CPL-sup1

Q1.2

Exercise 1.2. One of the most important factors in the design of FORTRAN was the ease of compiling it. Identify some aspects of the original FORTRAN abstract machine that make it a language that can be compiled down to efficient machine code. [*Hint:* think of the optimisations you mentioned in the *Compiler Construction* course.]

- In FORTRAN, the abstract machine is a flat register machine where memory is arranged as a linear array. All storage is allocated statically before the program execution, so the compiler knows at the compile time the memory storage of the program; therefore, it can do compile-time meta-programming or constant propagation to compute the results. ✓
- FORTRAN compiler assumes arrays do not overlap in memory. Aliasing is not allowed in FORTRAN 77, so the same memory location can only be modified by one variable; hence the compiler can reorder the machine instructions to keep multiple internal arithmetic units running simultaneously. ✓

💡 Comments: 🟨

Q1.4

Exercise 1.4. LISP features an `eval` keyword and an ability to *quote* expressions.

(*a*) Show how the following expressions are evaluated, step by step:

```
(eval '(+ 1 2 (eval '(+ 3 4))))
(cons 1 (list 2 3 (eval (cdr (cons 4 '(cdr '(5 6)))))))
```

(*b*) Give a brief overview how `eval` can be implemented using quoting, list processing and pattern matching constructs in Lisp.

(*c*) How do quoting and `eval` interact with scoping?

- (a)(1):

```
(eval '(+ 1 2 (eval '(+ 3 4))))
-(eval)--> (+ 1 2 (eval '(+ 3 4)))
-(eval)--> (+ 1 2 (+ 3 4))
-(+ 3 4)--> (+ 1 2 7)
-(+ 1 2 7)--> 10
```

- (a)(2):

```
(cons 1 (list 2 3 (eval (cdr (cons 4 '(cdr '(5 6)))))))
-(eval)--> (cons 1 (list 2 3 (cdr (cons 4 '(cdr '(5 6))))))
-(cdr '(5 6))--> (cons 1 (list 2 3 (cdr (cons 4 '(6)))))
-(cons 4 '(6))--> (cons 1 (list 2 3 (cdr '(4 6))))
-(cdr '(4 6))--> (cons 1 (list 2 3 '(6)))
-(list 2 3 '(6))--> (cons 1 '(2 3 (6)))
-(cons 1 '(2 3 (6)))--> (1 2 3 (6))
```

- (b):
  - **A Lisp interpreter** `eval(e)` **in Lisp** with a read-eval-print loop (i.e., read an expression, evaluate its value, and then print it)
  - `eval(e)` where `e` is an expression and we have a separate environment `a` which maps from function to function definitions, and variables to values.
  - Case 1 `e = atom`: we look up the value in the environment
  - Case 2 `e = (e1 e2 ... en)`: then `e1` could be any of the seven primitives (`quote, atom, eq, car, cdr, cons, cond`) or a handler to functions that passed as arguments.
    - All of the cases except `quote` case would recursively call `eval` to find the value of the arguments
    - Case 2.1 `e1 = quote`: return `e2` unevaluated
    - Case 2.2 `e1 = atom`: eval `e2` to `v2` → `(atom v2)`
    - Case 2.3 `e1 = eq`: eval `e2` & `e3` → `(eq v2 v3)`
    - Case 2.4 `e1 = car`: eval `e2` to `v2` → `(car v2)`

        - Case 2.5 `e1 = cdr`: eval `e2` to `v2` → `(cdr v2)`
        - Case 2.6 `e1 = cons`: eval `e2` & `e3` → `(cons v2 v3)`
        - Case 2.7 `e1 = cond`: evaluate the list of condition-value pairs from left to right, looking for one in which the first condition returns `t` and returns the value corresponding to that condition ✓
        - Case 2.8 handles function as arguments: look up the function in the environment, then recursively evaluate the function with arguments. ✓
    - Case 3 recursive function: push a list of function names and the function itself onto the environment `a` and then recursively call `eval` on inner function body (i.e., the inner lambda expression) ✓
    - Case 4 lambda function: first get a list of values `(v1 v2 ... vn)` of the arguments `(a1 a2 ... an)` and then evaluate `e` with an updated environment `a` which includes `(p1 v1) (p2 v2) ... (pn vn)`. ✓

```
;; eval function Lisp interpreter
(defun eval (e a) ;eval takes an list of expression e and an environment (a list of arguments a)
    (cond
        ((atom e) (lookup. e a)); case 1: e is an atom, lookup value in the environment
        ((atom (car e)); case 2: e is of the form (e1 e2 ... en)
            (cond
                ((eq (car e) 'quote) (cadr e)); case 2.1: e1 = quote
                ((eq (car e) 'atom) (atom (eval. (cadr e) a))); case 2.2: e1 = atom
                ((eq (car e) 'eq) (eq (eval. (cadr e) a)
                                      (eval. (caddr e) a))); case 2.3: e1 = eq
                ((eq (car e) 'car) (car (eval. (cadr e) a))); case 2.4: e1 = car
                ((eq (car e) 'cdr) (cdr (eval. (cadr e) a))); case 2.5: e1 = cdr
                ((eq (car e) 'cons) (cons (eval. (cadr e) a)
                                      (eval. (caddr e) a))); case 2.6: e1 = cons
                ((eq (car e) 'cond) (evcon. (cdr e) a)); case 2.7: e1 = cond
                ('t (eval. (cons (lookup. (car e) a) (cdr e))
                           a)))); case 2.8: e1 = customised function
        ((eq (caar e) 'label); case 3: recursive function call
                (eval. (cons (caddar e) (cdr e))
                (cons (list (cadar e) (car e)) a)));
        ((eq (caar e) 'lambda); case 4: lambda function call
                (eval. (caddar e)
                (append. (pair. (cadar e) (evlis. (cdr e) a));
                a)))))

;; for case 2.7 e1 = cond
(defun evcon (c a)
    (cond ((eval (caar c) a) (eval (cadar c) a))
            ('t (evcon (cdr c) a))))

;; for case 4 getting all argument values
(defun evlis (m a)
    (cond ((null. m) '())
            ('t (cons (eval. (car m) a)
            (evlis (cdr m) a)))))
```

- (c):

- `eval` updates the environment during expression evaluation and selects the most recent binding of a variable or function definition, and dynamic scoping is used.
- `quote` prevents the arguments from being evaluated; if we use `quote` around function arguments, we perform call-by-text.

> 💡 Comments:

Q1.7

Exercise 1.7.

(*a*) What are Pascal *discriminated unions*, and what machine-level structures they capture?

(*b*) Why are they unsafe?

(*c*) How can they be represented directly in C?

(*d*) Give similar, but safe, analogues in Java and ML.

(*e*) What would be a LISP analogue?

- (a):
  - A discriminated union is a data structure used to hold a value that could take on several different but fixed types, and only one of the types would be used at any time. We usually use a tag field to indicate which one of the types is in use.
  - At the machine level, the discriminated union can save storage by overlapping storage areas for each type; ideally, the storage requirement would be the largest of any type since only one of the types is in use at a time.
- (b):
  - Type checking of the memory chunks is complex if we store a discriminated union inside; that's because we don't know the exact type at compile time. The compiler also does not check the consistency between the tag field and the exact type of the record.
  - At run-time, since the tag field is optional, no checking is possible to determine whether the selected field is of the correct type.
- (c): we could use a union to store discriminated union data types and use a struct to store both a tag field and a nested union data structure.

```
typedef struct exp Exp;
struct exp{
  enum tag{
    Var = 1,
    Neg = 2,
    Divide = 3
  } t;
  union{
    Exp *arg1;
    char *var_name;
  };
  Exp *arg2;
};
```

- (d):
  - In Java, we can use a base class with an integer tag field and each inherited classes corresponding to one type.

```
public class Exp {
    private int tag;

    Exp(int tag) {
        this.tag = tag;
    }
}

class Var extends Exp {
    private char var_name;

    public Var(char var_name) {
        super(1);
        this.var_name = var_name;
    }
}

class Neg extends Exp {
    private Exp arg;

    public Neg(Exp arg) {
        super(2);
        this.arg = arg;
    }
}

class Divide extends Exp {
    private Exp arg1;
    private Exp arg2;

    public Divide(Exp arg1, Exp arg2) {
        super(3);
        this.arg1 = arg1;
        this.arg2 = arg2;
    }
}
```

- In ML, define a nested datatype `expType` contains an integer and an `exp`, where `exp` is a customised datatype.

```
type exp =
  | Var of char
  | Neg of exp
  | Divide of exp * exp
type expType = MKEXP of int * exp
```

- (e):
  - We could use `(cond (c1 e1) (c2 e2) ... (cn en))`, where `c1`, `c2`, ..., `cn` checks the tag of the expression; once `ci` returns true, we could then return the corresponding value for that tag.

```
(defun exp (tag value_list)
    (cond ((eq tag 1) (evalVar value_list))
          ((eq tag 2) (evalNeg value_list))
          ((eq tag 3) (evalDivide value_list))
    )
)
```

💡 Comments:

Q1.10

Exercise 1.10. Everything is an object in Smalltalk. This includes classes, and there are no primitives. Discuss this design decision, while drawing on comparisons to other languages you have studied (in this course and in the Tripos).

- Classes help programmers organise thoughts and establish relationships between different concepts (e.g., the superclass and subclass relationship); with this consistent system, a program can be understood much better within a team of programmers. This also helps reduce the complexity of the language as everything is treated the same - object instances of classes.
- Modularisation helps us to build programs by combining and interacting between different classes without the need to write from scratch.
- However, this restricts the way we implement algorithms; a programmer has to follow the particular object-oriented paradigm or programming model when they would like to use another one to solve the problem.

- For instance, in Java, if we use primitive type `int` instead of `java.lang.Integer`, at the machine code level, we can directly use the value of `int` without converting `java.lang.Integer` into suitable machine level primitive types (i.e., `int`); hence primitive types lead to higher efficiency and better performance in simple programs.
- The storage requirements for objects and primitive types also differ, we would need to create a new reference for every object in memory, but for primitive types, two variables with the same type and value can point to the same memory chunk, thus saves memory.

```
int x = 1;
int y = 1;
x == y; // returns true
Integer x = 1;
Integer y = 1;
x == y; // returns false
```

- This model also abstracts away the low-level machine code, programmers are unable to optimise the code to improve efficiency directly like what they could do in C/C++. But on the other hand, encapsulation and abstraction help avoid programming errors (e.g., array bound checking).

💡 Comments:

Q2.3

Exercise 2.3. Use the type inference algorithm described in the notes to find the type of the following Standard ML expression:

```
fn x => fn y => fn z => z (x y) y
```

```
assume Γ = {x:'a1, y:'a3, z:'a5}
                ------------ -------------
                Γ |- x : 'a9 Γ |- y : 'a10
-------------  ------------------------  ------------
Γ |- z : 'a5   Γ |- (x y) : 'a7          Γ |- y : 'a8
-----------------------------------------------------
x : 'a1, y : 'a3, z : 'a5 => z(x y) y : 'a6
-----------------------------------------------------
x : 'a1, y : 'a3 |- fn z => z(x y) y : 'a4
-----------------------------------------------------
x : 'a1 |- fn y => fn z => z(x y) y : 'a2
-----------------------------------------------------
|- fn x => fn y => fn z => z (x y) y : 'a0

'a0 = 'a1 -> 'a2
'a2 = 'a3 -> 'a4
'a4 = 'a5 -> 'a6
```

```
'a5 = 'a7 -> 'a8 -> 'a6
'a9 = 'a10 -> 'a7
'a8 = 'a3
'a9 = 'a1
'a10 = 'a3

hence, we have:
'a9 = 'a3 -> 'a7
'a0 = ('a3 -> 'a7) -> 'a3 -> ('a7 -> 'a3 -> 'a6) -> 'a6
    = ('a -> 'b) -> 'a -> ('b -> 'a -> 'c) -> 'c
```

Comments:

Q2.5

Exercise 2.5. One of the criticisms of Pascal in *Why Pascal is Not My Favorite Programming Language* is that array length is part of the array type.

(*a*) Give an example of a function that cannot be written in Pascal because of this constraint, but can be written in Java and C.

(*b*) How would you still be able to write it using other language features?

(*c*) Is there any merit to this design decision?

(*d*) Suppose a programming language had a type `array[k] of T`, meaning 'array with items of type `T`, of size at most `k`', and this language defined a subtyping relation:

$$\texttt{array[}p\texttt{] of } \tau \preccurlyeq \texttt{array[}q\texttt{] of } \tau \quad \text{whenever} \quad p \leq q$$

Discuss this approach.

- (a):
  - Many array manipulation programs are not possible such as `reverse(arr)`, which reverses an array of any length, `sort(arr)`, which sorts an array of any length, and `concate(arr1, arr2)`, which concatenates two arrays of any length, that's because all arguments need to specify a type, but if the array length is included in the type, then the function would be restricted to only those arrays with specified length, not general arrays with any length.

```
// Java
public class Reverse {
    public static void reverse(int[] arr) {
        for (int i = 0; i < arr.length / 2; i++) {
            int temp = arr[i];
```

```
            arr[i] = arr[arr.length - i - 1];
            arr[arr.length - i - 1] = temp;
        }
    }

    public static void main(String[] args) {
        int a[] = { 1, 2, 3, 4, 5 };
        reverse(a);
        for (int i = 0; i < a.length; i++) {
            System.out.println(a[i]);
        }
    }
}
// C
#include <stdio.h>
void reverse(int *arr[], int len)
{
    for (int i = 0; i < len / 2; i++)
    {
        int temp = arr[i];
        arr[i] = arr[len - i - 1];
        arr[len - i - 1] = temp;
    }
}
void main()
{
    int *a[] = {1, 2, 3, 4, 5};
    reverse(a, 5);
    for (int i = 0; i < 5; i++)
    {
        printf("%d\n", a[i]);
    }
}
```

- (b):
  - (1) use a built-in data type `string` where the constant `MAXSTR` is big enough, and all strings in all programs are exactly this size; all unused bytes are padded with zeros.
  - (2) then we have to specify in the `reverse` program two arguments: the string with length `MAXSTR` and its original length `n`, thus we can reverse the string appropriately.

very sneaky

```
// a built-in data type string
type string = array [1..MAXSTR] of char;
```

- (c):
  - Easy for array alignment in the low-level machine memory;
  - There will be no array index out of bounds error because we already know at compile time the maximum array length;
  - We do not have to call a utility function `len` to find out the length of an array and such a procedure is very common for array manipulations.
- (d):

- A subtype relationship means any array of length p is a subtype of (i.e., can be used anywhere when the program requires) an array of length q such that $p \leq q$. This approach may cause problems because if we fetch an array of length p when we require an array of length q, then any indexes i such that $p < i < q$ are not valid and will cause an error of array index out of bounds. ✓
- If we extend the array of length p to an array of length q by appending $q - p$ zeros at the end of the array of length p, then these two arrays can be considered the same type, and the above problem of index access can be solved (e.g., all indexes accessing i where $p < i < q$ will return 0 instead of triggering an error)

💡 Comments: 

Q2.6

Exercise 2.6. Why does this C++ code give exactly one type error?

```
1  class A {};
2  class B : public A {};
3
4  A **p;
5  B **q;
6
7  void foo() {
8    *p = *q;
9  }
10 void foo2() {
11   p = q;
12 }


13 int main() {
14   // initialise p and q before calling foo and/or foo2
15   return 0;
16 }
```

Why is this error message necessary for type safety?

- It gives only one type error message “a value of type `**B` cannot be assigned to an entity of type `A**`” ✓
- It is not type-safe because we cannot guarantee a type `**A` would eventually point to a type `B`. ✓

- In the following example, if we can use `**B` anywhere that requires `**A`, then we can set `*p = new A()`, where `*p` has a type `*A`, i.e., the location pointed by `p` would point to a new instance of `A`.
- But in the example, initially, the location pointed by `p` stores a pointer `b` which points to an instance of `B`, hence `b` would end up pointing to an instance of `A`, which is bad because that means we might return an instance of `A` when anywhere requires an instance of `B` which contains more fields and methods.

```
// using *B anywhere requires *A is allowed
B *b = new B();
A *a = b;

// using **B anywhere requires **A is not allowed
B **q = &b;
A **p = q; // type error

// if we can use **B anywhere requires **A
*p = new A(); // and b ends up pointing to an instance of A
```

💡 Comments:

Q2.7

Exercise 2.7. Why does this Java code raise an exception? What happens when the various commented-out code is uncommented?

```
1  import java.util.ArrayList;
2  import java.util.Arrays;
3
4  class Fruit {
5    int weight;
6  }
7
8  class Apple extends Fruit {
9    boolean isRed;
10 }
11
12 public class Foo {
13    public static void main(String[] args) {
14       System.out.println("Starting...");
15       Fruit f = new Fruit();
16       Apple a = new Apple();
17
18       System.out.println("Simple casting:");
19       Fruit OKf = a;
20 //      Apple ERRORa = f;
21
22       // olde-style arrays:
23       Apple[] av = new Apple[10];
24       Fruit[] fv = new Fruit[10];
25       // make ArrayLists containing the same elements as the arrays av,fv:
26       ArrayList<Apple> al = new ArrayList<Apple>(Arrays.asList(av));
27       ArrayList<Fruit> fl = new ArrayList<Fruit>(Arrays.asList(fv));
28       System.out.println("Checking: al.size="+al.size()+"; fl.size="+fl.size());
29
30       // now explore variance ...
31 //      ArrayList<Fruit> ERRORq = al; //ERROR!
32       ArrayList<? extends Fruit> p = al;
33       ArrayList<Fruit> q = fl;
34       Fruit gotf = p.get(3);
35 //      p.set(3,f);    // ERROR!
36       q.set(3,f);
37
38       System.out.println("Olde-style arrays and variance:");
39       Fruit[] r = av;
40       r[3] = f;
41
42       System.out.println("Stopping");
43    }
44 }
```

- An exception at line 40 `r[3] = f` is raised because Java `Array` supports covariance, since `Fruit` is a supertype of `Apple`, we have `Fruit[]` is a supertype of `Apple[]`, hence no compile error is raised when we set `Fruit[] r = av` where `av` is of type `Apple[]`. But an array can only store a single type element, we cannot put an element of type `Fruit` into an array of elements with type `Apple`, hence run-time exception `java.lang.ArrayStoreException` is raised.
- (1) `Apple ERRORa = f;`: compile error because we cannot use a supertype whenever it requires a subtype, since a subtype has more fields or methods than a supertype.

- (2) `ArrayList<Fruit> ERRORq = a1;`: compile error because although `Fruit` is a supertype of `Apple`, Java `ArrayList` does not support covariance, generics are invariant, i.e., `ArrayList<Fruit>` is not a supertype of `ArrayList<Apple>`, because due to type erasure, JVM has no way of knowing at runtime the type information of the type parameters, hence to protect against heap pollution, type parameters must match exactly.
- (3) `p.set(3,f)`: compile error because, for covariant types, we can only read from but not write to, `ArrayList<? extends Fruit>` makes sure the `ArrayList` stores any subtype of `Fruit` or `Fruit` type, hence we can use `Fruit f = p.get(3)` to get an instance which has a base class `Fruit`; but since we have no exact type information of the objects inside `ArrayList`, we cannot be written to the `ArrayList`.

💡 Comments:

Q2.11

Exercise 2.11.

(*a*) Write a signature for a Queue abstract data type in Standard ML.

(*b*) Write two structures implementing this signature: one using a single list, and another one using a pair of lists (with amortised constant time for its operations, as covered in *Foundations of Computer Science*). You should use the same kind of signature constraint for both of them.

(*c*) Did you use an opaque or transparent signature constraint? What difference does it make?

- (a):

```
module type Queue = sig
  type 'a queue
  val qempty : 'a queue (* an empty queue *)
  val qnull : 'a queue -> bool (* test queue is empty *)
  val qhd : 'a queue -> 'a (* get head element of queue *)
  val deq : 'a queue -> 'a queue (* dicard head element, return queue *)
  val enq : 'a queue -> 'a -> 'a queue (* add element at end of queue *)
  val norm : 'a queue -> 'a queue (* normalise a queue *)
end
```

- (b):

```
module QSinList : Queue = struct
  exception Empty
  type 'a queue = Q of 'a list
```

```
  let qempty = Q([])
  let qnull q = (q == Q([]))
  let qhd = function
    |Q(x::xs) -> x
    |_ -> raise Empty
  let deq = function
    |Q(x::xs) -> Q(xs)
    |_ -> raise Empty
  let enq (Q(l)) x = Q(l @ [x])
  let norm q = q
end

module QDoubList : Queue = struct
  exception Empty
  type 'a queue = Q of 'a list * 'a list
  let qempty = Q([],[])
  let qnull q = (q = Q([],[]))
  let qhd = function
    |Q(x::xs,tls) -> x
    |_ -> raise Empty
  let norm = function
    | Q([],tls) -> Q(List.rev tls, [])
    | q -> q
    let deq = function
    |Q(x::xs,tls) -> norm (Q(xs,tls))
    |_ -> raise Empty
  let enq (Q(hds,tls)) x = norm (Q(hds,x::tls))
end
```

- (c):
  - I used an opaque signature because the structures are free to use any implementation for `queue`, which could be `'a list` or `'a list * 'a list` etc.
  - This allows flexible implementations of the same data structure, and we can use the same signature inside algorithms, the change of underlying implementation will not affect those algorithms.

💡 Comments: