1 OCaml Basics

- Suppose we have a type intlist defined as:

  \[
  \text{type intlist} = \text{Nil} \mid \text{Cons of int * intlist}
  \]

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

2 Grammars and ASTs

Consider the following CFG:

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

- 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 $\text{int}$ type and $\langle \text{ident} \rangle$ is represented by the $\text{string}$ type.

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

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

$$\text{if } x = y \text{ then false else true}$$

type-correct? Why or why 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.
Suppose we were writing a typing function \( \text{type} \_\text{of} : \text{context} \rightarrow \text{exp} \rightarrow \text{typ} \text{ option} \) that takes a type context and an AST for an expression and returns its type, where \( \text{typ} \) is defined as
\[
\text{type} \ \text{typ} = \text{Tint} \mid \text{Tbool}
\]. 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 ->
  | Var x ->
  | Add (e1, e2) ->
  | If (e, e1, e2) ->
```

4 Operational Semantics

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

- Write the next configuration that

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

steps to.

- Write the proof tree for the step above.

- What is the next step that the resulting configuration takes?
• 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”.

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

• 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
define eval_exp (e : exp) (s : state) : value option =
  match e with
  | Num n ->
  | Var x ->
  | Add (e1, e2) ->
  | If (e, e1, e2) ->
```
A  Typing Rules for a Simple Imperative Language

\[
\begin{array}{ccc}
\text{Γ ⊢ } n : \text{int} & \text{Γ ⊢ } b : \text{bool} & \text{Γ ⊢ } x : \tau \\
\text{(n is a number)} & \text{(b is a boolean)} & \text{(Γ(x) = τ)} \\
\hline
\Gamma ⊢ e_1 : \text{int} & \Gamma ⊢ e_2 : \text{int} & \Gamma ⊢ e_1 ⊕ e_2 : \text{int} \\
\text{where } ⊕ \text{ is an arithmetic operator} \\
\hline
\Gamma ⊢ e_1 : \text{bool} & \Gamma ⊢ e_2 : \text{bool} & \Gamma ⊢ e_1 ⊗ e_2 : \text{bool} \\
\text{where } ⊗ \text{ is a boolean operator} \\
\hline
\Gamma ⊢ e_1 = e_2 : \text{bool} & \Gamma ⊢ e_1 : \tau & \Gamma ⊢ e_2 : \tau \\
\hline
\end{array}
\]

B  Operational Semantics for a Simple Imperative Language

\[
\begin{array}{ccc}
\text{(n is a number)} & \text{(b is a boolean)} & \text{(σ(x) = v)} \\
\hline
\text{(n, σ) ↓ n} & \text{(b, σ) ↓ b} & \text{(x, σ) ↓ v} \\
\hline
(e_1, σ) ↓ v_1 & (e_2, σ) ↓ v_2 & (v_1 ⊕ v_2 = v) \\
\text{where } ⊕ \text{ is an arithmetic or boolean operator} \\
\hline
(e, σ) ↓ \text{true} & (e_1, σ) ↓ v & \text{(e, σ) ↓ false} & (e_2, σ) ↓ v \\
\hline
\text{(if e then e_1 else e_2, σ) ↓ v} & \text{(e, σ) ↓ v} & \text{(if e then e_1 else e_2, σ) ↓ v} \\
\hline
\text{(x := e, σ) \rightarrow (\text{skip, } σ[x \mapsto v])} \\
\hline
\text{(c_1, σ) \rightarrow (c'_1, σ')} & \text{(c_1; c_2, σ) \rightarrow (c'_1; c_2, σ')} & \text{(\text{skip; c_2, } σ) \rightarrow (c_2, σ)} \\
\hline
\text{(e, σ) ↓ true} & \text{(e, σ) ↓ false} & \text{(e, σ) ↓ false} \\
\text{(if e then c_1 else c_2, σ) \rightarrow (c_1, σ)} & \text{(if e then c_1 else c_2, σ) \rightarrow (c_2, σ)} & \text{(if e then c_1 else c_2, σ) \rightarrow (c_2, σ)} \\
\hline
\text{(while e do c, σ) \rightarrow (if e then (c; while e do c) else skip, σ)} \\
\end{array}
\]