1 Instructions

Begin by downloading the file `hw4-base.ml` from the course website, and renaming it to `hw4.ml`. Then fill in your answers to the problems, adding or modifying definitions as you see fit. Submit your completed `hw4.ml` via Gradescope. As always, please don’t hesitate to ask for help on Piazza (https://piazza.com/class/ksknvq6ogb2kc).

2 Adding Functions to the Interpreter

The file `hw4-base.ml` defines the types `exp` of expressions and `cmd` of commands. It also defines two main functions: `eval_exp`, a big-step-style interpreter for expressions, and `step_cmd`, a small-step-style interpreter for commands.

A state maps identifiers to the entry type, which can be either `Val` of a value (an `IntVal`/`BoolVal`) or a function definition `Fun`, which contains the list of parameter names and the function body. For instance, if `lookup s "f"` returns `Some (Fun (["x"; "y"], Return (Var "x")))`, this means that in the state `s`, the function `f` is defined as `f(x, y) { return x }`.

The function `eval_exp : exp -> state -> value option` takes an expression `e` and a state `s` and returns either `Some v`, if `e` evaluates to `v`, or `None`, if `e` fails to evaluate. The function `step_cmd : cmd -> stack -> state -> config option` takes a command `c`, a stack `k`, a state `s` and returns either:

- `Some (c', k', s')`, if `(c, k, s) \rightarrow (c', k', s')`
- `None`, if there is no step that `(c, k, s)` can take

The `cmd` type already includes constructors for function calls and returns. `Call` takes two identifiers, representing the variable and function name, and a list of expressions, representing the arguments. For example, `Call ("x", "f", [Num 1, Num 2])` represents the command `x := f(1, 2)`. `Return` takes one expression, which computes the return value of the function. Your job is to extend the `step_cmd` function to implement these commands.

1. (5 points) Add a case to `step_cmd` for the return statement, according to the following rule:

\[
\frac{(e, \sigma) \Downarrow v}{(\text{return } e, (\sigma_0, x) :: k, \sigma) \rightarrow (\text{skip}, k, \sigma_0[x \mapsto v])}
\]
You can test your code with the `run_config` function, which takes a `config` and applies `step_cmd` to the `config` for as many steps as possible. For instance, you could run the command

```ml
let (res_c, res_k, res_s) = run_config fd1 config1;;
```

to store the resulting command, stack, and state in variables `res_c`, `res_k`, and `res_s` respectively. (In this case, `res_c` should be `Skip` and `res_k` should be `[]`.) You can then look up variables in `res_s` to see whether the right state was produced:

```ml
lookup res_s "x";;
- : value option = Some (Val (IntVal 3))
```

2. (10 points) Add a case to `step_cmd` for the call statement, according to the following rule:

\[
\frac{([e_1; \ldots; e_n], \sigma) \Downarrow [v_1; \ldots; v_n]}{(x := f(e_1, \ldots, e_n), k, \sigma) \rightarrow (c, (\sigma, x) :: k, \sigma[x_1 \mapsto v_1; \ldots; x_n \mapsto v_n])}
\]

A function `add_args` has been provided that takes a state `s`, a list of variables, and a list of values, and returns the new state `s[x_1 \mapsto v_1; \ldots; x_n \mapsto v_n]`. There is also a function `eval_exps` that takes a list of expressions and returns the list of their values.

You can test your code with `run_config`, or use `run_prog`, which runs an entire program starting from an initial state. If you have correctly defined all commands, then `run_prog prog1 state0` should result in the command `Skip`, the stack `[]`, and a state in which `x` is 3 and `y` is undefined (i.e., `None`).