German University in Cairo
Computer Science Department
Prof. Dr. Slim Abdennadher

June 25, 2005

Concepts of Programming Languages
Spring term 2005

Final Exam

Bar Code

Instructions: Read carefully before proceeding.

1) Duration of the exam: **3 hours**.
2) No books or other aids are permitted for this test.
3) Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem. Do not put part of the answer to one problem on the back of the sheet for an other problem, since the pages may be separated for grading.
4) This exam booklet contains 16 pages, including this one. Three extra sheets of scratch paper are attached and have to be kept attached.
5) When you are told that time is up, stop working on the test.

Good Luck!

Don't write anything below ;-)

Exercise	1	2	3	4	5	6	7	8	9	10	11	$\sum$
Possible Marks	8	8	6	10	10	6	6	6	8	8	8	75
Final Marks												

Exercise 1 (8 Marks)

a) Give two main differences between the functional and imperative paradigms.

Solution:

The imperative paradigm is characterised by the storage of values in variables, while the functional paradigm does not have this notion of storage. Programs in the imperative paradigm consist of statements, which are blocks of instructions to be executed, and which update the state of the machine executing the program; the functional paradigm does not have state or instructions, and computation is achieved by rewriting expressions.

b) What is an abstract data type?

Solution:

An abstract data type is a set of values, together with a set of operations for manipulating these values. The implementation of the data type is effectively hidden, and the user of the type manipulates these values only through the given operations.

c) In functional languages, what is meant by currying?

Solution:

Currying is the technique of transforming a function taking multiple arguments into a function that takes a single argument (the first of the arguments to the original function) and returns a new function that takes the remainder of the arguments and returns the result.

d) Write the letter corresponding with the application domain each programming language was designed to support in the space provided.

A: Artificial Intelligence

B: Systems Programming

C: Software Engineering

Programming Language	**Application Domain**
C	**B**
Java	**C**
Prolog	**A**
Haskell	**A**

Exercise 2 (8 Marks)

Consider the following declarations in a C program

```
int a = 15, b= 20;
int *p = &a, *q;
int **s = & p;
```

a) Draw a box and pointer diagram showing the memory allocations and contents of the variables declared above.

Solution:

```
a          b
 --         --
|15|      |20|
 --         --
 ^
 |
 |
p         q
 --        --
|  |     |  |
 --        --
 ^
 |
 |
s
 --
|  |
 --
```

b) What would be printed out when the following code is executed.

```
q = p;
*q = 25;
printf("%d\t%d", *p, a);
```

Solution:

Assigning the value of `p` to `q` means that `q` is pointing to `a`. So the assignment changes `a` and that is reflected in both the output values.

```
25      25
```

c) What would be printed out when the following code is executed after all of the above code is executed.

```
a = b;
p = (int*) malloc( sizeof(int));
*p = 13;
printf("%d\t%d", *q, **s);
```

Solution:

The first assignment copies the value `b` (20) into `a`. Creating new memory for `p` means that the 13 is stored in a new location. Since `s` points to `p`, dereferencing twice gets the new value.

```
20      13
```

d) What is the use of `malloc()`? Why not just declare arrays and use them?

Solution:

We don't always know how big should we declare an array. If we overestimate, this results in a waste of memory. If we underestimate, the program is out of space and may crash. With `malloc()`, you can get just as little or as much as you need dynamically (i.e., at run-time). Also, a program compiled to use a huge array may not run on a computer with a small memory. With `malloc()`, you know at run-time that you're out of memory (`malloc()` returns `NULL`) and you can give the user options to lower the memory requirement, use a different algorithm, or just quit.

Exercise 3 (6 Marks)

Consider the following structure definition:

```
struct student {
   char *name;
   int id;
};
```

a) Declare a variable, `s1` of type `student`.

b) Set the name and id of `s1` to "Sara" and 1213 respectively.

c) Declare a pointer `p` and set it to point to `s1`.

d) Print both `name` and `id` of s1.

Solution:

```
#include <stdio.h>
struct student{
 char *name;
 int id;};
int main()
{
     struct student s1;
     s1.name = "Sara";
     s1.id = 1231;
     struct student *p;
     p = &s1;
     printf("Name: %s \n",s1.name);
     printf("ID: %i \n",s1.id);
     getch();
 }
```

Exercise 4 (10 Marks)

Object oriented languages provide encapsulation, polymorphism, and inheritance.

a) What is *encapsulation* and how can a class provide it?

Solution:

Encapsulation separates an interface from an implementation; it ensures that a client can perform only specified and valid operations on an object.

A class can provide encapsulation by appropriate use of `private` and `protected` qualifiers. However, encapsulation ultimately depends on how the programmer uses these features: a class that provides `get` and `set` methods for each of its instance variables is not encapsulated, even if all of the instance variables are `private`.

b) Describe two kinds of *polymorphism* used in object-oriented programming.

Solution:

In each case, polymorphism means that one name can refer to more than one function. The differences are in the way the name is matched to the appropriate function.

1. Inheritance Polymorphism: a single name is associated with different methods in a class and its subclasses. When a method is invoked for an object at run-time, the class of the object determines which method is used
2. Overloading: There are several methods with the same name but different signatures (that is, different type and or number of parameters). The appropriate function is chosen by the compiler, using the type of the arguments. For example, `2+7` would call integer addition but `2.0+7.5` would call floating-point addition
3. Parametric Polymorphism: The same method can be used with many types of arguments, because it does not depend on particular properties of the type. For example, a function that finds the length of a list does not care about the type of the components of the list. The idea can be executed, as in Haskell, so that a function might have a parameter of any type that provides an equality comparison, for example.

c) Distinguish *single inheritance* and *multiple inheritance*.

Solution:

With single inheritance, a child class inherits features from exactly one parent class. Java uses single inheritance.

With multiple inheritance, a child class may inherit features from more than one parent class. C++ uses multiple inheritance.

d) What problems arise in implementing multiple inheritance?

Solution:

A child class may inherit two or more different features with the same name from different parents. Even worse, a child class may inherit the same feature by more than one route.

e) What purpose do *interfaces* serve in Java?

Solution:

Interfaces provide a compromise between single and multiple inheritance. Interfaces provide a way of specifying behaviour without constraining the way in which the behaviour is achieved.

Java allows a class to implement more than one interface but to inherit only one class. The problem of inheriting two different methods with the same name from different parents no longer exists.

Exercise 5 (10 Marks)

The class `Arrays` of the `java.util` package contains a static method `sort`, which can be used to sort arrays. For example, the following piece of code:

```
int [] x = {1,4,2,3};
Arrays.sort(x);
for(int i = 0; i < x.length; i++)
        System.out.print(x[i] + " ");
```

prints `1 2 3 4` to screen.

The same method `sort` is overloaded so it can sort any array of objects, for example,

```
String [] y = {"d", "c", "a", "b"};
Arrays.sort(y);
for(int i = 0; i < y.length; i++)
  System.out.print(y[i] + " ");
```

prints `a b c d` to screen.

However, the documentation for this method states the following:

```
public static void sort(Object[] a) sorts the specified array of
objects into ascending order, according to the natural ordering of its
elements. All elements in the array must implement the Comparable
interface.
```

The following is the documentation for the interface `Comparable`:

```
public interface Comparable
     Method Summary:
        int compareTo(Object o)
        Compares this object with the specified object for order. Returns a
        negative integer, zero, or a positive integer as this object is less
        than, equal to, or greater than the specified object.
```

Given the following Book class:

```
 class Book
 {
  String title;
  int id;

  public Book(String t, int i)
  {
  title = t;
  id = i;
  }
 }
```

Implement a class `CompBook`, which is identical to class Book, except that an array of `CompBook` can be sorted using `Arrays.sort()`. The resulting array should be sorted ascendingly according to `title` then `id`, for example:

```
 title: "Programming Languages", id: 4
 title: "Computer Graphics", id: 1
 title: "Programming Languages", id: 3
 title: "Java Unleashed", id: 8
```

would be sorted into

```
title: "Computer Graphics", id: 1
title: "Java Unleashed", id: 8
title: "Programming Languages", id: 3
title: "Programming Languages", id: 4
```

Note: You are not able to sort instances of class `Book` using `Arrays.sort()`.

Do not rewrite any code that is not essential for accomplishing your task. The class Book should be reused.

Solution:

```
import java.util.*;

class Book
{
 String title;
 int id;

 public Book(String t, int i)
 {
 title = t;
 id = i;
 }
}

class CompBook extends Book implements Comparable
{
 public CompBook(String t, int i)
 {
 super(t, i);
 }
 public int compareTo(Object a)
 {
 CompBook other = (CompBook) a;
 return (title + id).compareTo(other.title + other.id);
 }

 // The following part is only for demonstrating the correctness of the
 // solution.
 public static void main(String args[])
 {
 CompBook [] x =
 new CompBook [] {new CompBook("Programming Languages", 4),
  new CompBook("Computer Graphics", 1),
  new CompBook("Programming Languages", 3),
  new CompBook("Java Unleashed", 8)};
 Arrays.sort(x);
 for(int i = 0; i < x.length; i++)
 {
 System.out.println(x[i].title + "\t" + x[i].id);
 }
 }
}
```

Exercise 6 (6 Marks)

Write a Prolog predicate `repeats(Xs,Ys)` that succeeds when `Ys` is the list of distinct items that appear more than once in the list `Xs`, i.e. one occurrence in `Ys` of each repeated item in `Xs`. Your predicate should be able to solve for `Ys` given `Xs`.

For example, the query

```
?- repeats([a,b,c,c,a], Bs).
```

should succeed with

```
Bs = [a,c]
```

Hint: You may use the following `member` and `delete` predicates:

```
member(X,[X|_]).
member(X,[Y|L]) :- member(X,L).

delete(X,[ ],[ ]).
delete(X,[X|Xs],Ys) :- delete(X,Xs,Ys).
delete(X,[Y|Xs],[Y|Ys]) :- X\==Y, delete(X,Xs,Ys).
```

Solution:

Recursive solution checks each item to see if it is repeated using `member`. If so, retain in answer, delete other occurrences, and continue; if not, just continue with next item.

```
repeats([ ],[ ]).                     % stopping case
repeats([X|Xs], [X|Ys]) :- member(X,Xs), delete(X,Xs,Zs), repeats(Zs,Ys).
repeats([X|Xs], Ys) :- \+member(X,Xs), repeats(Xs,Ys).
```

Exercise 7 (6 Marks)

Assume that you have the implementation of the predicate `insert(E,List1,List2)` that succeeds if `List2` is the result of inserting `E` in `List1` in order.

Implement a Prolog predicate `sort(List1,List2)` that succeeds if `List2` consists of the same elements as in `List1` but in order.

Hint: You may use the accumulator technique.

Solution:

```
sort1(List1, List2) :-
     sort2(List1, [], List2).

sort2([], L, L).

sort2([H|T], List1, L):-
     insert(H, List1, List2),
     sort2(T, List2, L).
```

Exercise 8 (6 Marks)

Consider the following Haskell definition:

```
mystery [] = []
mystery ((x,y):ps) = x : mystery ps
```

a) Give a step-by-step reduction of `mystery [(1,2),(3,4),(5,6)]`.

Solution:

```
mystery ((1,2) : (3,6) : (5,6) : [])
  =>
    1 : mystery ((3,6) : (5,6) : [])
  =>
    1 : 3 : mystery ((5,6) : [])
  =>
    1 : 3 : 5: mystery []
  =>
    1 : 3 : 5 : []
  =>
   [1,3,5]
```

b) What does the function return for any list of pairs?

Solution:

For any list of pairs, the function returns a list of the first elements in the pairs.

c) Determine the type of the function `mystery`.

Solution:

```
mystery :: [(a,b)] -> [a]
```

Exercise 9 (8 Marks)

a) Write a Haskell function `map2 f list1 list2` that for a binary function `f` and two lists `list1` and `list2` with the same length, applies the function `f` on the elements of both lists and returns a list as a result. For example

```
map2 + [1,2,3] [4,5,6] ==> [5,7,9]
```

Solution:

```
map2 f [] [] = []
map2 f (x:xs) (y:ys) = f x y : (map2 f xs ys)
```

b) Determine the type of `map2`.

Solution:

```
map2 :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Exercise 10 (8 Marks)

a) Write a Haskell function `forsome` that takes a predicate `p` (i.e. boolean function) and a list as arguments and returns `true` if the predicate satisfies at least one element of the list, otherwise it returns `false`.

```
forsome (> 4) [1,2,3,4,5]   ======> True
forsome (> 4) [1,2,3]       ======> False
forsome (> 4) []            ======> False
```

Solution:

```
forsome p [] = False
forsome p (x:xs) = if p x then True
                    else forsome p xs
```

b) Write a Haskell function `forall` that takes a predicate `p` (i.e. boolean function) and a list as arguments and returns `true` if the predicate satisfies all elements of the list, otherwise it returns `false`.

```
forall (> 4) [6,7,8]   ======> True
forall (> 4) [1,2,5]   ======> False
forall (> 4) []        ======> True
```

Solution:

```
forall p [] = True
forall p (x:xs) = if (p x) == False then False
                  else forall p xs
```

Exercise 11 (8 Marks)

An initial segment of a list l is a list `x` such that for some list `y`, `l = x++y`. For example, the initial segments of `[1,2,3]` are

```
[], [1], [1,2], [1,2,3]
```

a) Write a Haskell function `inits` that takes a list and returns the list of all its initial segments. For example, `inits [1,2,3]` will return

```
[[], [1], [1,2], [1,2,3]]
```

Hint: You may use the predefined higher-order function `map`.

Solution:

```
inits [] = [[]]
inits (x:xs) = [] : map (x:) (inits xs)
```

b) Determine the type of `inits`.

Solution:

```
inits :: [a] -> [[a]]
```