Concepts in Programming Languages

Exercises

2021

Contents

This course briefly covers some old and new languages, and discusses their history, innovations and influence. It is very instructive to extend this analysis to other languages you are familiar with, be they ones you studied as part of the Tripos or ones you have learnt independently. Therefore, when answering the questions below, you are very much encouraged to think about and refer to languages not discussed in this course, and consider how they fit in the timeline of PL research and design, where their features originate and what new concepts they introduced.

1. Meet the ancestors

1. The notes mention that in the 1950s the utility of high-level programming languages was open to question, so there was some reluctance to the adoption of FORTRAN. This skepticism might seem surprising to us in hindsight. What objections could an Assembly programmer at the time have against high-level languages, and how would you try convincing them to switch to FORTRAN?

2. An author writes:

   Most successful language design efforts share three important characteristics:

   - *Motivating application*: the language was designed so that a specific kind of program could be written more easily.
   - *Abstract machine*: there is a simple and unambiguous program execution model.
   - *Theoretical foundations*: theoretical understanding was the basis for including certain capabilities and omitting others.

   Briefly discuss the merits and/or shortcomings of the above three statements, giving examples and/or counterexamples from procedural, functional, logic, and/or object-oriented programming languages.

3. Consider the following two program fragments.

```
let x = 1 ;                         ( defvar x 1 )
let g(z) = x + z ;                  ( defun g(z) (+ x z) )
let f(y) =                          ( defun f(y)
  g(1) +                              (+ (g 1)
  (let x = y + 3                         ( let ( ( x (+ y 3) ) )
  in g (y+x)) ;                                ( g (+ y x) ) ) ) )
f(2) ;                              ( f 2 )
```

   What are their respective output values when run in their corresponding interpreters? Justify your answer, explaining it in a conceptual manner.

4. Give a brief overview of the following parameter passing mechanisms, demonstrating their similarities and differences with (pseudo)code examples: *call by value*, *call by reference*, *call by*

Some questions by Andy Rice (acr31) and Andrej Ivašković (ai294).

value/result, *call by name*, *call by text*. Make sure you use the terms *aliasing*, *formal parameter* and *actual parameter* in your explanations.

5. 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, and how it fits in with the other distinguishing characteristics of object-oriented languages.

2. Types in programming languages

1. *Alice*: 'Urgh, I'm so sick of these ML type errors… Why can't I just test my code without all these types getting in my way?'

   Bob: 'At least you are getting type errors! I've been writing a Javascript web app and it's like I'm walking around in a dark forest filled with traps…'

   Continue the debate above about the merits and drawbacks of type systems. If you want, bring in a third person into the conversation, representing a different viewpoint or typing discipline.[1]

2. For the programming languages FORTRAN, LISP, Algol, Pascal, C, ML and Java, briefly discuss and evaluate their typing disciplines. Further compare the advantages and disadvantages that their designs impose on the programmer.

3. 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.

4. Use the type inference algorithm described in the notes to find the type of the following OCaml expression: `fun x -> fun y -> fun z -> z (x y) y`

5. Explain the meaning and the context of the terms *covariance*, *contravariance* and *invariance*. Give examples in ML and Java to demonstrate why this distinction is needed. # Scripting and modularity

6. What is a scripting language? What kinds of applications *should* scripting languages be used for? In contrast, what kinds of applications *are* they used for today?

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

8. Compare and contrast module systems with principles of object-oriented programming such as abstraction and inheritance. How do ML structures and signatures differ from OOP classes and interfaces? Compare these with Haskell's type classes, if you are familiar with them.

9. 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 one using a pair of lists (with amortised constant time for its operations, as covered in Foundations

[1] Or you can make a boring old table as well. It's exam term. I'll understand.

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?

3. Further concepts

1. You manage two junior programmers and overhear the following conversation:

   Alice: “I don’t know why anyone needs a language other than Java, it provides clean thread-based parallel programming.”

   Bob: “Maybe, but I write my parallel programs in a functional programming language because they are embarrassingly parallel.”

   Discuss the correctness of these statements and the extent to which they cover the range of languages for parallel programming. *Note*: this was an exam question in 2014, *after* the release of Java 8 – make sure to consider this in your answer to the first part of the question.

2. Describe the *expression problem* and how it relates to data types in functional and object-oriented languages, with the help of code examples. How can both of the “competing” approaches be implemented in Scala? Can you think of a time when you encountered the expression problem in your own projects?

3. Reason for and/or against the following statement:

   Functional programming is the future.

4. 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.

5. 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 : unit IO` and `n : int`, give a program which performs `c`
      (i) twice;
      (ii) `n` times.

6. Suppose you want to define a datatype for lists containing *alternating* integers and Booleans, e.g. `[5, true, 2, false, 9, true]`. How can you statically enforce this alternation property using GADTs?

7. Briefly explain the intuition behind using CPS to represent a function of type `a -> b` as `(b -> unit) -> (a -> unit)`. How and why does contravariance come into play?