

CS421 Summer 2006 Midterm Study Guide

CS 421 — Introduction to Programming Languages
Summer Term 2006

1. OCaml and Higher-Order Functions

Solution Note: You just need to give the functions below; anything additional just shows that they work.

- (a) Using just fold, write a function `max` that finds the largest element in a list of numbers. Consider this to be 0 for an empty list.

```
1 # let max list =
2     List.fold_right (fun x y -> if x > y then x else y) list 0;;
3 val max : int list -> int = <fun>
4
5 # max [];;
6 - : int = 0
7
8 # max [ 1 ; 2 ; 5 ; 3 ; 8 ; 4 ; 10 ; 2 ; 5 ];;
9 - : int = 10
```

- (b) Say you are given the following definition of lists:

```
type 'a mylist = Empty | Cons of 'a * 'a mylist
```

Write a function `myMap` that takes a function and maps it over the elements of a `mylist`.

```
1 # type 'a mylist = Empty | Cons of 'a * 'a mylist;;
2 type 'a mylist = Empty | Cons of 'a * 'a mylist
3
4 # let rec myMap f l =
5     match l with
6     | Empty -> Empty
7     | Cons (x,xs) -> Cons (f x, myMap f xs);;
8 val myMap : ('a -> 'b) -> 'a mylist -> 'b mylist = <fun>
9
10 # myMap (fun x -> x * 3) (Cons (3, Cons (4, Cons (5, Empty)))) ;;
11 - : int mylist = Cons (9, Cons (12, Cons (15, Empty)))
```

- (c) Say you are given the following definition of trees, with an arbitrary number of children:

```
type 'a mytree = Leaf of 'a | Node of 'a mytree list
```

Write a function `fold_mytree` that takes a function for the node case, a function for the leaf case, an identity, and a `mytree`, and returns the result of folding the functions over the tree. Note, you may want to think about mutually recursive functions here.

```
1 let rec fold_mytree f g z t =
2   match t with
3     | Leaf x -> g x
4     | Node tl -> fold_mytree_list f g z tl
5 and fold_mytree_list f g z tl =
6   match tl with
7     | [] -> z
8     | t :: ts -> f (fold_mytree f g z t) (fold_mytree_list f g z ts);;
9
10 val fold_mytree : ('a -> 'a -> 'a) -> ('b -> 'a) -> 'a ->
11   'b mytree -> 'a = <fun>
12 val fold_mytree_list :
13   ('a -> 'a -> 'a) -> ('b -> 'a) -> 'a -> 'b mytree list -> 'a = <fun>
14
```

- (d) Using `fold_mytree`, write a function `sum_mytree` that will sum the values of the leaves.

```
1 # let t1 = Leaf 3 ;;
2 val t1 : int mytree = Leaf 3
3
4 # let t2 = Node [ Node [ Node [Leaf 3 ; Leaf 4 ; Leaf 5] ;
5                       Node [Leaf 6]
6                       ] ;
7                       Node [Leaf 7 ; Leaf 8]
8                       ];;
9 val t2 : int mytree =
10   Node
11     [Node [Node [Leaf 3; Leaf 4; Leaf 5]; Node [Leaf 6]];
12     Node [Leaf 7; Leaf 8]]
13
14 # let rec sum_mytree t =
15   fold_mytree (fun x y -> x + y) (fun x -> x) 0 t;;
16 val sum_mytree : int mytree -> int = <fun>
17
18 # sum_mytree t1;;
19 - : int = 3
20
21 # sum_mytree t2;;
22 - : int = 33
```

2. Type Derivations

Note, type derivation rules are on the last page.

Solution Note: Sorry the solutions are tiny, it may help to magnify them in Acrobat or your PDF reader of choice.

(a) Show the complete type derivation for the following term:

(fun x -> if x = 0 then true else false) 3

$$\begin{array}{c}
 \frac{}{\{x : \text{int}\} \vdash x : \text{int}} \quad \frac{}{\{x : \text{int}\} \vdash 0 : \text{int}} \\
 \frac{}{\{x : \text{int}\} \vdash x = 0 : \text{bool}} \quad \frac{}{\{x : \text{int}\} \vdash \text{true} : \text{bool}} \quad \frac{}{\{x : \text{int}\} \vdash \text{false} : \text{bool}} \\
 \frac{}{\{x : \text{int}\} \vdash \text{if } x = 0 \text{ then true else false} : \text{bool}} \\
 \frac{}{\{\} \vdash \text{fun } x \rightarrow \text{if } x = 0 \text{ then true else false} : \text{int} \rightarrow \text{bool}} \quad \frac{}{\{\} \vdash 3 : \text{int}} \\
 \frac{}{\{\} \vdash (\text{fun } x \rightarrow \text{if } x = 0 \text{ then true else false}) 3 : \text{bool}}
 \end{array}$$

(b) Show the complete type derivation for the following term:

let x = 5 in g x

where g is assigned type $g : \text{int} \rightarrow \text{string}$

$$\begin{array}{c}
 \frac{}{\{g : \text{int} \rightarrow \text{string}, x : \text{int}\} \vdash g : \text{int} \rightarrow \text{string}} \quad \frac{}{\{g : \text{int} \rightarrow \text{string}, x : \text{int}\} \vdash x : \text{int}} \\
 \frac{}{\{g : \text{int} \rightarrow \text{string}\} \vdash 5 : \text{int}} \quad \frac{}{\{g : \text{int} \rightarrow \text{string}, x : \text{int}\} \vdash g x : \text{string}} \\
 \frac{}{\{g : \text{int} \rightarrow \text{string}\} \vdash \text{let } x = 5 \text{ in } g x : \text{string}}
 \end{array}$$

3. Unification

- (a) Give a most general unifier for the following unification problem. Lower case letters (f, g, h, d) are constants or term constructors: specifically, f and g are term constructors with arity 1, h is a term constructor with arity 2, and d is a constant term with arity 0. Letters α, β , and γ are variables. Show your work by listing the operation performed in each step of unification – decompose, orient, delete, eliminate – and the result of the step. If unification is not possible, work as far as possible and show where unification fails. If unification does not fail, show the final substitution, which should be a set of variable to term mappings.

$$\{f(\alpha) = f(g(\gamma)); \gamma = d; \alpha = g(\beta); h(\alpha, \gamma) = h(\alpha, \beta)\}$$

| | |
|-----------|---|
| eliminate | $\{f(\alpha) = f(g(d)); \gamma = d; \alpha = g(\beta); h(\alpha, d) = h(\alpha, \beta)\}$ |
| decompose | $\{\alpha = g(d); \gamma = d; \alpha = g(\beta); h(\alpha, d) = h(\alpha, \beta)\}$ |
| decompose | $\{\alpha = g(d); \gamma = d; \alpha = g(\beta); \alpha = \alpha; d = \beta\}$ |
| delete | $\{\alpha = g(d); \gamma = d; \alpha = g(\beta); d = \beta\}$ |
| orient | $\{\alpha = g(d); \gamma = d; \alpha = g(\beta); \beta = d\}$ |
| eliminate | $\{\alpha = g(d); \gamma = d; \alpha = g(d); \beta = d\}$ |
| delete | $\{\alpha = g(d); \gamma = d; \beta = d\}$ |
| | |
| FINAL | $\{\alpha = g(d); \gamma = d; \beta = d\}$ |

- (b) Using the unifier discovered above, apply the substitution and show the final terms, which should be equalities and which should have no variables. If unification got stuck above, just write “unification failed” below.

- $f(\alpha) = f(g(\gamma)) \longrightarrow f(g(d)) = f(g(d))$
- $\gamma = d \longrightarrow d = d$
- $\alpha = g(\beta) \longrightarrow g(d) = g(d)$
- $h(\alpha, \gamma) = h(\alpha, \beta) \longrightarrow h(g(d), d) = h(g(d), d)$

4. λ -Calculus

- (a) Write the
- λ
- calculus version of
- `or`
- . You will need the following definitions:

$$\begin{aligned}\text{True} &\equiv \lambda x.\lambda y.x \\ \text{False} &\equiv \lambda x.\lambda y.y \\ \text{if_then_else} &\equiv \lambda b\ c\ d.b\ c\ d\end{aligned}$$

Your intuition should be the following: when taking the logical or of two values, if the first value is true, the result will always be true. If the first value is false, then the result will match the second value – if it is true, the or is true, if it is false, the or is false. Using that, we can write or as:

$$\text{or} \equiv \lambda a\ b.a\ \text{True}\ b$$

You can also think of this using the `if_then_else` construct. In that case, you would want to say `if a then True else b`, where `a` and `b` are the two booleans. Substituting in for `if_then_else`, we get the same result (technically, we get `a True b`, but if we lambda-lift `a` and `b`, we get the same result as above).

- (b) Write a function that, given a pair
- `p`
- , will return pair
- `q`
- such that the first element of
- `p`
- is the second of
- `q`
- and the second of
- `p`
- is the first of
- `q`
- (i.e. that flips the two elements of the pair). You will need the following definitions:

$$\begin{aligned}\text{pair}\ a,\ b &\equiv \lambda a\ b\ x.x\ a\ b \\ \text{fst}\ p &\equiv \lambda p.p(\lambda x\ y.x) \\ \text{snd}\ p &\equiv \lambda p.p(\lambda x\ y.y)\end{aligned}$$

If we think of this outside of λ calculus, we would want something like

$$\text{flip}\ p = (\text{snd}\ p,\ \text{fst}\ p)$$

We can encode that with λ terms as

$$\text{flip} \equiv \lambda p.\text{pair}\ (\text{snd}\ p)(\text{fst}\ p)$$

Substituting in for our “macros”, we get

$$\text{flip} \equiv \lambda p.(\lambda a\ b\ x.x\ a\ b)((\lambda p.p(\lambda x\ y.y))\ p)((\lambda p.p(\lambda x\ y.x))\ p)$$

We can then β reduce it a bit

$$\text{flip} \equiv \lambda p.(\lambda a\ b\ x.x\ a\ b)(p(\lambda x\ y.y))(p(\lambda x\ y.x))$$

And a bit more, to get our final answer

$$\text{flip} \equiv \lambda p.(\lambda x.x(p(\lambda x\ y.y)))(p(\lambda x\ y.x))$$

- (c) Using the definition of plus for Church numerals, show the result of adding the Church numeral for 1 to itself – show all β and α reduction steps needed, and use a lazy evaluation order.

$$1 \equiv \lambda f x. f x$$

$$\text{plus} \equiv \lambda n m f x. n f (m f x)$$

$$(\lambda n m f x. n f (m f x))(\lambda f x. f x)(\lambda f x. f x) \rightarrow^{\beta}$$

$$\lambda f x. (\lambda f x. f x) f ((\lambda f x. f x) f x) \rightarrow^{\beta}$$

That's all the further we can go with a lazy order, since we are now in a situation where everything is under the top-most lambda. If we wanted to reduce further, starting from where we stopped above we could continue as follows. **If you stopped above in your solution, I would encourage you to work out the remainder of the reduction before looking at the answer below.**

$$\lambda f x. (\lambda f x. f x) f ((\lambda f x. f x) f x) \rightarrow^{\beta}$$

$$\lambda f x. (\lambda x. f x)((\lambda f x. f x) f x) \rightarrow^{\beta}$$

$$\lambda f x. f ((\lambda f x. f x) f x) \rightarrow^{\beta}$$

$$\lambda f x. f ((\lambda x. f x) x) \rightarrow^{\beta}$$

$$\lambda f x. f (f x)$$

This gives us 2, like we would expect it to.

- (d) Using the function to flip the elements of pairs defined above, illustrate the use of this function on the pair $\lambda x. x c d$ (here c and d could be any two terms, but we don't need to look inside them so we can just abstract them away). You may use any evaluation order.

Just to recall, our flip function wound up being:

$$\text{flip} \equiv \lambda p. (\lambda x. x(p(\lambda x y. y)))(p(\lambda x y. x))$$

Now,

$$(\lambda p. (\lambda x. x(p(\lambda x y. y)))(p(\lambda x y. x)))(\lambda x. x c d) \rightarrow^{\beta}$$

$$(\lambda x. x((\lambda x. x c d)(\lambda x y. y))((\lambda x. x c d)(\lambda x y. x))) \rightarrow^{\beta}$$

$$(\lambda x. x((\lambda x y. y) c d)((\lambda x y. x) c d)) \rightarrow^{\beta}$$

$$(\lambda x. x((\lambda y. y) d)((\lambda x y. x) c d)) \rightarrow^{\beta}$$

$$(\lambda x. x d((\lambda x y. x) c d)) \rightarrow^{\beta}$$

$$\lambda x. x d((\lambda y. c) d) \rightarrow^{\beta}$$

$$\lambda x. x d c$$

5. Regular Expressions

- (a) Given the alphabet $\{x, y, z\}$, provide a regular expression that defines all strings ending in z that have at least one y

The intuition here is that we have a y embedded somewhere in the string, and a z at the end. The y may be first, with anything between it and the z , or right before the final z , or somewhere in the middle. Thus, 0 or more characters are before the y , and 0 or more are after it, but we definitely have at least 1 y .

$$(x + y + z)^*y(x + y + z)^*z$$

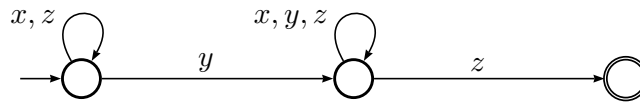
- (b) Given the alphabet $\{a, b\}$, provide a regular expression that defines all strings of a 's and b 's that start and end with the same letter.

Here we know that the first and last letters are the same, but we don't care what's in the middle. We will need two cases, then – all strings starting and ending with a with whatever between the a 's, and all strings starting and ending with b with whatever between the b 's.

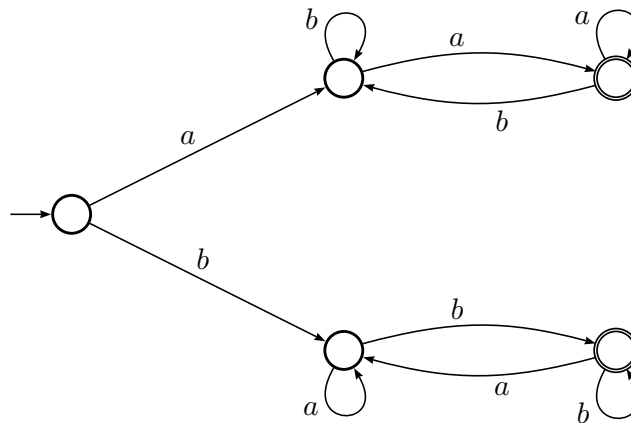
$$(a(a + b)^*a) + (b(a + b)^*b)$$

6. DFAs and NFAs

- (a) Given the alphabet $\{x, y, z\}$, define an NFA that recognizes all strings ending in z that have at least one y



- (b) Given the alphabet $\{a, b\}$, define a DFA that recognizes all strings of a 's and b 's that start and end with the same letter.



Rules for type derivations:**Constants**

$$\frac{}{\vdash n : \text{int}} \text{(assuming } n \text{ is an int)}$$

$$\frac{}{\vdash \text{true} : \text{bool}}$$

$$\frac{}{\vdash \text{false} : \text{bool}}$$

Variables

$$\frac{}{\Gamma \vdash x : \tau} \text{if } (x : \tau) \in \Gamma$$

Arithmetic Operators

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \oplus e_2 : \text{int}} \quad (\oplus \in \{+, -, *, /, \dots\})$$

Relational Operators

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \sim e_2 : \text{bool}} \quad (\sim \in \{<, >, \leq, \geq, =, \neq, \dots\})$$

Booleans

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \ \&\& \ e_2 : \text{bool}}$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \ \|\| \ e_2 : \text{bool}}$$

$$\frac{\Gamma \vdash e_1 : \text{bool}}{\Gamma \vdash ! e_1 : \text{bool}}$$

If

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}$$

Function Abstraction

$$\frac{\Gamma \cup [x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2}$$

Function Application

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \ e_2 : \tau_2}$$

Let

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \cup [x : \tau] \vdash e_2 : \tau'}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau'}$$

Letrec

$$\frac{\Gamma \cup [x : \tau] \vdash e_1 : \tau \quad \Gamma \cup [x : \tau] \vdash e_2 : \tau'}{\Gamma \vdash \text{let rec } x = e_1 \text{ in } e_2 : \tau'}$$