CS301 Compiler Design and Implementation **Handout # 20**
Prof. Lyn Turbak **October 17, 2003**
Wellesley College

Problem Set 1 Solutions

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.

The goal is to construct a Java Virtual Machine (JVM) from the given pieces. Here's a bottom-up description of how it can be done:

1. A Pentium-based computer is a Pentium Virtual Machine (PVM).
2. A Scheme-in-Pentium interpreter running on the PVM gives a Scheme Virtual Machine (SVM).
3. A Java-to-C-in-Scheme compiler running on an SVM gives a Java-to-C translator.
4. A C-to-Pentium-in-Pentium compiler running on a PVM gives a C-to-Pentium translator.
5. Composing the Java-to-C and C-to-Pentium translators gives a Java-to-Pentium translator.
6. A JVM can be obtained by using the Java-to-Pentium translator to translate a given Java program to Pentium code, and then executing the given Pentium code on the PVM.

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

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

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

1. A MIPS-based computer is a MIPS Virtual Machine (MVM).
2. An OCaml-in-MIPS interpreter running on an MVM yields an OCaml Virtual Machine (OVM).
3. An OCaml-to-MIPS-in-OCaml compiler running on an OVM yields an OCaml-to-MIPS translator.
4. Using the OCaml-to-MIPS translator to translate the OCaml-to-MIPS-in-OCaml compiler yields an 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.*

The OCaml interpreter is no-longer necessary. We still have an MVM, and together with the OCaml-to-MIPS-in-MIPS compiler, we have an OCaml-to-MIPS translator. We use this translator to translate any OCaml program to MIPS code, and then execute the MIPS code on the MVM.

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

Figures 1 and 2 show each of the OCaml functions, along with the type given to them by the OCaml type reconstructor. A few notes:

- The particular type variable names used don't matter, so any consistent renamings of the following types are also valid. For example, the type `('a -> 'b) -> ('c -> 'a) -> 'c -> 'b` could also be written`('c -> 'e) -> ('q -> 'c) -> 'q -> 'e`.o
- Arrow types are right associative, so the type `'a -> 'b -> 'c -> 'd` is parsed as if it were written `'a -> ('b -> ('c -> 'd))`
- The product constructor `*` binds more tightly than `->`, so `'a * 'b -> 'c * 'd` means `('a * 'b) -> ('c * 'd)` and not `'a * ('b -> 'c) * 'd`
- A normal type variable (such as `'a` or `'b`) can be instantiated to any type in each use of a function with that type. For example, `repeated` can be used with type

  ```
  int -> (int -> int) -> int -> int
  ```

  at one point and

  ```
  int -> (bool -> bool) -> bool -> bool
  ```

  at another point within the same program.
- The underscore type variable `'_a` appearing in the type of `flatten` is an unknown type that can be instantiated to at most one type in the whole program. For example, if it is used at the type `(int list list -> int list)` at one point in the program, it *cannot* be used at type `(bool list list -> bool list)` at another point in the same program. Such type variables are introduced by the OCaml reconstruction algorithm to satisfy something called the **value restriction**. They can be avoided in this case by "eta-expanding" the function definition so that it has an explicit list argument:

  ```
  let flatten xs = foldr (@) [] xs
  val flatten : 'a list list -> 'a list
  ```

```
let id x = x
val id : 'a -> 'a

let compose f g x = (f (g x));;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b

let rec repeated n f =
  if (n = 0) then id else compose f (repeated (n - 1) f)
val repeated : int -> ('a -> 'a) -> 'a -> 'a

let uncurry f (a,b) = (f a b)
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c

let curry f a b = f(a,b)
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c

let churchPair x y f = f x y
val churchPair : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c

let rec generate seed next isDone =
  if (isDone seed) then
    []
  else
    seed :: (generate (next seed) next isDone)
val generate : 'a -> ('a -> 'a) -> ('a -> bool) -> 'a list

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

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

let product fs xs =
  map (fun f -> map (fun x -> (f x)) xs) fs
val product : ('a -> 'b) list -> 'a list -> 'b list list

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

Figure 1: Types of OCaml functions, part 1.

```
let rec unzip xys =
  match xys with
    [] -> ([], [])
  | ((x,y)::xys') ->
      let (xs,ys) = unzip xys'
       in (x::xs, y::ys)
val unzip : ('a * 'b) list -> 'a list * 'b list

let rec foldr binop init xs =
  match xs with
    [] -> init
  | (x::xs) -> binop x (foldr binop init xs)
val foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

let foldr2 ternop init xs ys =
  foldr (fun (x,y) ans -> (ternop x y ans)) init (zip(xs,ys))
val foldr2 : ('a * 'b * 'c -> 'c) -> 'c -> 'a list -> 'b list -> 'c

let flatten = foldr (@) []
val flatten : '_a list list -> '_a list

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

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

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

let oneListOpToTwoListOp f =
  let twoListOp binop xs ys = f binop (zip(xs,ys))
   in twoListOp
oneListOpToTwoListOp : ('a -> ('b * 'c) list -> 'd)
                       -> 'a -> 'b list -> 'c list -> 'd

let some2 pred = oneListOpToTwoListOp some pred
val some2 : ('a * 'b -> bool) -> 'a list -> 'b list -> ('a * 'b) option
```

Figure 2: Types of OCaml functions, part 2.

Problem 3 [25]: Merge Sort

- *`val split : 'a list -> 'a list * 'a list`*: There are many ways to define `split`:
  - A recursive approach suggested by the divide/conquer/glue strategy is as follows:

```
let rec split xs =
  match xs with
    [] -> ([], [])
  | (x::xs') ->
      let (ys,zs) = split xs'
       in (x::zs,ys)
```

    With this version, `split [1;2;3;4;5]` yields `([1;3;5],[2;4])`.
  - An iterative version suggested by dealing out a pile of cards into two piles is:

```
let rec split xs =
  let rec sp xs ys zs =
    match xs with
      [] -> (ys,zs)
    | (x::xs') -> sp xs' zs (x::ys)
  in sp xs [] []
```

    With this version, `split [1;2;3;4;5]` yields `([4;2],[5;3;1])`.
  - An auxiliary function `alts` that returns every other element of a list can be called on the list and its tail to split the list:

```
let rec split xs =
  let rec alts ys =
    match ys with
      [] -> []
    |        [y] -> [y]
    | (y1::y2::ys') -> y1::(alts ys')
  in match xs with
    [] -> ([], [])
  | (x::xs') -> (alts xs, alts xs')
```

    With this version, `split [1;2;3;4;5]` yields `([1;3;5],[2;4])`.
  - Auxiliary functions `take` (returning the first n elements of a list) and `drop` (returning all but the first n elements of a list) can be used to split the list:

```
let rec split3 xs =
  let rec take n ys =
    if n = 0 then
      []
    else match ys with
      [] -> raise (Failure "take: too few values")
    | (y::ys') -> y::(take (n-1) ys')
  and drop n ys =
    if n = 0 then
      ys
    else match ys with
      [] -> raise (Failure "take: too few values")
    | (y::ys') -> (drop (n-1) ys')
  in let half = (List.length xs)/2
      in (take half xs, drop half xs)
```

    With this version, `split [1;2;3;4;5]` yields `([1;2],[3;4;5])`.

- *val merge:'a list -> 'a list -> 'a list*: Merge can be defined in a straightforward manner using pattern matching. `match` can be used for each of the two lists, but it is more concise to match against the pair `(xs,ys)`:

```
let rec merge xs ys =
  match (xs,ys) with
  | ([], _) -> ys
  | (_, []) -> xs
  | (x::xs', y::ys') -> if (x <= y) then x::(merge xs' ys)
                        else y::(merge xs ys')
```

  Note that it is not necessary to have a special case for `([],[])` since this is already handled by the case `([], _)`.

- *val msort : 'a list -> 'a list*: The merge sort algorithm can be defined via the divide/conquer/glue strategy. The single case must be handled specially to avoid an infinite recursion on a singleton list:

```
let rec msort xs =
  match xs with
    [] -> []
  | [_] -> xs
  | _ -> let (ys,zs) = split xs
         in merge (msort ys) (msort zs)
```

Problem 4 [45]: Red-Black Trees

- *val member: 'a -> 'a rbt -> bool*: This is just like the `BST.member` function, except that it ignores node colors:

```
let rec member v t =
  match t with
    Leaf -> false
  | Node (_, l, x, r) -> (v = x) || ((v < x) && (member v l)) || (member v r)
```

- *val elts : 'a rbt -> 'a list*: This is just like the `BST.elts` function, except that it ignores node colors:

```
let rec elts t =
  match t with
    Leaf -> []
  | Node(_, l,v,r) -> (elts l) @ [v] @ (elts r)
```

- *val toString: ('a -> string) -> 'a rbt -> string*: The code (presented in Fig. 3) is similar to that for `BST.toSTring` except that is uses the helper function `colorToString` to display the color of each node.

- *val insert : 'a -> 'a rbt -> 'a rbt*: The insertion process (Fig. 4) for red-black trees is similar to that for binary search trees except:

  – A `fixup` function is called at every node along the search path for inserting the element. This function removes red-red violations on the way back out of the `ins` recursion. Note how OCaml's patterns (particulary "or patterns") make the specification of the four tree transformations incredibly concise.

  – The root node of the tree must be blackened in case the red-red violation transformations propagate a red node to the root of the tree.

```
let rec toString f t = String.concat "\\n" (toStrings f t)

and toStrings f t =
  match t with
    Leaf -> []
  | Node(c,l,v,r) ->
      (List.map (prefix 1) (toStrings f r))
    @ [(colorToString c) ^ ":" ^ (f v)]
    @ (List.map (prefix 1) (toStrings f l))

and prefix n s =
  if n = 0 then
    s
  else
    " " ^ (prefix (n-1) s)

and colorToString c =
  match c with
    Red -> "R"
  | Black -> "B"
```

Figure 3: The red-black tree `toString` function.

```
let rec insert v t = blackenRoot (ins v t)

and ins v t =
  match t with
    Leaf -> Node(Red,Leaf, v, Leaf)
  | Node(c,l,x,r) ->
      if (v = x) then
        t (* Assume duplicates are not inserted *)
      else if (v < x) then
        fixup (Node(c,ins v l, x, r))
      else
        fixup (Node(c,l, x, ins v r))

and fixup t =
  match t with
      Node(Black,a,x,Node(Red,b,y,Node(Red,c,z,d)))
    | Node(Black,Node(Red,a,x,Node(Red,b,y,c)),z,d)
    | Node(Black,Node(Red,Node(Red,a,x,b),y,c),z,d)
    | Node(Black,a,x,Node(Red,Node(Red,b,y,c),z,d))
      -> Node(Red,Node(Black,a,x,b),y,Node(Black,c,z,d))
  | _ -> t

and blackenRoot t =
  match t with
    Leaf -> Leaf
  | Node(_,l,v,r) -> Node(Black,l,v,r)
```

Figure 4: The red-black tree `insert` function.

- *val isRBT : 'a rbt -> bool*: This function requires checking the red-black tree properties. Some properties need not be checked: all nodes are necessarily either red or black (Property 1), and all leaf nodes are black (Property 2) by definition. The following properties *do* need to be checked:

  – the tree is a binary search tree;
  – there are no red-red violations (Property 3);
  – the tree is black-height balanced (Property 4); and
  – the root of the tree is black (Property 5).

  These properties are tested by the following function:

```
let rec isRBT t =
   isBST(t) (* it's a BST *)
   && hasNoRedRedViolations(t) (* Property 3 *)
   && isBlackHeightBalanced(t) (* Property 4 *)
   && isRootBlack(t) (* Property 5 *)
```

  The easiest property to check is for blackness of the root:

```
and isRootBlack(t) =
  match t with
    Leaf -> true
  | Node(Black,l,_,r) -> true
  | _ -> false
```

  Checking the BST property is also straightforward – test if the inorder elements are sorted:

```
and isBST t = isSorted(elts(t))

and isSorted xs =
  match xs with
    [] -> true
  | [_] -> true
  | (x1::((x2::_) as xs')) -> (x1 <= x2) && isSorted xs'
```

  The lack of red-red violations is verified by the following tree-walking function:

```
and hasNoRedRedViolations t =
  match t with
    Leaf -> true
  | Node(Black,l,_,r) -> hasNoRedRedViolations(l) && hasNoRedRedViolations(r)
  | Node(Red,l,_,r) -> isRootBlack(l) && isRootBlack(r)
                       && hasNoRedRedViolations(l) && hasNoRedRedViolations(r)
```

  The trickiest of the four properties to check is that the tree is black-height balanced at every node. The most direct way to do this is to create a list of the black-heights of all root-to-leaf paths in the tree and verify that they are all the same:

```
and isBlackHeightBalanced' (t) = allSame (blackHeights (t))

and blackHeights t =
  match t with
    Leaf -> [1]
  | Node(Black,l,_,r) ->
      map (fun x -> x + 1) (blackHeights(l) @ blackHeights(r))
  | Node(Red,l,_,r) -> blackHeights(l) @ blackHeights(r)

and allSame ns =
  match ns with
    [] -> true
  | [_] -> true
  | n1::((n2::_) as ns') -> (n1=n2) && allSame ns'
```

One drawback of the direct approach is the creation and processing of large lists. Indeed, for the large dictionary tests in the problem. processing these lists causes a stack overflow in our version of OCaml.

An alternative solution that does not require creating any intermediate lists is as follows: check at every node that same number of black nodes are encountered on the leftmost path from the left and right subtrees of any given non-leaf node in the tree. An induction on the height of the tree shows that this is equivalent to showing that all paths from a given node have the same number of black nodes:

- If the height is 0 (i.e., the node is a leaf), then the tree is necessarily black-height balanced;
- If the height of the node is $h \geq 1$, by the inductive hypothesis, the left and right subtrees are black-height balanced. So from each subtree node, all paths encounter the same number of black nodes. So it is safe to count the number of blacks along *any* path from either subnode, as long as the blackness of the root of each subtree is considered, If any path from the left subtree has the same number of blacks as any path from the right subtree, then the given node is necessarily black-height balanced.

This alternative strategy is easily expressed in OCaml as follows:

```
;;; One way to test black-height balance:
and isBlackHeightBalanced(t) =
  match t with
    Leaf -> true
  | Node(_,l,_,r) ->
      (countBlacks(l) = countBlacks(r))
    && isBlackHeightBalanced(l)
    && isBlackHeightBalanced(r)

(* arbitrarily choose leftmost path for black height *)
and countBlacks(t) =
    match t with
    Leaf -> 1
  | Node(Black,l,_,_) -> 1 + countBlacks(l)
  | Node(Red,l,_,_) -> countBlacks(l)
```

Using this implementation of `isBlackHeightBalanced`, there is no stack overflow for the large dictionary tests in the problem.