Concepts in Programming Languages
Exercise Sheet

Andrej Ivašković (ai294)
Compiled on: 30th April 2019

Before attempting the problems

This exercise sheet covers most of the examinable material in this course. Some of the questions are more on the factual side, some require writing or understanding code, and others are discussion questions (which you should answer by giving reference to the languages covered in the course, or reading additional material). It would be inappropriate to work through all of these exercises in only two supervisions; your supervisor should assign an adequate subset of these exercises.

A lot of the concepts are best learnt by writing programs. I suggest you download the following compilers/interpreters and write some code to see the ideas covered in the course in action:

- Standard ML: you probably already have a Standard ML interpreter from the *Foundations of Computer Science* course (most likely PolyML)
- OCaml: `http://www.ocaml.org/docs/install.html`
- Fortran: on Linux, install `gfortran`
- Lisp: either Common Lisp (`clisp` package on Linux) or Scheme (`scheme` package on Linux)
- Pascal: `https://www.freepascal.org/download.html`
- Haskell: `https://www.haskell.org/platform/` (for examples with monads)

Some of these questions have been taken from a set devised by Alan Mycroft.

1 The ancestors

Exercise 1.1. What do we mean by *abstract machine* when describing a programming language? Outline the basic characteristics of the abstract machines of:

(*a*) FORTRAN (the original 1957 variant)

(*b*) LISP (the original McCarthy dialect)

(*c*) ALGOL 68

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.]

Exercise 1.3. The two most famous variants of LISP are Common Lisp and Scheme.

(*a*) Scheme has a smaller set of keywords and a much shorter specification than Common Lisp. Why might either one of these design decisions be desirable?

(*b*) Both of these modern dialects feature *static scoping*, whereas McCarthy's original used *dynamic scoping*. Give an example of an expression whose differs depending on the type of scoping used.

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 these constructs would allow you to construct a LISP interpreter written in LISP.

(*c*) How do these features interact with scoping?

Exercise 1.5. LISP contains a `funcall` keyword, which is similar to `apply`, but assumes the arguments of the function in question are not in a list. This means that

```
(apply (lambda (x) (* x 2)) '(3))
```

can be alternatively written as

```
(funcall (lambda (x) (* x 2)) 3)}
```

Consider the following LISP program:

```
(defun mystery (f x)
    (cond ((consp x) (cons (funcall f (car x))
                           (mystery f (cdr x))))
          (T nil)
    )
)
(mystery (lambda (x) (+ x 1)) '(10 20 30))
```

What Standard ML function does `mystery` resemble? How would you write it using `apply` instead of `funcall`?

Exercise 1.6. Give a brief overview of the following parameter passing mechanisms, and provide code that will distinguish between them: *call by value*, *call by reference*, *call by value/result*, *call by name*, *call by text*. Make sure you use the terms *aliasing*, *formal parameter* and *actual parameter* in your explanations.

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?

Exercise 1.8. What are the four distinguishing characteristics of object-oriented programming languages? How are they desirable when creating physics simulation software?

Exercise 1.9. A commonly used practice in object-oriented programming is *encapsulation*. You have already informally defined it in *Object-Oriented Programming* as 'a class should expose a clean interface that allows full interaction with it, but should expose nothing about its internal state or how it manipulates it'. SIMULA 67 had no facilities to achieve encapsulation. How did Smalltalk improve on this? Comment on why encapsulation might be desirable.

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).

Exercise 1.11. One of the most commonly used early programming languages was COBOL. Edsger W. Dijkstra famously wrote in *How do we tell truths that might hurt?* (EWD498):

> The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense.

Write an explanation, in no more than 200 words, justifying or criticising both Dijkstra's comment and COBOL being omitted from this course. You might want to read a bit more about COBOL in order to answer this question.

Exercise 1.12. Brian Kernighan, in his essay *Why Pascal is Not My Favorite Programming Language* criticises Pascal's design decisions[1]: the language did not feature any `break` and `continue` statements. Provide an argument for and an argument against this design decision.

[1] This was Pascal as defined by Niklaus Wirth. Some of these criticisms have been addressed since then.

2 Types

Exercise 2.1.

(*a*) What do we mean by *strong typing*?

(*b*) Show how hard-to-find errors can result from the absence of strong typing.

(*c*) Distinguish *static typing* and *dynamic typing*.

(*d*) Does static or dynamic typing imply strong typing?

(*e*) Is Java completely statically typed or completely dynamically typed?

Exercise 2.2. We observe three different meanings of the word *polymorphism* in this course. Show how they are exemplified in Java and give their alternative names when they exist.

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
```

Exercise 2.4. In the context of subtyping, explain the words *invariant*, *covariant* and *contravariant*.

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.

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;
```

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

Why is this error message necessary for type safety?

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

```
import java.util.ArrayList;
import java.util.Arrays;

class Fruit {
  int weight;
}

class Apple extends Fruit {
  boolean isRed;
}

public class Foo {
   public static void main(String[] args) {
      System.out.println("Starting...");
      Fruit f = new Fruit();
      Apple a = new Apple();

      System.out.println("Simple casting:");
      Fruit OKf = a;
//      Apple ERRORa = f;

      // olde-style arrays:
      Apple[] av = new Apple[10];
      Fruit[] fv = new Fruit[10];
      // make ArrayLists containing the same elements as the arrays av,fv:
      ArrayList<Apple> al = new ArrayList<Apple>(Arrays.asList(av));
      ArrayList<Fruit> fl = new ArrayList<Fruit>(Arrays.asList(fv));
      System.out.println("Checking: al.size="+al.size()+"; fl.size="+fl.size());

      // now explore variance ...
//      ArrayList<Fruit> ERRORq = al; //ERROR!
      ArrayList<? extends Fruit> p = al;
      ArrayList<Fruit> q = fl;
      Fruit gotf = p.get(3);
//      p.set(3,f);    // ERROR!
      q.set(3,f);

      System.out.println("Olde-style arrays and variance:");
      Fruit[] r = av;
      r[3] = f;

      System.out.println("Stopping");
   }
}
```

Exercise 2.8. What is a *scripting language*?

Exercise 2.9.

(*a*) Explain how JavaScript code is executed in a browser.

(*b*) Explain how event-driven code is written in JavaScript using callbacks.

Exercise 2.10. Compare and contrast *duck typing* and dynamic typing. Why do they tend to be features of scripting languages?

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?

Exercise 2.12. How do Java interfaces differ from ML signatures, and classes from structures?

3 Further concepts

Exercise 3.1. How can theoretical models of concurrency (such as CSP, CCS, π-calculus and PRAM) provide insight about writing concurrent programs?

Exercise 3.2.

(*a*) Why might threads be hard for users and compilers to reason about?

(*b*) Why might Cilk's construct be 'better'?

Exercise 3.3. Investigate the notion of *autovectorization* and write a brief explanation of what it is. How does this it relate to OpenMP's facilities for concurrency?

Exercise 3.4. [2014 Paper 3 Question 6, part (*b*)] You manage two junior programmers and overhear the following conversation:

> A: "I don't know why anyone needs a language other than Java, it provides clean thread-based parallel programming."
>
> B: "Maybe, but I write my parallel programs in a functional programming language because they are then embarrassingly parallel."

Discuss the correctness of these statements and the extent to which they cover the range of languages for parallel programming.

Exercise 3.5. Suppose you were tasked with creating an index that maps all of the K known keywords to the documents and positions in which they appear. You are given N documents that are used to create this index. The expected time of constructing an index given a single document is T seconds, and you

have already written code that does this. Give a rough estimate of the total running time for computing the index when the computation is adapted to work with MapReduce on N machines concurrently. Estimate the running time of a program that computes this index a single CPU. What is the relative improvement?

Exercise 3.6. Define the terms *internal iteration* and *external iteration*. Why might one be better than the other for exploiting parallelism?

Exercise 3.7. What is the *expression problem*? How does it affect the ease of making small changes to a program spread over many files?

Exercise 3.8.

(*a*) What is a *monad*? What are its operations?

(*b*) Distinguish between a side-effecting function, a pure function, and a 'computation' value in a monad.

Exercise 3.9.

(*a*) Write a Standard ML signature `MONAD` that represents a monad. This signature should use a `'a m` type to represent a monadic computation.

(*b*) Show that you can define a `List` structure that transparently matches this signature.

(*c*) Use the operators in this monad in order to implement a `concat` function of type `'a List.m List.m -> 'a List.m` that 'flattens' a list of lists.

Exercise 3.10. Assume the existence of an IO monad in a functional language.

(*a*) Give the types of expressions which:

(*i*) read a line from `stdin`;

(*ii*) read a line from a file specified by parameter `f`;

(*iii*) write a line to `stdout`;

(*iv*) write a line to a file specified by parameter `f`.

(*b*) Given values `c` and `n` of type `unit IO` and `int` respectively, give a program which performs `c`

(*i*) twice;

(*ii*) `n` times.

Exercise 3.11. How does a GADT enable users to represent data structures more precisely than ML can? Why might this be useful?

Exercise 3.12.

(*a*) Rewrite `foo` into CPS (as `foo_cps`), assuming `f`, `g` are also rewritten into CPS and `f`'s first argument is now a function what is expected to be in CPS:

```
fun foo b f g x = if b then 17
                  else f (fn y => g (g (y - 1))) (x + 1)
```

(*b*) Consider the following function declaration (in a hypothetical ML-like language with reified continuations):

```
1 fun list_prod [] = raise Index
2   | list_prod [x] = x
3   | list_prod (x::xs) = callcc
4       (fn k => if x = 0 then k 0
5                 else x * list_prod xs)
```

What is the purpose of `callcc` here? How can you achieve the same behaviour using exceptions?