CS301 Compiler Design and Implementation **Handout # 6**
Prof. Lyn Turbak **Friday, September 12**
Wellesley College

Problem Set 1
Due: September 6, 2003

Reading:

- Handout #7: CVS. (Follow the steps in this handout to create a local `cs301` subdirectory in your Linux home directory.
- Handout #8: OCaml Tutorial

Submission:
Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by 5pm on the due date. The packet should include:

1. a team header sheet (see the end of this assignment for the header sheet);
2. a written solution to Problem 1;
3. a written solution to Problem 2;
4. your final version of `MSort.ml` from Problem 3.
5. your final version of `RBT.ml` from Problem 4.

Problem 1 [10]: Interpreters and Compilers

a. **[5]** Suppose you are given the following:

- a Java-to-C-in-Scheme compiler (that is, a Java-to-C compiler written in Scheme.)
- a Scheme-in-Pentium interpreter.
- a C-to-Pentium-in-Pentium compiler.
- a Pentium-based computer.

Describe a sequence of steps you can perform to execute a given Java program on the computer.

b. **[5]** Suppose you are given the following:

- a OCaml-in-MIPS interpreter.
- a OCaml-to-MIPS-in-OCaml compiler
- a MIPS-based computer

i Describe how to generate a OCaml-to-MIPS-in-MIPS compiler.

ii After you successfully complete part i, you accidentally delete the OCaml-in-MIPS interpreter. Describe how you can still execute any OCaml program on your MIPS-based computer.

Problem 2 [20]: OCaml **Types**

Figs. 1–2 contain twenty higher-order OCaml functions. For each function, write down the type that would be automatically reconstructed for it. For example, consider the following OCaml `length` function:

```
let length xs =
  match xs with
    [] -> 0
  | (_::xs') = 1 + (length xs')
```

The type of this OCaml function is:

```
'a list -> int
```

Note: you can check your answers by typing them into the OCaml interpreter. But please write down the answers first before you check them — otherwise you will not learn anything!

```
let id x = x

let compose f g x = (f (g x))

let rec repeated n f =
      if (n = 0) then id else compose f (repeated (n - 1) f)

let uncurry f (a,b) = (f a b)

let curry f a b = f(a,b)

let churchPair x y f = f x y

let rec generate seed next isDone =
  if (isDone seed) then
    []
  else
    seed :: (generate (next seed) next isDone)

let rec map f xs =
  match xs with
    [] -> []
  | (x::xs') -> (f x) :: (map f xs')

let rec filter pred xs =
  match xs with
    [] -> []
  | (x::xs') ->
      if (pred x) then
        x::(filter pred xs')
      else
        filter pred xs'

let product fs xs =
  map (fun f -> map (fun x -> (f x)) xs) fs
```

Figure 1: A sampler of higher-order functions in OCaml, part 1.

```
let rec zip pair =
  match pair with
    ([], _) -> []
  | (_, []) -> []
  | (x::xs', y::ys') -> (x,y)::(zip(xs',ys'))

let rec unzip xys =
  match xys with
    [] -> ([], [])
  | ((x,y)::xys') ->
      let (xs,ys) = unzip xys'
       in (x::xs, y::ys)

let rec foldr binop init xs =
  match xs with
    [] -> init
  | (x::xs) -> binop x (foldr binop init xs)

let foldr2 ternop init xs ys =
  foldr (fun (x,y) ans -> (ternop x y ans)) init (zip(xs,ys))

let flatten = foldr (@) []

let rec forall pred xs =
  match xs with
    [] -> true
  | (x::xs') -> pred(x) && (forall pred xs')

let rec exists pred xs =
  match xs with
    [] -> false
  | (x::xs') -> (pred x) || (exists pred xs')

let rec some pred xs =
  match xs with
    [] -> None
  | (x::xs') -> if (pred x) then Some x else some pred xs'

let oneListOpToTwoListOp f =
  let twoListOp binop xs ys = f binop (zip(xs,ys))
   in twoListOp

let some2 pred = oneListOpToTwoListOp some pred
```

Figure 2: A sampler of higher-order functions in OCaml, part 2.

Problem 3 [25]: Merge Sort

In this problem, you will implement a merge sort algorithm in OCaml. Recall that merge sort on lists works by:

1. splitting the list into two sublists of (nearly) equal size;
2. recursively sorting the two sublists;
3. merging the two sorted sublists into the final sorted list.

Start this problem by performing `cvs update -d` in your local `cs301` directory. (If you haven't created this yet, follow the instructions in Handout #7.) Among other things, this will install a `ps1` subdirectory in your local `cs301` directory. In the `ps1` subdirectory is a file named `MSort.ml` that contains a skeleton of the module `MSort`. You should flesh out this skeleton by defining the following three functions:

- `val split : 'a list -> ('a list * 'a list)`
  `split` *xs* returns a pair of lists *ys* and *zs* such that (1) *ys* `@` *zs* is a permutation of *xs* and (2) `|` `length`(*ys*) `-` `length`(*zs*) `|` ≤ 1. For example, some possible results of `split [7;3;5;2;6]` are `([7;3;5],[2;6])`, `([3;7],[6;2;5])`, and `([7;5;6],[3;2])`.
- `val merge : 'a list -> 'a list -> 'a list`
  Assume that *xs* and *ys* are lists sorted according to `<=`. Then `merge` *leq xs ys* should return a list of the elements in *xs* and *ys* sorted according to `<=`. For example, `merge [1;4;5] [3;4;7]` should yield `[1;3;4;4;5;7]`.
- `val msort : 'a list -> 'a list`
  `msort` *xs* returns a list of the elements in *xs* sorted according to `<=`. Sorting should be performed by the merge sort algorithm. For example, `msort [7;1;3;8;2;3]` should return `[1;2;3;3;7;8]`.

Depending on the approach you take in your definitions, you may need to define one or more auxiliary functions.

You should test each of your three functions to make sure that they work correctly. In order to test them, you will first need to launch an OCaml interpreter. You can do this either by (1) executing `ocaml` at the Linux shell or by (2) executing `M-x run-caml` in Emacs and then hitting the return key to select `ocaml` (the default version of Caml).

Then you need to issue the following directives to OCaml:

```
# #cd "/students/gdome/cs301/ps1";;
  (* use your username in place of gdome's! *)
# #use "MSort.ml";;
```

Then you can test your functions. For example:

```
# MSort.split [7;3;5;2;6];;
- : int list * int list = ([7; 5; 6], [3; 2])
# MSort.merge [1;4;5] [3;4;7];;
- : int list = [1; 3; 4; 4; 5; 7]
# MSort.msort [7;1;3;8;2;3];;
- : int list = [1; 2; 3; 3; 7; 8]
```

Problem 4 [45]: Red-Black Trees

Red-black trees are a kind of balanced tree often used in practical implementations of sets, bags, and tables. In this problem, you will implement some operations on red-black trees in OCaml.

Begin this problem by reading the **first four pages** of the CS231 handout on Red-Black Trees at `http://cs.wellesley.edu/~cs231/fall01/red-black.pdf`. Ignore the material after the first four pages! These pages define the 5 properties of red-black trees and describe a simple algorithm for inserting elements into red-black trees.

Red-black trees can be implemented in OCaml using the following datatypes:

```
type color = Red | Black

type 'a rbt =
    Leaf
  | Node of color * ('a rbt) * 'a * ('a rbt)
```

In `Node(`c`,`l`,`v`,`r`)`, c is the color of the node, v is the value of the node, and l and r are the left and right subtrees of the node, respectively.

A skeleton of the module `RBT` has been provided for you in the file `ps1/RBT.ml` within your local CVS directory. (Remember to execute `cvs update -d` in this directory before you start work on the problem and every time you log in to work thereafter.) You should flesh out the following five functions in the `RBT` module (some of which will require the definition of auxiliary functions):

- `val insert : 'a -> 'a rbt -> 'a rbt`
  `insert` v t returns a new red-black tree that results from inserting the element v into the red-black tree t according to the insertion algorithm described in the red-black tree notes. For instance, the following test creates a tree named `algorit` by inserting the letters `A L G O R I T` one-by-one into an initially empty tree:

```
# open RBT;;
# let algorit = List.fold_left (fun t c -> insert c t)
                               Leaf
                               ['A';'L';'G';'O';'R';'I';'T'];;
val algorit : char rbt =
  Node (Black, Node (Black, Leaf, 'A', Leaf), 'G',
   Node (Red,
    Node (Black, Node (Red, Leaf, 'I', Leaf), 'L',
     Leaf),
    'O',
    Node (Black, Leaf, 'R',
     Node (Red, Leaf, 'T', Leaf))))
```

- `val member: 'a -> 'a rbt -> bool`
  `member` v t determines if v is in the red-black tree t. For example:

```
# member 'G' algorit;;
- : bool = true
# member 'H' algorit;;
- : bool = false
```

- `val elts : 'a rbt -> 'a list`
  `elts` t returns a list of all elements in the red-black tree t sorted by `<=`. For example:

```
# elts algorit;;
- : char list = ['A'; 'G'; 'I'; 'L'; 'O'; 'R'; 'T']
```

- `val isRBT : 'a rbt -> bool`
  `isRBT` t determines if the tree t is a legal red-black tree – i.e., it is a binary search tree that satisfies all five properties described in the red-black tree notes. For example:

```
# isRBT algorit;;
- : bool = true

# isRBT (Node(Black,Node(Red,Leaf,1,Leaf),2, Node(Red,Leaf,3,Leaf)));;
- : bool = true

# isRBT (Node(Black,Node(Red,Leaf,1,Leaf),
                    2,
                    Node(Red,Leaf,3,Node(Red,Leaf,4,Leaf))));;
- : bool = false (* red-red violoation *)

# isRBT (Node(Red,Node(Red,Leaf,1,Leaf),2, Node(Red,Leaf,3,Leaf)));;
- : bool = false (* root isn't black *)

# isRBT (Node(Black,Node(Black,Leaf,1,Leaf),2, Node(Red,Leaf,3,Leaf)));;
- : bool = false (* not black-height balanced *)

# isRBT (Node(Black,Node(Red,Leaf,2,Leaf),1, Node(Red,Leaf,3,Leaf)));;
- : bool = false (* not a BST -- elements in wrong order *)
```

- `val toString: ('a -> string) -> 'a rbt -> string`
  `toString` *eltToString* t returns a string representation of the red-black tree t, using the function *eltToString* to convert each element to a string. The string representation of a red-black tree should be the same as the one in Handout #8 for binary search trees, except that each black node should be prefixed with `B:` and each red node should be prefixed with `R:`. For example:

```
# print_string (RBT.toString (String.make 1) algorit);;
   R:T
  B:R
 R:O
  B:L
   R:I
B:G
 B:A- : unit = ()
```

  In the above example, the library function `String.make` is used to convert a character to a string. For instance, the value of `String.make 1 'A'` is the string `"A"`.

You need not define and test your functions in the order given above. Choose any order that is convenient for you.

You should first test your implementation with simple examples of the sort shown above. In order to test your module, you will need to launch an OCaml interpreter and execute the following directives in it:

```
# #cd "/students/gdome/cs301/ps1";;
  (* use your username in place of gdome's! *)
# #use "RBT.ml";;
```

For a more extensive test of your implementation, you should test your red-black trees in the generic set-testing framework studed in in class on Tue. Sep. 9. It is easy to view red-black trees as a set, using the following modules (which have been provided for you in `ps1/RBTSet.ml`):

```
module RBTSet : SIMPLE_SET = struct
  type 'a set = 'a RBT.rbt
  let empty = RBT.Leaf
  let verify = RBT.isRBT
  let insert = RBT.insert
  let member  = RBT.member
  let elts = RBT.elts
end

module RBTSetTest = SimpleSetTest(RBTSet)
```

You can test your red-black tree implementation in the context of these sets as shown below:

```
# #use "load-rbt-set.ml";;
  (* lots of printed output omitted here. *)

# RBTSetTest.test "../dicts/tiny-unsorted.txt";;
Reading ../dicts/tiny-unsorted.txt into list ...done
List has 16 elements
Sorting list ...done
Creating set from list ...done
Verifying set ... done
Enumerating elements of set ...done
Comparing sorted list to enumerated elements:
Comparison is successful!
- : unit = ()

# RBTSetTest.test "../dicts/tiny-sorted.txt";;
Reading ../dicts/tiny-sorted.txt into list ...done
List has 16 elements
Sorting list ...done
Creating set from list ...done
Verifying set ... done
Enumerating elements of set ...done
Comparing sorted list to enumerated elements:
Comparison is successful!
- : unit = ()
```

The word set files `tiny-unsorted.txt` and `tiny-sorted.txt` (in the directory `cs301/dicts/`) contain 16 words in unsorted and sorted order, respectively. Start with these files to debug your implementation. The `RBTSetTest.test` function will detect certain kinds of errors and print out the tree in these cases. For example, if you forget to include one of the cases for removing a red-red violation, you might get the following:

```
# RBTSetTest.test "../dicts/tiny-unsorted.txt";;
Reading ../dicts/tiny-unsorted.txt into list ...done
List has 16 elements
Sorting list ...done
Creating set from list ...done
Verifying set ... ***ERROR FOUND IN VERIFICATION***
   R:pace
  B:oaf
   R:nab
 B:mace
   R:lab
  B:kanji
   R:jab
B:ibex
   B:ha
  R:gab
   B:fable
 B:each
   R:dad
    R:cab
     R:babe
  B:aback
- : unit = ()
```

Once you have debugged your implementation on the `tiny` word sets, you should give it a workout on the bigger word sets:

- `small-sorted.txt`, `small-unsorted.txt`: 476 words
- `medium-sorted.txt`, `medium-unsorted.txt`: 5525 words
- `large-sorted.txt`, `large-unsorted.txt`: 45425 words

As part of debugging, or just if you're curious, you can also manipulate the word sets in other ways, as illustrated in Fig. 3.

```
# let tiny = RBTSetTest.fromFile "../dicts/tiny-unsorted.txt";;
Reading ../dicts/tiny-sorted.txt into list ...done
List has 16 elements
Creating set from list ...done
val tiny : string RBTSet.set = <abstr>

# RBTSet.member "zoo" tiny;;
- : bool = false

# RBTSet.member "cab" tiny;;
- : bool = true

# RBTSet.elts tiny;;
- : string list =
["aback"; "babe"; "cab"; "dad"; "each"; "fable"; "gab"; "ha"; "ibex"; "jab";
 "kanji"; "lab"; "mace"; "nab"; "oaf"; "pace"]

# RBTSet.verify tiny;;
- : bool = true

# RBTSetTest.print tiny;;
    R:pace
   B:oaf
  R:nab
   B:mace
 B:lab
  B:kanji
   R:jab
B:ibex
   B:ha
  R:gab
   B:fable
 B:each
   B:dad
  R:cab
    R:babe
   B:aback
- : unit = ()
```

Figure 3: Sample manipulations of sets based on red-black trees.

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS301 Problem Set 1
Due Friday, September 12

Names of Team Members:

Date & Time Submitted:

Collaborators (*anyone you or your team collaborated with on the problem set*):

In the **Time** *column, please estimate the time you or your team spent on the parts of this problem set. Team members should be working closely together, so it will be assumed that the time reported is the time for each team member. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the* **Score** *column when grading you problem set.*

Part	**Time**	**Score**
General Reading		
Problem 1 [10]		
Problem 2 [20]		
Problem 3 [25]		
Problem 4 [45]		
Total		