Midterm 2 Study Guide Solutions

Handout 27
CSCI 334: Spring 2025

The second midterm is cumulative, drawing from all topics covered throughout the semester. More emphasis will be placed on topics from the second half of the semester.

First half	Second half
What is a Programming Language?	*Gumptionology 101*
An Introduction to F#	*Beating the Averages*
More F#	*Parsing*
F#: The Cool Stuff	*Evaluation*
Higher Order Functions	*Implementing Variables*
Syntax	*Implementing Scope*
Lambda Calculus, Part 1 and *Part 2*	*Type Inference* (handout)
Function Graphs (handout)	

Q1. (10 points) ..... Terminology

You should do this exercise on your own.

Q2. ..... F# Coding

Write a function:

```
isPalindrome: string -> bool
```

that returns `true` if the given string is a palindrome, otherwise `false`. Recall that a palindrome is a `string` that reads the same forwards and backwards.

Here is a simple recursive solution.

```
let isPalindrome (s: string) : bool =
    let rec isP (i: int)(j: int) : bool =
        if i >= j then
            true
        elif s[i] = s[j] then
            isP (i+1) (j-1)
        else
            false

    isP 0 ((String.length s) - 1)
```

Q3. ..... Lambda Calculus Reductions

Reduce the following expressions. If the expression does not have a normal form, explain why.

(a)

`(λf.fz)(λx.x)`	given
`((λx.x)z)`	β-reduce `(λx.x)` for `f`
`((z))`	β-reduce `z` for `x`
`z`	eliminate parens

(b)

`(λx.λy.x(yy))λz.z`	given
`(λy.(λz.z)(yy))`	β-reduce `(λz.z)` for `x`
`(λy.(yy))`	β-reduce `(yy)` for `z`
`λy.yy`	eliminate parens

(c) Observe here that I am careful to use the normal order reduction. This expression has no normal form because we end up deriving a subexpression that contains our given, which means that our derivation will never terminate.

`(λx.λy.xxy)(λx.λy.xxy)`	given
`(λa.λy.aay)(λx.λy.xxy)`	α-reduce `a` for `x`
`(λa.λb.aab)(λx.λy.xxy)`	α-reduce `b` for `y`
`(λb.(λx.λy.xxy)(λx.λy.xxy)b)`	β-reduce `(λx.λy.xxy)` for `a`
...	...

(d)

`(λf.λx.fx)(λy.z)((λx.xx)(λx.xx))`	given
`(λx.(λy.z)x)((λx.xx)(λx.xx))`	β-reduce `(λy.z)` for `f`
`(λx.(z))((λx.xx)(λx.xx))`	β-reduce `x` for `y`
`(λx.z)((λx.xx)(λx.xx))`	eliminate parens
`(λa.z)((λx.xx)(λx.xx))`	α-reduce `a` for `x`
`(z)`	β-reduce `((λx.xx)(λx.xx))` for `a`
`z`	eliminate parens

Q4. (10 points) ..... Partial and Total Functions

(a) Partial function: $\{\langle n, \texttt{fibonacci}\ n\rangle \mid n \in \mathbb{Z} \wedge n \geq 0\}$

(b) Looking at our `gcd` function, the only operation that might be problematic is `%`. The mathy definition is that $\bmod$ is defined for all divisors and quotients in $\mathbb{Z}$. Some programming languages, like C, assume that the quotient and divisor are not negative, and will give the wrong answer otherwise (eek!). But if we stick with the mathy definition (instead of C), then `gcd` appears to be undefined nowhere (because the case where `b = 0` is explicitly handled and defined), so the function is total: $\{\langle a, b, \texttt{gcd}\ a\ b\rangle \mid a, b \in \mathbb{Z}\}$

(c) Total function: $\{\langle x, |x|\rangle \mid x \in \mathbb{Z}\}$

One can use function graphs to enforce preconditions when implementing the above functions. These are also a concise form of documentation that you might consider putting into a Javadoc or Python docstring. Other programmers (or "future you") will thank you.

Q5. (10 points) ..... Parsing and Evaluation

Here is one possible solution.

```
open Combinator

type Expr =
| Increment
| Decrement
| MoveLeft
| MoveRight
| Print

type State = { tape: int list; pos: int }

let op =
    (
        (pchar '+' |>> fun _ -> Increment) <|>
        (pchar '-' |>> fun _ -> Decrement) <|>
        (pchar '<' |>> fun _ -> MoveLeft) <|>
        (pchar '>' |>> fun _ -> MoveRight) <|>
        (pchar 'p' |>> fun _ -> Print)
    ) <!> "op"

let expr = pmany0 op <!> "expr"

let grammar = pleft expr peof <!> "grammar"

let rec getAtIndex xs i =
    match xs with
    | [] -> failwith "Cannot find index in list."
    | y::_ when i = 0 -> y
    | _::ys -> getAtIndex ys (i - 1)

let rec updateAtIndex xs i v =
    match xs with
    | [] when i <> -1 -> failwith "Cannot find index in list."
    | [] -> []
    | _::ys when i = 0 -> v::ys
    | y::ys -> y::updateAtIndex ys (i - 1) v

let parse input =
    let i = prepare input
    match grammar i with
    | Success(ast,_) -> Some ast
    | Failure(_,_) -> None

let ephsPrint v =
    let s =
        match v with
        | 0 -> "a"
        | 1 -> "b"
        | 2 -> "c"
        | 3 -> "d"
        | 4 -> "e"
```

```
        | 5 -> "f"
        | 6 -> "g"
        | 7 -> "h"
        | 8 -> "i"
        | 9 -> "j"
        | 10 -> "k"
        | 11 -> "l"
        | 12 -> "m"
        | 13 -> "n"
        | 14 -> "o"
        | 15 -> "p"
        | 16 -> "q"
        | 17 -> "r"
        | 18 -> "s"
        | 19 -> "t"
        | 20 -> "u"
        | 21 -> "v"
        | 22 -> "w"
        | 23 -> "x"
        | 24 -> "y"
        | 25 -> "z"
        | _ -> "poo"
    printf "%s" s

let evalOp (e: Expr)(s: State) : State =
    match e with
    | Increment ->
        let cur = getAtIndex s.tape s.pos
        let upd = cur + 1
        let tape' = updateAtIndex s.tape s.pos upd
        { s with tape = tape' }
    | Decrement ->
        let cur = getAtIndex s.tape s.pos
        let upd = cur - 1
        let tape' = updateAtIndex s.tape s.pos upd
        { s with tape = tape' }
    | MoveLeft ->
        if s.pos > 0 then
            { tape = s.tape; pos = s.pos - 1 }
        else
            printfn "Tape head out-of-bounds (on left)."
            exit 1
    | MoveRight ->
        if s.pos < 10 then
            { tape = s.tape; pos = s.pos + 1 }
        else
            printfn "Tape head out-of-bounds (on right)."
            exit 1
    | Print ->
        let cur = getAtIndex s.tape s.pos
        ephsPrint cur
        s

let rec eval (es: Expr list)(s: State) : State =
```

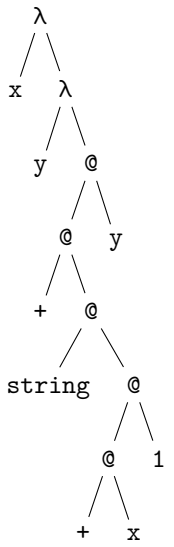
```
    match es with
    | [] -> s
    | op::ops ->
        let s' = evalOp op s
        eval ops s'

[<EntryPoint>]
let main args =
    let file = args[0]
    let input = System.IO.File.ReadAllText file
    let ast_maybe = parse input
    match ast_maybe with
    | Some ast ->
        eval ast { tape = [0;0;0;0;0;0;0;0;0;0]; pos = 0 } |> ignore
        0
    | None ->
        printfn "Invalid program"
        1
```

Q6. ..... Type Inference

Infer the type of the following function.

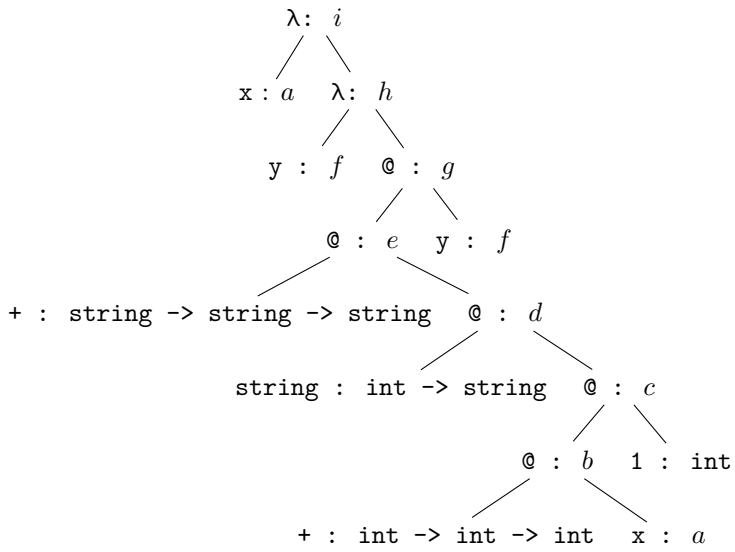
```
let f (x: int) y = string (x + 1) + y
```

We start by turning the F# function into a lambda expression. Putting the function calls `string` and `+` in prefix form clarifies their relationship to their arguments.



It can sometimes be hard to see how a curried function, like `+`, ends up appearing in an AST. First, recognize that `+` takes two curried arguments, which means it must have two applications. Second, remember that the rightmost argument must be higher in the subtree, because the leftmost argument is the first one applied.

Let's now label the tree with all of the known and unknown types.



You might be wondering how I knew that the inner most `+` took `int`s while the outermost `+` took `string`s. The Hindley-Milner algorithm will try all combinations of possible `+` expressions. However, since I am human, and I don't like doing lots of extra writing, by carefully examining the tree one can see that in each case only one choice will work. If you feel like being a stickler for detail, feel free to work through it yourself.

Let's put the above tree into tabular form, find all of the constraints, and solve for the unknown types. The constraints were generated by asking whether the subexpression is an abstraction, an application, or something else.

subexpression	type	constraint
`+` (inner)	`int -> int -> int`	n/a
`x`	`a`	n/a
`+ x`	`b`	`int -> int -> int = a -> b`
`1`	`int`	n/a
`string`	`int -> string`	n/a
`+ x 1`	`c`	`b = int -> c`
`+` (outer)	`string -> string -> string`	n/a
`string (+ x 1)`	`d`	`int -> string = c -> d`
`+ string (+ x 1)`	`e`	`string -> string -> string = d -> e`
`y`	`f`	n/a
`(+ string (+ x 1) y)`	`g`	`e = f -> g`
`λy.(+ string (+ x 1) y)`	`h`	`h = f -> g`
`λx.λy.(+ string (+ x 1) y)`	`i`	`i = a -> h`

There's a lot going on here! But if we work through carefully, we can solve this. Let's start at the top.

(a) If we substitute `b = int -> c` into `int -> int -> int = a -> b`, we get `int -> int -> int = a -> int -> c`. That tells us that `a = int` and `c = int`.

(b) Returning to `b = int -> c`, we also now know that `b = int -> int`.

(c) If we take `int -> string = c -> d` with `c = int`, we derive `d = string`.

(d) If we substitute `d = string` into `string -> string -> string = d -> e`, we can conclude that `e = string -> string`.

(e) If `e = f -> g` and `e = string -> string`, then `f = string` and `g = string`.

(f) If `h = f -> g`, `f = string`, and `g = string`, then `h = string -> string`.

(g) If `i = a -> h`, `a = int`, and `h = string -> string`, then `i = int -> string -> string`.

(h) Therefore, the function `f : int -> string -> string`. Does our work check out in `dotnet fsi`?

```
> let f (x: int) y = string (x + 1) + y;;
val f: x: int -> y: string -> string
```

Yes!

Q7. ..... Global Variables

Suppose you have the following program, in a language with globally-scoped variables, where `:=` stands for assignment.

```
x := 2
y := x := x + 1
y := y + x
print y + x
```

The tree below shows values as they propagate upward. The result of evaluation is that the program both prints `9` and returns `9`.