Midterm Examination

CAS CS 320: Principles of Programming Languages

February 27, 2025

Name: Nathan Mull

BUID: 12345678

- ▷ You will have approximately 75 minutes to complete this exam. Make sure to read every question, some are easier than others.
- ▷ Do not remove any pages from the exam.
- ▷ Make very clear what your final solution for each problem is (e.g., by surrounding it in a box). We reserve the right to mark off points if we cannot tell what your final solution is.
- ▷ For all coding problems, you may write helper functions.
- ▷ Unless stated otherwise, you should only need the rules provided **in that problem** for your derivations.
- ▷ We will not look at any work on the pages marked "*This page is intentionally left blank.*" You should use these pages for scratch work.

Problem #	Points
1	8
2	10
3	5
4	15

This page is intentionally left blank.

1 Tail Recursion

(8 points) Without using any functions in the standard library (except for basic constructors like `(::)` and `[]`), implement the function

```
val split_by : ('a -> bool) -> 'a list -> 'a list * 'a list
```

so that `split_by f l` is a pair of lists `(lt, lf)` where `lt` consists of those elements of `l` (in order, with repetitions) which satisfy the predicate `f`, and `lf` consists of those elements which do not satisfy the predicate `f`. **Your solution must be tail recursive.**

```
let split_by f l
  let rev acc l =
    match l with
    | [] -> acc
    | x::xs -> rev (x::acc) xs
  in
  let rec go (acc1, acc2) l =
    match l with
    | [ ] -> (rev acc1, rev acc2)
    | x :: xs ->
        if f x
        then go (x :: acc1, acc2) xs
        else go (acc1, x:: acc2) xs
  in go ([], []) l
```

```
let _ = assert (split_by (fun x -> x > 3) [0;5;4;2;1;4]
                = ([5;4;4], [0;2;1]))
let _ = assert (split_by (fun x -> x = 2 || x = 4) [1;2;3;4;5;6]
                = ([2;4], [1;3;5;6])
```

(Solution Continued)

2 S-Expressions

Recall the type `'a ntree` from `stdlib320`.

```
type 'a ntree = Node of 'a * 'a ntree list
```

In lab 4, we implemented a parser for S-expressions with the following type.

```
val parse : string -> string ntree option
```

The issue is that using nonempty trees made it impossible to parse S-expressions like `((foo) bar)`, where the first element was another S-expression. Alternatively, we could write a parser with the type:

```
val parse' : string -> string sexpr option
```

so that it passes a test like:

```
let _ = assert (parse' "((foo) bar)"
                = Some (List [List [Atom "foo"]; Atom "bar"])
```

In this problem, we'll be looking at conversions between between this better-suited type `'a sexpr` and nonempty trees. Consider the following code. You should read through it and try to understand what it's doing.

```
let all (l : 'a option list) : 'a list option = assert false (* PART B *)

let ntree_of_sexpr (e : 'a sexpr) : 'a ntree option =
  let rec go (e : 'a sexpr) : 'a ntree option =
    match e with
    | Atom a -> Some (a, [])
    | List (Atom a :: es) -> (
      match all (List.map go es) with
      | None -> None
      | Some ts -> Node (a, ts)
    )
    | _ -> None
  in go e

let sexpr_of_ntree (t : 'a ntree) : 'a sexpr = assert false (* PART C *)

let _ = assert (ntree_of_expr (List [Atom "foo"; Atom "bar"; Atom "baz"])
                = Some (Node ("foo", [Node "bar" []; Node "bar" []])))
let _ = assert (ntree_of_expr (List [List [Atom "foo"]; Atom "bar"])
                = None)
let _ = assert (ntree_of_expr (List [List [List []]]) = None)
let _ = assert (ntree_of_expr (List [Atom "one"; List [Atom "two"]])
                = Some (Node ("one", [Node "two" []])))
```

The problem continues on the following pages, and refer to the code above.

A. (3 points) There is enough information in the given code to determine the structure of the type `'a sexpr`, assuming all pattern matches are exhaustive. Write down the OCaml ADT definition for the type `'a sexpr`.

```
type 'a sexpr =
  | Atom of 'a
  | List of 'a sexpr list
```

Note: There was a typo in the problem statement which allowed for any additional constructors, we accepted this without penalty

B. (7 points) The function `ntree_of_sexpr` depends on the function

```
val all : 'a option list -> 'a list option
```

which converts a list of options into an optional list according to the following specification.

- ▷ If `None` is an element of the list `l`, then `all l` is `None`.
- ▷ Otherwise, `all l` is `Some l'` where `l'` is a list of the arguments of the `Some` elements of `l`.

Without using any functions from the standard library (except for basic constructors like `Some` and `None` and `(::)` and `[]`), implement this function.

```
let rec all l =
    match l with
    | [] -> Some []
    | Some x :: xs -> (
      match all xs with
      | None -> None
      | Some l -> Some (x :: l)
    )
    | None :: _ -> None
```

```
let _ = assert (all [Some 1; Some 2; Some 3] = Some [1;2;3])
let _ = assert (all [Some 1; None; Some 3] = None)
```

C. **Extra Credit.** (2 points) Implement the function

```
val sexpr_of_ntree : 'a ntree -> 'a sexpr
```

So that it is a (partial) inverse of `ntree_of_sexpr`. You may use any functions in the standard library.

```
let sexpr_of_ntree =
  let rec go (Node (a, ts)) =
    match ts with
    | [] -> Atom a
    | l -> List (Atom a :: List.map go l)
  in go
```

3 Double Fold

(5 points) Suppose we wanted to fold multiple functions over the same data. We could imagine a function with the following signature.

```
val fold_left2 :
  ('acc1 -> 'a -> 'acc1) -> ('acc2 -> 'a -> 'acc2)
  -> 'acc1 -> 'acc2
  -> 'a list -> 'acc1 * 'acc2
```

where `fold_left2 f g b1 b2 l` is equivalent to `(List.fold_left f b1 l, List.fold_left g b2 l)`. We can implement this function with a *single* call to `List.fold_left`:

```
let fold_left2 f g b1 b2 l =
  let combine = assert false (* TODO *) in
  let b = assert false (* TODO *) in
  List.fold_left combine b l
```

Using the skeleton above, implement the function `fold_left2`. **Reproduce the entire definition, not just the incomplete parts**. Recall that the type of `List.fold_left` is:

```
('acc -> 'a -> 'acc) -> 'acc -> 'a list -> 'acc
```

```
let fold_left2 f g b1 b2 l =
  let combine = fun (acc1 , acc2) a -> (f acc1 a, g acc2 a) in
  let b = (b1 ,b2) in
  List.fold_left combine b l
```

This page is intentionally left blank.

4 Optional Binding

The programming language Swift has a feature called *optional binding*, which allows us to conditionally bind the inner value of an option. In this problem, we'll be looking at typing and semantic rules for a variant of this. We introduce `let?`-expressions, a version of `let`-expressions specialized to options. Make sure to read the rules carefully. The new `let?`-expressions are similar to `let`-expressions, but not identical.

A. (5 points) Consider the following typing rules, which include some rules from `320Caml` and some new rules. Note that side-conditions are highlighted, not colored (you should see a grey box around the side-conditions).

$$\frac{(v : \tau) \in \Gamma}{\Gamma \vdash v : \tau}\,(\text{var}) \qquad \frac{\texttt{n} \text{ is an integer literal}}{\Gamma \vdash n : \texttt{int}}\,(\text{int-lit}) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}\,(\text{int-add})$$

$$\frac{}{\Gamma \vdash \texttt{None} : \tau \texttt{ option}}\,(\text{none}) \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{Some } e : \tau \texttt{ option}}\,(\text{some})$$

$$\frac{\Gamma \vdash e_1 : \tau' \texttt{ option} \qquad \Gamma, x : \tau' \vdash e_2 : \tau \texttt{ option}}{\Gamma \vdash \texttt{let? } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau \texttt{ option}}\,(\text{let-option})$$

Write a derivation of the following typing judgment. You may shorthand rule names, as long as it is clear which rule you are referring to.

$$\varnothing \vdash \texttt{let? x = Some 2 in Some (x + 2)} : \texttt{int option}$$

$$\frac{\dfrac{\dfrac{}{\varnothing \vdash 2 : int}\,(\text{int-lit})}{\varnothing \vdash \text{Some } 2 : int \text{ option}}\,(\text{some}) \qquad \dfrac{\dfrac{\dfrac{}{\{x : int\} \vdash x : int}\,(\text{var}) \qquad \dfrac{}{\{x : int\} \vdash 2 : int}\,(\text{int-lit})}{\{x : int\} \vdash x + 2 : int}\,(\text{int-add})}{\{x : int\} \vdash \text{Some } (x+2) : int \text{ option}}\,(\text{some})}{\varnothing \vdash \text{let? } x = \text{Some } 2 \text{ in Some } (x+2) : int \text{ option}}\,(\text{let-option})$$

(Solution Continued)

B. (5 points) Considering the following semantic rules. Note that we introduce an option value for the result of evaluating an option.

$$\frac{\mathsf{n}\text{ is an integer literal}}{\mathsf{n} \Downarrow n}\;\text{(int-lit-eval)} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 + v_2 = v}{e_1\ \texttt{+}\ e_2 \Downarrow v}\;\text{(int-add-eval)}$$

$$\frac{}{\texttt{None} \Downarrow \mathsf{None}}\;\text{(none-eval)} \qquad \frac{e \Downarrow v}{\texttt{Some}\ \ e \Downarrow \mathsf{Some}(v)}\;\text{(some-eval)}$$

$$\frac{e_1 \Downarrow \mathsf{Some}(v_1) \qquad e' = [v_1/x]e_2 \qquad e' \Downarrow v}{\texttt{let?}\ \ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Downarrow v}\;\text{(let-option-some-eval)}$$

$$\frac{e_1 \Downarrow \mathsf{None}}{\texttt{let?}\ \ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Downarrow \mathsf{None}}\;\text{(let-option-none-eval)}$$

Write a derivation of the following semantic judgment.

$$\texttt{let? x = Some 2 in let? y = Some 3 in Some (x + y)} \Downarrow \mathsf{Some}(5)$$

$$\dfrac{\dfrac{\dfrac{}{2 \Downarrow 2}\,\text{(int-lit-eval)}}{\text{Some } 2 \Downarrow \text{Some}(2)}\,\text{(some-eval)} \qquad \dfrac{\dfrac{\dfrac{}{3 \Downarrow 3}\,\text{(int-lit-eval)}}{\text{Some } 3 \Downarrow \text{Some}(3)}\,\text{(some-eval)} \qquad \dfrac{\dfrac{\dfrac{}{2 \Downarrow 2}\,\text{(int-lit-eval)} \quad \dfrac{}{3 \Downarrow 3}\,\text{(int-lit-eval)}}{2+3 \Downarrow 5}\,\text{(int-add-eval)}}{\text{Some}(2+3) \Downarrow \text{Some}(5)}\,\text{(some-eval)}}{\text{let? } y = \text{Some } 3 \text{ in Some}(2+y) \Downarrow \text{Some}(5)}\,\text{(let-option-some-eval)}}{\text{let? } x = \text{Some } 2 \text{ in let? } y = \text{Some } 3 \text{ in Some } (x+y) \Downarrow \text{Some}(5)}\,\text{(let-option-some-eval)}$$

(Solution Continued)

C. (5 points) These are the same rules as in the previous part, reproduced for convenience.

$$\frac{\text{n is an integer literal}}{\mathsf{n} \Downarrow n}\ (\text{int-lit-eval}) \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 + v_2 = v}{e_1\ \texttt{+}\ e_2 \Downarrow v}\ (\text{int-add-eval})$$

$$\frac{}{\texttt{None} \Downarrow \mathsf{None}}\ (\text{none-eval}) \qquad \frac{e \Downarrow v}{\texttt{Some}\ \ e \Downarrow \mathsf{Some}(v)}\ (\text{some-eval})$$

$$\frac{e_1 \Downarrow \mathsf{Some}(v_1) \qquad e' = [v_1/x]e_2 \qquad e' \Downarrow v}{\texttt{let?}\ \ x\ \texttt{=}\ e_1\ \ \texttt{in}\ \ e_2 \Downarrow v}\ (\text{let-option-some-eval})$$

$$\frac{e_1 \Downarrow \mathsf{None}}{\texttt{let?}\ \ x\ \texttt{=}\ e_1\ \ \texttt{in}\ \ e_2 \Downarrow \mathsf{None}}\ (\text{let-option-none-eval})$$

Determine the value v such that the following semantic judgment is derivable, and then write its derivation.

$$\texttt{let?}\ \ \texttt{x = Some 2 in let?}\ \ \texttt{y = None in Some (x + y)} \Downarrow v$$

$$\dfrac{\dfrac{\dfrac{}{2 \Downarrow 2}\ (\text{int-lit-eval})}{\text{Some } 2 \Downarrow \text{Some } (2)}\ (\text{some-eval}) \qquad \dfrac{\dfrac{}{\text{None} \Downarrow \text{None}}\ (\text{none-eval})}{\text{let? } y = \text{None in Some}(2+y) \Downarrow \text{None}}\ (\text{let-option-none-eval})}{\text{let ? } x = \text{Some } 2 \text{ in let ? } y = \text{None in Some } (x+y) \Downarrow \text{None}}\ (\text{let-option-some-eval})$$

(Solution Continued)