1 OCaml Basics

- Suppose we have a type intlist defined as:

  type intlist = Nil | Cons of int * intlist

  Write a function evens : intlist -> int that takes a list l and returns a list containing every other element of l, starting with the second. For example, evens (Cons (1, Cons (2, Cons (3, Cons (4, Nil))))) should return Cons (2, Cons (4, Nil)).

  let rec evens (l : intlist) : int =
    match l with
    | Nil -> Nil
    | Cons (i, rest) -> (match rest with
          | Nil -> Nil
          | Cons (i2, rest2) -> Cons (i2, evens rest2))

2 Grammars and ASTs

Consider the following context-free grammar:

\[
D ::= \text{North} \mid \text{East} \mid \text{South} \mid \text{West}
\]

\[
A ::= \langle \text{num} \rangle \ D \ \langle \text{ident} \rangle \ \text{Street} \mid \langle \text{num} \rangle \ D \ \langle \text{ident} \rangle \ \text{Avenue}
\]

- Draw the abstract syntax tree in this grammar for the term 623 West Flournoy Street.
Write OCaml datatypes representing the ASTs for $D$ and $A$, where $\langle\text{num}\rangle$ is represented by the int type and $\langle\text{ident}\rangle$ is represented by the string type.

```ocaml
type ast_D = North | East | South | West

type ast_P = St of int * ast_D * string | Ave of int * D * string
```

Write the OCaml value corresponding to the AST for the term 623 West Flournoy Street.

`St (623, West, "Flournoy")`

## 3 Type Systems

The type system for a simple imperative language is given in Appendix A.

Given a type context $\Gamma$ such that $\Gamma(x) = \text{int}$ and $\Gamma(y) = \text{bool}$, is the term

```
if x = y then false else true
```

type-correct? Why or why not?

It is not type-correct. The typing rule for $=$ requires that the left-hand side and right-hand side have the same type, which $x$ and $y$ do not.

Write the proof tree for the typing judgment described above. If it is not type-correct, indicate the place in the proof tree where a rule fails to apply or is impossible to complete.

```
\[
\frac{(\Gamma(x) = \text{int}) \quad (\Gamma(y) = \text{bool}) \quad \text{type mismatch}}{\Gamma \vdash y : \text{bool}} \quad \frac{\Gamma \vdash x = y : \text{bool}}{\Gamma \vdash \text{false} : \text{bool}} \quad \frac{\Gamma \vdash \text{true} : \text{bool}}{\Gamma \vdash \text{if x = y then false else true} : \text{bool}}
\]
```
• Suppose we were writing a typing function `type_of : context -> exp -> typ option` that takes a type context and an AST for an expression and returns its type, where `typ` is defined as `type typ = Tint | Tbool`. Fill in the indicated cases of the function below, by translating the corresponding typing rules into OCaml code.

```ocaml
let rec type_of (gamma : context) (e : exp) : typ option =
  match e with
  | Num n -> Some Tint
  | Var x -> gamma x
  | Add (e1, e2) -> (match type_of gamma e1, type_of gamma e2 with
    | Some Tint, Some Tint -> Some Tint
    | _, _ -> None)
  | If (e, e1, e2) -> (match type_of gamma e, type_of gamma e1, type_of gamma e2 with
    | Some Tbool, Some t1, Some t2 ->
      if t1 = t2 then Some t1 else None
    | _, _ -> None)
```

4 Operational Semantics

The hybrid-style operational semantics for a simple imperative language is given in Appendix B.

• Write the next configuration that

\[(i := 5; \text{while } i = 5 \text{ do } i := i - 1, \{\})\]

steps to.

\[(\text{skip; while } i = 5 \text{ do } i := i - 1, \{i = 5\})\]

• Write the proof tree for the step above.

\[
\frac{
  (5, \{\}) \Downarrow 5
}{
  (i := 5, \{\}) \rightarrow (\text{skip, } \{i = 5\})
}
\]

\[
\frac{
  (i := 5; \text{while } i = 5 \text{ do } i := i - 1, \{\}) \rightarrow
}{
  (\text{skip; while } i = 5 \text{ do } i := i - 1, \{i = 5\})
}
\]

• What is the next step that the resulting configuration takes?

\[(\text{skip; while } i = 5 \text{ do } i := i - 1, \{i = 5\}) \rightarrow (\text{while } i = 5 \text{ do } i := i - 1, \{i = 5\})\]
• Suppose we wanted to extend our simple imperative language with a command “do c until e” that executes c at least once, and continues to execute it until e is false at the end of an iteration. Give small-step semantics rules for “do c until e”.

\[
(\text{do } c \text{ until } e, \sigma) \rightarrow c; \text{if } e \text{ then (do } c \text{ until } e) \text{ else skip}
\]

• Values in our simple imperative language are either integers or booleans. Write a type value that represents values.

```ocaml
type value = IntV of int | BoolV of bool
```

• Suppose we were writing an interpreter function `eval_exp : exp -> state -> value option` that evaluates an expression to a value. Fill in the indicated cases of the function below, by translating the corresponding typing rules into OCaml code.

```ocaml
let rec eval_exp (e : exp) (s : state) : value option =
  match e with
  | Num n -> Some (IntV n)
  | Var x -> s x
  | Add (e1, e2) -> (match eval_exp e1 s, eval_exp e2 s with
    | Some (IntV i1), Some (IntV i2) -> Some (IntV (i1 + i2))
    | _, _ -> None)
  | If (e, e1, e2) -> (match eval_exp e s with
    | Some (BoolV b) ->
      if b then eval_exp e1 s else eval_exp e2 s
    | _ -> None)
```
A Typing Rules for a Simple Imperative Language

\[
\begin{align*}
\Gamma &\vdash n : \text{int} & \text{(n is a number)} \\
\Gamma &\vdash b : \text{bool} & \text{(b is a boolean)} \\
\Gamma &\vdash x : \tau & \text{(\(\Gamma(x) = \tau\))}
\end{align*}
\]

\[
\begin{align*}
\Gamma &\vdash e_1 : \text{int} & \Gamma &\vdash e_2 : \text{int} & \Gamma &\vdash e_1 \oplus e_2 : \text{int} & \text{where \(\oplus\) is an arithmetic operator}
\end{align*}
\]

\[
\begin{align*}
\Gamma &\vdash e_1 : \text{bool} & \Gamma &\vdash e_2 : \text{bool} & \Gamma &\vdash e_1 \otimes e_2 : \text{bool} & \text{where \(\otimes\) is a boolean operator}
\end{align*}
\]

\[
\begin{align*}
\Gamma &\vdash e_1 : \tau & \Gamma &\vdash e_2 : \tau & \Gamma &\vdash e_1 = e_2 : \text{bool} & \text{if \(e_1\) then \(e_1\) else \(e_2\) : \(\tau\)}
\end{align*}
\]

\[
\begin{align*}
\Gamma &\vdash e : \text{bool} & \Gamma &\vdash c_1 : \text{ok} & \Gamma &\vdash c_2 : \text{ok} & \Gamma &\vdash \text{while } e \text{ do } c : \text{ok}
\end{align*}
\]

B Operational Semantics for a Simple Imperative Language

\[
\begin{align*}
\text{(n is a number)} & & \text{(b is a boolean)} & & \text{(\(\sigma(x) = v\))}
\end{align*}
\]

\[
\begin{align*}
(n, \sigma) \Downarrow n & & (b, \sigma) \Downarrow b & & (x, \sigma) \Downarrow v
\end{align*}
\]

\[
\begin{align*}
(e_1, \sigma) \Downarrow v_1 & & (e_2, \sigma) \Downarrow v_2 & & (v_1 \oplus v_2 = v) & & \text{where \(\oplus\) is an arithmetic or boolean operator}
\end{align*}
\]

\[
\begin{align*}
(e, \sigma) \Downarrow \text{true} & & (e_1, \sigma) \Downarrow v & & (e, \sigma) \Downarrow \text{false} & & (e_2, \sigma) \Downarrow v
\end{align*}
\]

\[
\begin{align*}
(x := e, \sigma) \Downarrow (\text{skip}, \sigma[x \mapsto v]) & & \text{(if \(e\) then \(e_1\) else \(e_2, \sigma\)) \Downarrow v}
\end{align*}
\]

\[
\begin{align*}
(c_1, \sigma) \to (c_1', \sigma') & & \text{(skip;} c_2, \sigma) \to (c_2, \sigma)
\end{align*}
\]

\[
\begin{align*}
(c_1; c_2, \sigma) \to (c_1'; c_2, \sigma') & & \text{(if \(e\) then \(c_1\) else \(c_2, \sigma\)) \to (c_1, \sigma)}
\end{align*}
\]

\[
\begin{align*}
(e, \sigma) \Downarrow \text{true} & & (e, \sigma) \Downarrow \text{false}
\end{align*}
\]

\[
\begin{align*}
(\text{while } e \text{ do } c, \sigma) \to (\text{if } e \text{ then } (c; \text{while } e \text{ do } c) \text{ skip}, \sigma)
\end{align*}
\]