

1. Give IR translations for:

(a)  $[\text{break}]_L$

(b)  $[x]_{Ll,Lf}$  (where  $x$  is a variable)

(c)  $[e_1 || e_2]_{Ll,Lf}$  (using short-circuit evaluation for  $e_1$  and  $e_2$ )

(d) Generate code for the repeat-until statement: “repeat  $S$  until  $e$ ” executes  $S$  and tests  $e$ , and repeats until  $e$  becomes true. Thus, it is equivalent to “ $S$ ; while ! $e$  do  $S$ ”.

Code generation: Translation from AST to IR

$[s]$  - statement  $\rightarrow$  IR

$[e]$  - expression  $\rightarrow$  IR  $\times$  loc

$[e]_x$  - expression  $\rightarrow$  IR

$[s]_L$  - statement  $\rightarrow$  IR, for statements inside while or switch where L is label to break to.

$[e]_{L_t, L_f}$  - boolean expr  $\rightarrow$  IR ("short circuits")

$[x=e]$  - boolean expr,  
using short-circuit eval for  $e$ .

$[x=e]$  = let  $L_t, L_f, L = \text{newlabel}()$  in

$[e]_{L_t, L_f}$

$L_t$ :  $x = \text{true}$

JUMP  $L$

$L_f$ :  $x = \text{false}$

$L$ :

$[S]_{Lb, Lc}$  -  $S$  in a loop, where  $Lb$  is the break label and  $Lc$  is the continue label

$$[x=e]_{Lb, Lc} = [x=e]$$

$$[if (e) S]_{Lb, Lc} = [e]_{Lt, Lf}$$

$$Lt: [S]_{Lb, Lc}$$

$Lf:$

$$[break]_{Lb, Lc} = \text{JUMP } Lb$$

$$[continue]_{Lb, Lc} = \text{JUMP } Lc$$

$[ \text{while } (e) S ] = \text{JUMP } L_2$

$L_1: [S]_{L_3, L_2}$

$L_2: [e]_{L_1, L_3}$

$L_3:$

$[ \text{while } (e) S ]_{L_b, L_c} = \text{JUMP } L_2$

$L_1: [S]_{L_3, L_2}$

$L_2: [e]_{L_1, L_3}$

$L_3:$

2. Write APL expressions for the following calculations.

(a) the average of the numbers from 1 to n

(b) the sum of the squares of the elements of a vector V

(c) the product of all positive elements of a vector V

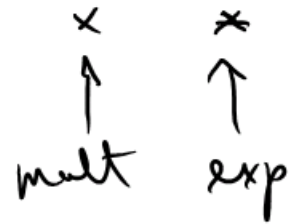
$$\# / \underbrace{(V > 0)}_{\text{elements of } V \text{ where } V > 0} / V = 1$$

(d) a matrix with the numbers 1, 2, ..., n on the diagonal and 0 everywhere else. You may use the function `idmat(x)` to produce the identity matrix of size x.

3. (a) Name the two parts of a compiler's front end.

(b) Name the two parts of a compiler's back end.

Actual APL:



$$\oplus / V$$

If  $V = v_1, \dots, v_n$ ,

$$\text{then } \oplus / V = v_1 \oplus v_2 \oplus \dots \oplus v_n$$

(c) What are the two outputs of the front end?

4. (a) Give two advantages of the copying garbage collection algorithm over the non-copying (mark-and-sweep) algorithm.

- Allocation, esp of large chunks is simpler
- GC often cheaper: proportional to size of reachable data, not entire heap
- May be more efficient due to memory system performance.

(b) Give two advantages of the non-copying (mark-and-sweep) garbage collection algorithm over the copying algorithm.

- Don't have to copy
  - Updating pointer is complicated
  - Can be expensive for large collections of non-pointers
- Don't waste half of memory

(c) Reference counting is not a popular algorithm. What is its major drawback?

- Circular structures

5. (a) What is the type of the following function? `fun f -> fun g -> fun x -> f (g x)`
- (b) Write an OCaml function that reverses a list, using `fold_right` instead of explicit recursion.
- (c) Use `map` to write a function `map_first f l` which applies `f` to the first element of each item in `l`, assuming that `l` is a list of pairs.
- (d) Write a function `curry` that converts a function `f` on pairs to curried form. In other words, if `f` is defined by `let f (x,y) = e` for some expression `e`, `curry f` should return the function `g` defined by `let g x y = e`.
- (e) Using `fold_right` and no explicit recursion, define a function that concatenates the elements of a string list.





7. Write a function object for `case_map` (see the OCaml definition below). For the sake of simplicity, we assume that  $f : \text{int} \rightarrow \text{bool}$ ,  $g, h : \text{int} \rightarrow \text{int}$ .

```
let case_map f g h lis = map (fun x -> if (f x) then (g x) else (h x)) lis;
```

Your answer:

```
interface BoolFun{
  boolean apply(int n);
}
interface IntFun{
  int apply(int n);
}

class Map{
  static int[] map(IntFun f, int lis[]){
    int lis2[] = new int[lis.length];
    for(int i = 0; i < lis.length; i++){
      lis2[i] = f.apply(lis[i]);
    }
    return lis2;
  }
}

class Case_Map{
  static int[] case_map(BoolFun f, IntFun g, IntFun h, int lis[]){
    //complete this method

  }
}
```

