Name: ID#: X500: @umn.edu **A**

CS 2041: Practice Final SOLUTION

Fall 2018
University of Minnesota

Exam period: 20 minutes
Points available: 40

Background: OCaml's standard library has mutable, polymorphic hash table implementation which maps keys to values in the `Hashtbl` module which is demonstrated in a REPL nearby. Like the tree maps we created, `Hashtbl` provides higher-order functions for operating on the key/value associations in the map.

```
# let table = Hashtbl.create 20;;
# Hashtbl.add table "Goku"     8001;;
# Hashtbl.add table "Krillin"  1770;;
# Hashtbl.add table "Piccolo"  3500;;
# Hashtbl.add table "Vegeta"  18000;;

# let kpower = Hashtbl.find_opt table "Krillin";;
val kpower : int option = Some 1770
# let gpower = Hashtbl.find_opt table "Gohan";;
val gpower : int option = None

# #use "hash_funcs.ml";;
val print_all : (string, int) Hashtbl.t -> unit = <fun>
val total_power : ('a, int) Hashtbl.t -> int = <fun>

# print_all table;;     (* demo print_all *)
Krillin -> 1770
Vegeta -> 18000
Piccolo -> 3500
Goku -> 8001
- : unit = ()

# total_power table;;   (* demo total_power  *)
- : int = 31271
```

Problem 1 (5 pts): Write a function `print_all` which prints out all key/value bindings in a hash table of string/integers. Use the higher-order function `Hashtbl.iter func tbl` where `func` is passed keys and values from the hash table and returns `unit`. It is demonstrated in the REPL session.

SOLUTION

```
let print_all table =
  let print k v =
    printf "%s -> %d\n" k v
  in
  Hashtbl.iter print table;
;;
```

Problem 2 (5 pts): Write a function `total_power` which totals the values stored in a hash table with integer values. Use the higher-order function `Hashtbl.fold func tbl initial` where `func` is passed keys, values, and a running total. It is demonstrated in the REPL session.

SOLUTION

```
let total_power table =
  let total k v sum =
    sum+v
  in
  Hashtbl.fold total table 0
;;
```

Problem 3 (5 pts): A5's Calculon drew a distinction between a lambda expression and a closure. Describe the similarities and differences between these two things.

SOLUTION: The parser produces lambda expressions which are part of the parse tree. They have a parameter and a body of code to execute. The evaluator produces a Closure which is a variant of `data_t` *like* `IntDat` *and* `BoolDat`*. Closure's also have a parameter and code to evaluate when applied but add a variable map which tracks all variables that were defined at the time the Closure was created.*

Name:

Problem 4 (10 pts): To the right is a program which makes use of lazy evaluation. Show what you expect the output for the program to be below. **Justify your answer** by describing when and how many times various outputs are printed.

```
 1 open Printf;;
 2
 3 let _ =
 4   let exprA = lazy (printf "eval exprA\n";  5) in
 5   let exprB =      (printf "eval exprB\n"; 10) in
 6   let exprC = lazy (printf "eval exprC\n"; 15) in
 7
 8   printf "AB: %d\n"
 9     ((Lazy.force exprA) +           exprB );
10   printf "AC: %d\n"
11     ((Lazy.force exprA) + (Lazy.force exprC));
12   printf "BC: %d\n"
13     (           exprB  + (Lazy.force exprC));
14 ;;
```

SOLUTION

```
> ocamlc lazy_eval.ml
> a.out
eval exprB
eval exprA
AB: 15
eval exprC
AC: 20
BC: 25
```

exprB is not lazy so it will print immediately on reaching line 5. To print AB, exprA must be forced which executes its printing on line 9. The output for AB: 15 is then seen. Once forced, exprA will not print again so at line 11, only exprC is printed, then AC: 20. Finally, line 13 will not print exprC again as it has already been printed so only BC: 25 is seen.

Problem 5 (5 pts): Write a function `constantly x` which creates an infinite stream which always returns the given value `x`. The function is demonstrated in the REPL session below.

```
# #use "constantly.ml";;
val constantly : 'a -> 'a Stream.t = <fun>
# let ones = constantly 1;;
val ones : int Stream.t = <abstr>
# Stream.next ones;;
- : int = 1
# Stream.next ones;;
- : int = 1
# let mines = constantly "mine";;
val mines : string Stream.t = <abstr>
# Stream.next mines;;
- : string = "mine"
# Stream.next mines;;
- : string = "mine"
```

SOLUTION

```
let constantly n =
  let help i = Some n in
  Stream.from help
;;
```

Problem 6 (10 pts): Describe how string data may be added to A5's Calculon language interpreter. Included in this addition would be string concatenation via the ~ operator shown below. Make sure to describe which parts of Calculon would need to be altered.

```
calculon> parsetree "hello";
Parse tree:
StrExp("hello")

calculon> def str = "hello";
str : StrDat("hello")

calculon> def hw = str ~ " world";
hw : StrDat("hello world")
```

SOLUTION: The lexer in `calclex.ml` *would need to be modified to recognize the " character as the start of a string then scan ahead to find another ". The characters between would become the string. The* `parse_ident` *function in* `calcparse.ml` *needs a string expression kind and* `calceval.ml` *needs to add* `StrDat` *to its* `data_t` *type. To support concatenation add the ~ to the parser as a token and then add in a* `StringOp` *kind in the parser similar to the* `IntOp` *kind. Evaluation would simply concatenate the two strings together to produce a new, larger* `StrDat`*.*