list using quick sort. (Hint: Use the `List.partition_tf` function.)

   (b) Write a function that sorts a list using merge sort.

5. (a) Write a function called `subsets` that takes a list and returns a list of all ordered sublists of it.

```
subsets [1;2] = [[];[1];[2];[1;2]]
subsets [0;6;2] = [[];[0];[6];[0;6];[2];[0;2];[6;2];[0;6;2]]
```

   The order of the output doesn't have to match the order in these examples.

   (b) Write a function called `choose` that takes a list and a number and returns all sublists of the list of that length.

```
choose 3 [1;2;3;4] = [[1;2;3];[1;2;4];[1;3;4];[2;3;4]]
choose 2 [66;77] = [[66;77]]
choose 0 [1;2;3;4;5;6;7;8] = [[]]
```

   (c) If you didn't already, write a version of `choose` so that `choose k` runs in polynomial time once you've chosen a value of `k`. (In particular, it shouldn't look at the entire output of `subsets`.)

6. (a) Write a function that...
   i. ...does the same thing as `List.tl`.
   ii. ...takes a list and an integer `n` and returns an `option` containing either the `n`'th element of the list, if there is one, or `None` otherwise.
   iii. ...reverses a list.
   iv. ...does the same thing as `List.find_map`.
   v. ...finds the maximum value in a list of integers.
   vi. ...takes a list and an element and "intersperses" that element with the list. For example,

   ```
   intersperse 3 [7;8;9] == [7;3;8;3;9]
   ```

   vii. ...does the same thing as `List.group`.
   viii. ...does the same thing as `List.partition_tf`.
   ix. ...does the same thing as `List.remove_consecutive_duplicates`.
   x. ...takes an integer `n` and returns the `n`'th prime number.

   (b) For the ones that aren't duplicates of functions from Core, figure out the type of each of these functions before the typechecker tells you the answer.

   (c) (*) How many of these can be implemented using folds without any explicit recursion?

7. Consider the following function definition:

```
let rec f = function
| 0 -> 1
| 1 -> 1
| n -> f (n - 1) + f (n - 2)
```

   (a) What does `f` compute?

   (b) How many time will `f` be called to compute `f 5`? What about `f n` for any `n`?

   (c) Write a version of `f` that uses a list to internally keep track of the values it computes. Make sure it can compute `f 100` in a reasonable amount of time.

8. (a) Create a datatype for representing binary trees. Every node should either branch into two subtrees (the *left* and *right* subtrees) or be a leaf. Each node, whether it branches or not, should have a value attached to it, the type of which is arbitrary but the same throughout the binary tree.

   (b) Write a function that...
   i. ...tells whether two trees are equal.
   ii. ...outputs the contents of the binary tree as a list. Write three versions: one for preorder, one for inorder, and one for postorder. (Look these up if you don't know them.)
   iii. ...outputs the maximum element of a binary tree.
   iv. ...adds all the elements of a binary tree.
   v. ...determines whether a binary tree is symmetric, that is, whether flipping the left side about the middle produces the right side.

vi. ...takes a tree and an integer `n` and outputs a list of all the nodes at the `n`'th level from the top.

vii. ...outputs the depth of a tree, that is, how far down the deepest node is.

(c) Figure out the type of each of these functions before the typechecker tells you the answer.

(d) (*) Write a version of `List.fold` for binary trees. What should its type be? How many functions from 8b can you implement in terms of your function?

9. In this and a few other problems, you'll write a simple interpreter for a small programming language. The language is described at the very end of this packet on page 7, which I encourage you to go read now.

   Our interpreter runs in two steps. The first step, called *parsing*, involves taking the text of the program and turning it into a data structure, called an *abstract syntax tree* or *AST*, which represents the program. The second step involves taking the AST and running the corresponding program.

   We will begin tackling a portion of this second part. These are the data structures we will worry about in this problem:

```
type expression =
  | Var of string
  | IntLit of int
  | BoolLit of bool
  | Neg of expression
  | Not of expression
  | Plus of expression * expression
  | Minus of expression * expression
  | Times of expression * expression
  | Divide of expression * expression
  | LT of expression * expression
  | GT of expression * expression
  | Equal of expression * expression
  | And of expression * expression
  | Or of expression * expression

type value = IntVal of int | BoolVal of bool
```

   The different types of expression are described in more detail in the language specification.

   The files `parser.ml` and `parser.mli` that I've provided define a function `parse_expr : string -> expression option` which will be useful for testing purposes. (If you want to figure out parsing for yourself later on in Problem 12, be sure to only look at the `mli`!) For example:

```
parse_expr "-(x/2)" = Some (Neg (Divide (Var "x", IntLit 2)))
parse_expr "a and not b" = Some (And (Var "a", Not (Var "b")))
parse_expr "x+5*y" = Some (Plus (Var "x", Times (IntLit 5, Var "y")))
parse_expr "true or x > y + 5" =
  Some (Or (BoolLit true, GT (Var "x", Plus (Var "y", IntLit 5))))
```

```
parse_expr "this is not a valid expression" = None
```

You should play with the `parse_expr` function for a while if it's not clear.

Write a function

```
eval_expr : (string * value) list -> expression -> value option
```

which takes an expression to a value. The first parameter is an *environment* which contains a correspondence between variable names and values. Return `None` if you are unable to evaluate the expression.

You might find the functions in `List.Assoc` useful.

10. You might have noticed that it's possible to write down expressions that correspond to expressions in which the types don't make sense, like "`true < 5 + (false and 7)`". Read about how *generalized algebraic datatypes* (often called *GADT's*) work in OCaml and use them to fix this defect. In this problem — and only this problem — you can assume that variables in our language always have integer values.

11. In this problem we'll continue constructing the interpreter. Here we'll focus on writing the part that actually does things. We'll use this type:

```
type statement =
  | Assign of string * expression
  | If of expression * statement * statement
  | While of expression * statement
  | Block of statement list
  | Print of expression
  | Empty
```

The different types of statement are described in more detail in the language specification. (The `Empty` statement does nothing; its main purpose is to stand in for a missing `else` clause in an `if` statement.)

There are functions in the provided `parser.ml` called `parse_string` and `parse_file` to help you generate examples of statements for testing. Play around with `parse_string` for a while if you don't understand the syntax rules.

(a) Write a function `run : statement -> unit` that runs the provided program. The only visible side-effects will come from `print` statements. (It might be helpful at this point to learn how Core's hash tables work if you haven't already.)

(b) Write a purely functional version of `run` — that is, a version which doesn't have any side-effects — which returns a list of strings that would have been printed if it had been run for real.

(c) Which version was easier to write? What would the advantages be of each one in a more realistic setting? (Talk to me about this part when you get here.)

12. (*) When designing the interpreter problems, I wrote `parse_string` and the other functions like it using a library that implements something called *parser combinators*. In our context, a *parser* is a function that takes a string and produces the expression or statement

it corresponds to, and a parser combinator is a way of combining simple parsers to produce more complicated ones.

There are several parser combinator libraries available for OCaml. I wrote a small one and called it "Miniparse"; the code for it is included in the materials for the class. The file `miniparse.mli` has some documentation and examples in it as well.

One of the more famous parser combinator libraries was written in Haskell and is called Parsec; it might be useful to read about it to learn how parser combinators work in general.

Write a version of `parse_string` yourself using Miniparse, or any other parser combinator library you can get working.

(Note: if you look at enough examples in the documentation for some parser combinator libraries, you might find one that implements a parser for a toy language called "While." This language is very similar to the one I'm asking you to parse, so if you'd like the extra challenge, avoid reading this example in too much detail.)

13. (*) Implement Miniparse yourself. In particular, write a version of everything mentioned in `miniparse.mli` and make sure your interpreter still works with your code. (The type `Input.t` can simply be `char list`.)

3 Language Specification

This is an informal description of the language you'll be writing an interpreter for in Problems 9, 11, and 12.

An *expression* is one of the following things:

- A variable name, which must begin with a letter and contain only letters, numbers, and underscores.
- An integer.
- One of the constants `true` or `false`.
- A unary operator applied to an expression. The unary operators are `-` and `not`.
- A binary operator applied to two expressions. The binary operators are `and`, `or`, `+`, `-`, `*`, `/`, `==`, `<`, and `>`.
- Parentheses surrounding another expression.

There are two *types* an expression might have: integer and boolean. The operators work on types in the way you would expect. It's not possible to tell the type of an expression at compile time because variables can change types as they are assigned new values.

A *statement* is one of the following things:

- An assignment: `v := e;`, where `v` is a variable name and `e` is an expression. This has the effect of evaluating `e` and assigning the resulting value to `v`.
- An `if` statement: `if c then s; else t;`, where `c` is an expression which should evaluate to a boolean and `s` and `t` are statements. If `c` evaluates to `true`, then `s` is executed, otherwise `t` is. The `else t;` can be omitted, and both `s` and `t` are allowed to be blocks.
- A `while` statement: `while c do s;`, where `c` is an expression which should evaluate to a boolean and `s` is a statement. This has the effect of executing `s` as long as `c` keeps evaluating to `true`. As before, `s` can be a block.
- A `print` statement: `print e;`. This prints the value of `e` followed by a newline.
- A block statement: `{s1; s2; ... sn;}`. This executes the enclosed statements in order. Useful for the bodies of `if` and `while` statements.

All statements except for blocks (and therefore `if` and `while` statements which end in blocks) should end in a semicolon.

The following program computes the factorial of 7 and prints it.

```
a := 7;
fact := 1;

i := a;
while i > 0 do
{
    fact := fact * i;
```

```
    i := i - 1;
}

print fact;
```