Informatics Board of Studies — Course Proposal

Proposed course title: Elements of Programming Languages
Proposer(s): James Cheney
Date: January 14, 2015
Status: REVISED DRAFT for discussion at Board of Studies

1 Case for Support

1.1 Overall contribution to teaching portfolio

Programming language design and implementation (PL) is a broad research area that interacts with many other areas of computer science, AI and software engineering. It is ranks third largest among core areas in the ACM CS2013 curriculum, behind “Software Development Fundamentals” and “Data structures”, and second behind “Software Engineering” in terms of advanced “tier 2” material. PL corresponds to the “Comparative programming languages” area of the QAA Subject Benchmark for Computing.

![ACM CS2013 curriculum summary](image)

Figure 1: Summary of ACM CS2013 curriculum tier1 and tier2 hours, by subject area

The proposer carried out a curriculum analysis of PL-related courses in early 2014 to match our existing offerings against the ACM guidelines (this analysis is attached to this proposal). Currently only 12 of the ACM-recommended 28 hours of core PL material are covered in required courses such as INF1-FP, INF1-OOP and INF2A, and an additional 5-6 hours of core PL material is covered in optional honours courses such as CT and ITCS. This leaves roughly 10 hours of core material uncovered by any Informatics course, and that is the best case: it assumes that students take both CT and ITCS (currently, fewer than 10 students are enrolled in both courses in Spring 2015). Thus, the vast majority of our students currently only see 12–15 of the 28 recommended hours of material.

Although it might (at first glance) seem easy to fill this gap merely by adding an extra lecture or two to several existing courses, this would not really address the problem. This material is mutually reinforcing so simply adding piecemeal coverage to other courses would be far less effective than presenting it as part of a unified topic. Moreover, even if material were added to current PL-related courses to fill this gap, students would still not see this material unless they took all of these courses; this is unlikely because the more specialised PL-related UG4 electives are typically taken by under 30 students.

Therefore, we propose to develop a new course that covers the 10 hours of material that our students currently do not have any opportunity to learn. To round out the course, the remaining approximately 5 hours of examinable material can be designed to cover the ACM curriculum areas already covered by other optional courses CT and ITCS (from a different perspective, and without duplicating material.) The course will emphasise practical issues, complementing our existing slate of more theoretical or advanced courses, and will be designed to appeal to students who might not otherwise take CT, ITCS, or more advanced PL-related courses.

**Previous courses:** Two third year courses, Language Semantics and Implementation (LSI) and Functional Programming and Specification (FPS) used to cover this area. These courses are
no longer being taught, in part due to declining demand. However, there is no evidence that the
demand in demand for those two courses is due to decline in interest in PL topics more generally;
other PL-related courses such as CT, PPLS, APL, LP, COC, TSPL, or SP — some of which used to
depend on LSI or FPS as prerequisites — have not seen a comparable decrease in demand.

**Demand:** New programming languages (including scripting languages and domain-specific
languages) are appearing all the time, and we cannot (and probably should not try) to teach
all of them. Instead, teaching students the underlying ideas of programming language design
will prepare them to learn new languages more quickly and effectively; this does not just mean
knowing how to use existing features correctly but also what features to avoid and why their use
is problematic.

Although few CS graduates will ever design a new general-purpose language, many will be
faced with the task of learning, adapting or designing a new scripting language or domain-specific
language. Currently, many such languages are not consciously designed but instead grow from a
custom tool to meet some need; this usually involves reinventing known features, and sometimes
results in re-making known mistakes. It is important for computer science professionals to be
aware of programming language design as a coherent discipline, so that they know where to look
for ideas if they are faced with a similar task.

PL concepts arise in many research areas in the School, and familiarity with techniques
such as abstract syntax, interpreters, domain specific languages, translators, and types would
contribute to potential UG or MSc projects in several areas beyond core PL, such as databases
(query languages), natural language processing, security, networks, and quantum computation.

**Competition:** Peer UK and international institutions all offer courses on PL concepts, for
example:


and some of these (e.g. CMU’s 15-312) are listed as exemplars in the ACM CS curriculum.

The University of Edinburgh is known worldwide for fundamental contributions to the foun-
dations of programming languages, including many topics that are now routinely taught in the
courses listed above. We can and should offer one of the world’s best courses on programming
language design to our own students, whether or not they go on to pursue more advanced topics
offered by our UG4 courses on PL.

### 1.2 Target audience and expected demand

**Background:** UG3 or UG4 students having taken Inf1-FP, Inf1-OOP, Inf2A.

**Level of ability:** Medium to high. The course would be demanding.

**Interests of students likely to take this class:** Programming, programming languages,
symbolic computation, compilers, interpreters.

**Expected class size:** This course would likely appeal to students that currently take CT,
PPLS, COC, TSPL, APL, LP, AR, and ITCS, as well as to students who might be interested in
learning about programming languages but might not feel prepared to take the more advanced
courses we currently offer. The enrolment figures for these courses for the current and previous
academic year are:
where italics indicate enrolment figures for Spring 2015 courses that are not finalised at the time of writing. Based on comparable UG3 courses such as CT, LP, CT, and ITCS, it seems reasonable to anticipate initial class size in the range 15–25 students. This could increase if the course design is successful in attracting students who might not naturally have taken other upper-level courses on this topic, and may have the (desirable) side-effect of increasing demand for these courses or for honours projects in this area.

1.3 Relation to existing curriculum

Fit with existing courses The course is tentatively planned to be taught in semester 1, normally to be taken in UG3, but also open to UG4 or suitably interested MSc students. The following diagram indicates how it fits into the current curriculum:

In more detail:

- EPL will **build upon and consolidate** knowledge of functional programming, object-oriented programming, and formal languages established in INF1-FP, INF1-OOP and INF2A.
- EPL will **complement** other Honours courses normally taken in UG3 related to programming languages, such as CT, LP, and ITCS. Specifically:
  - Compiling Techniques focuses on parsing, compilation and low-level code generation, and EPL will complement this with understanding of interpreters, translation and high-level language features
  - Logic Programming focuses on Prolog and logic as a foundation for programming, while EPL will complement this with broader understanding of programming language design ideas (including declarative/functional/logic programming, object-oriented programming, etc.)
  - Introduction to the Theory of Computer Science includes 4-5 hours of coverage of lambda calculus and types from a theoretical perspective, including topics such as
type inference that may be covered more broadly and less formally in EPL. EPL will complement this material by presenting pragmatics about language design, interpretation and translation. The two courses would be independent, and could be taken in either order.

- **EPL will strengthen students’ preparation for advanced PL courses** normally taken in UG4, such as APL, TSPL and COC, and may increase demand for these courses or motivate additional offerings at UG4 or MSc level.

- **EPL will also strengthen students’ preparation for PL-related UG4 projects** involving language design or implementation as a component or main focus.

- **EPL may also be a helpful complement to CT or useful preparation for UG4 courses such as SP, PPLS and COPT, but this is not an explicit design goal, since the progression of students from CT and CS in UG3 to these UG4 courses appears to be healthy already.**

- Likewise, EPL would complement AR. AR is currently taught in UG4 but I am aware of tentative plans to move it to UG3 and add a fourth year Formal Verification course. One important application area for AR and formal verification techniques is proving properties of programs or programming languages, and EPL would provide useful background for such material as well as potentially attracting students to the area. Students taking both EPL and AR will be well-prepared for the new Formal Verification course as well as the advanced course TSPL, in which programming language concepts are formalised in a theorem prover.

**Degrees and specialisms**  I expect this course to contribute to:

- All undergraduate Degree Programmes spanning Computer Science and Software Engineering

- The MSc specialisms of Computer Systems, Software Engineering & High-Performance Computing, Design Informatics and Theoretical Computer Science. (It is relevant to Design Informatics in that programming language design often involves some aspects of aesthetics as well as technological problem-solving.)

This is based on the fact that the course covers core computer science concepts.

### 1.4 Resources

- **Lectures**: 18–20 per semester.

- **Tutoring**: 10 hours (one 2-hour introductory lab session + 8 smaller group tutorials)

- **Exam preparation**: As for a typical course taught in the autumn term. Anticipating the course would be open to visiting undergraduates, an extra December exam would need to be prepared whenever visiting undergraduates take the course.

- **Course organisation team**: 1-2 lecturers + 1-2 tutors/markers/lab demonstrators. In its first year it is anticipated that the course would be delivered by the proposer, to ensure consistency in its development. The proposer is currently supported by a fellowship until December 2016, but is willing to voluntarily accept this teaching load.

- **Alternative lecturers**: Because this material is close to the research interests of several Informatics teaching staff, it is expected that it would be straightforward to find alternative lecturers.
2 Course descriptor

Course Title: Elements of Programming Languages

SCQF Credit Points: 10

SCQF Credit Level: 10

Normal Year Taken: 3

Also available in years: 4/MSc

Subject Area and Specialism Classification:

- Any BSc or MInf degree programme including the words “Computer Science” or “Software Engineering” or “Informatics”.
- MSc specialisms:
  - Computer Systems, Software Engineering and High Performance Computing
  - Design Informatics
  - Theoretical Computer Science

Appropriate/Important for the Following Degree Programmes: This course covers core material recommended for all students taking a Computer Science, Software Engineering or Informatics degree.

Timetabling Information EPL is likely to be of interest to similar students to those taking UG3 level 9/10 courses such as LP, CT, AR, ITCS, and may also be taken in the fourth year by students also interested in courses such as APL, PPLS, COPT, AR, or TSPL. It would be helpful to avoid scheduling conflicts with these courses. For this reason it is proposed that EPL be taught in Semester 1, to avoid undue scheduling pressure, because the majority of these complementary courses (CT, AR, ITCS, PPLS, COPT, TSPL) are taught in Semester 2.

School Acronym: INF-?-???

Short Course Description: Programming languages are unique forms of communication that play a dual role: not only as ways for programmers to instruct machines, but as ways for programmers to talk to each other about computation. Paradoxically, they are among the most permanent features of the computing landscape (Fortran, for example, is still widely used 60 years since its invention), and among the most energetic and innovative, with new programming languages introduced every few months, often aiming to simplify Web programming, parallel, or distributed computing.

Although few computer scientists will ever design a new, general-purpose language like Java or C++, all computer scientists need the ability to learn new languages quickly, recognise and use (or avoid misusing) common language features, and even design new domain-specific languages for restricted problem domains. The design of programming languages involves many subtle choices and tradeoffs among performance, convenience, and elegance. In this course, you will not learn how to use any one language, but instead you will learn the basic elements you need to understand the next 700 programming languages, or even design your own.
Pre-Requisite Courses:  None (beyond required INF1 and INF2)

Co-Requisite Courses:  None

Prohibited Combinations:  None

Other Requirements:
- Undergraduate: Successful completion of year 2 of Informatics Single or Combined Honours degree, or equivalent by permission of the School
- Postgraduate: Available to postgraduates with proven programming ability (e.g., first degree in Computer Science, Software Engineering or similar)

Available to Visiting Students:  Yes

Summary of Intended Learning Outcomes: A student who has successfully completed this course should be able to:

1. investigate the design and behaviour of programming languages by studying implementations in an interpreter;

2. employ abstract syntax and inference rules to understand and compare programming language features;

3. design and implement a domain-specific language capturing a problem domain;

4. understand the design space of programming languages, including common elements of current languages and how they are combined to construct language designs;

5. critically evaluate the programming languages in current use;

6. acquire and use language features quickly, recognise problematic programming language features, and avoid their (mis)use.

2.1 Assessment Information

Assessment Weightings: Written Examination: 75%
Coursework: 25%

Time spent on assignments: One “warmup” lab-based programming exercise, submitted as a basis for formative feedback and not for credit (2 supervised hours in lab + 2 hours additional independent work).

Two assignments each worth 10–15% and taking roughly 8–12 hours each, for a total of 20 hours.
2.2 Academic description

The course first covers general considerations and fundamental concepts used throughout the rest of the course: the idea of abstract syntax and the use of inference rules to model the design of language features. This is expected to cover roughly the first five lectures. The remainder of the course builds on this foundation to explore the elements of programming languages, including data structures and abstractions, control structures and abstractions, and approaches to modularity and program organisation. This exploration is structured as a series of case studies, each presenting a programming problem, and exploring ways in which it has been addressed, ranging from (sometimes naive) early approaches to those taken in modern languages. The intention is not just to denigrate what the instructors believe to be bad design, but to highlight design “mistakes” where it is clear that a particular feature (or combination of features) is prone to misuse, while offering a nuanced view of the many tradeoffs and interactions between reasonable design choices where there is no single “best” way.

For example, Hoare once called null pointers the “billion dollar mistake”. It is common to need a way to represent missing data in a program, but null pointers / null references accommodate this in a way that is particularly hospitable to subtle bugs. This issue reappears in many languages and has led to a large number of crashes and security vulnerabilities causing economic and other forms of damage. This could form the basis of a lecture contrasting the conventional approach to missing data with alternatives such as option types in functional languages, or even NULL values in SQL.

A single programming language will be used as a running case study of language design, with ideas from other languages introduced in lectures as necessary. The same language will be used for the coursework (programming) assignments; a two hour lab session will introduce students to the language used for the course and eight weekly tutorials will cover practical programming exercises or mini-case studies.

Exposure to features arising in a variety of programming languages is important part of the course, however, due to the length of the course it is not judged to be feasible for students to learn two or more unfamiliar languages fluently. Instead it is proposed to select a single programming language to be used for the coursework assignments. Since the purpose of the course is to learn the elements of programming language design, it makes sense to ask the students to learn a non-mainstream language that has a number of interesting features that can be used to illustrate good (or critique questionable) design, as long it is clear we are not endorsing this (or any current) programming language as an ideal. It would also be strategically reasonable to choose a language with some current credibility in industry, though the course will not teach programming in that language so much as use it as a primary case study of a complete language design.

Languages arguably currently meeting the above criteria include Scala, F#, Python and Ruby. Scheme/Racket, OCaml, Haskell or Standard ML would also be sensible choices but are less credible beyond academia. Of these, Scala seems to offer the best tradeoff between familiarity (building on the Java ecosystem), interestingness of design (functional programming, polymorphism, and sophisticated object-oriented features such as traits), and industrial relevance. For the sake of concreteness we propose that the course initially use Scala, but the course is not dependent on the particular programming language — in principle, the coursework and tutorials could be adapted to use any other language.

The course will feature a distinctive practical component aimed at establishing a baseline of literacy in language-oriented problem solving techniques (consolidating and integrating initial exposure to many such ideas in INF1-FP, INF1-OOP and INF2A). There will be three coursework assignments:

- a “warm-up” exercise to (re)familiarise students with functional programming and ensure that they are prepared for the more rigorous programming exercises to come (0% credit, for
feedback only):

- a coursework focusing on implementing an interpreter for a simple language and extending it with some non-standard features (10% credit)
- a coursework focusing on the design and implementation of a domain specific language (e.g. translating high-level geometric primitives to triangles for input to a ray tracer, or simple markup/template language to HTML or LaTeX) (15% credit)

**Syllabus:** (not necessarily to be covered in order)

- Abstract syntax and name-binding; Lexical vs. dynamic scope
- Modelling programming language features using inference rules
- Interpreters as a way of defining and exploring language features
- Domain specific languages, translators, and pragmatics of language processing
- A selection of topics illustrated by case studies from current or historically important language designs:
  - data structures: pairs/record types; variant/union types; recursion
  - abstracting data: overloading; generics; ad hoc vs. parametric polymorphism
  - control structures: goto, loops, case/switch, exceptions
  - abstracting control: procedure call/return; function types; continuations
  - design dimensions: eager vs. lazy evaluation; purity vs. side-effects; state
  - object-oriented features: objects, classes, interfaces, subtyping, (multiple) inheritance
- Advanced topics/guest lectures on concurrent, parallel, distributed programming, security, or verification (linking to UG4 year courses)

**Relevant QAA Computing Curriculum Sections:** Comparative programming languages.

**Transferrable skills:** [No additional explicitly taught transferrable skills]

**Reading List:**

- Essentials of Programming Languages, Friedman, Wand and Haynes
- Practical foundations for programming languages, Harper

In addition online resources or readings specific to the programming language used for the course should be recommended, but need not appear in the course descriptor, for example:

- Programming F# 3.0, C. Smith, O'Reilly, 2013
- Functional Programming in Scala, P. Chisuano and R. Bjarnason, 2015
**Breakdown of Learning and Teaching Activities:** 2 lecture hours and 1 tutorial hour each week, with 3 coursework assignments.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lecture Hours</td>
<td>20</td>
</tr>
<tr>
<td>Seminar/Tutorial Hours</td>
<td>8</td>
</tr>
<tr>
<td>Supervised practical/Workshop/Studio hours</td>
<td>2</td>
</tr>
<tr>
<td>Summative assessment hours</td>
<td>25</td>
</tr>
<tr>
<td>Feedback/Feedforward hours</td>
<td>2</td>
</tr>
<tr>
<td>Directed Learning and Independent Learning hours</td>
<td>43</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100</strong></td>
</tr>
</tbody>
</table>

**Keywords:** programming languages, interpreters, domain-specific languages, types, abstraction, parameterisation, exceptions, generics.
3 Course materials

3.1 Sample exam question

1. Call-by-value and call-by-name.

   (a) Using a BNF grammar, define the syntax of the untyped lambda-calculus. (2 marks)
   (b) Briefly explain both call-by-value and call-by-name evaluation and list one advantage
       of each approach. (4 marks)
   (c) Write the small-step operational semantics rules for call-by-name and call-by-value
       evaluation in the lambda-calculus. (4 marks)

2. In the C/C++/Java family of languages, the following do...while construct is provided:

   ```
   do {
       stmt
   } while (exp)
   ```

   will evaluate the statement stmt and then test the Boolean value of expression exp; if the
   value is true, execution continues by evaluating the do...while statement again, otherwise
   execution continues.

   (a) Give operational semantics rules for do...while statements (extending the small-step
       semantics for while-programs) (6 marks)
   (b) Show how to rewrite a single do { stmt } while (exp) statement in terms of while
       and if ... then ... else. (5 marks)
   (c) List one advantage and one disadvantage of the do...while construct as opposed to
       the plain while. (4 marks)
### 3.2 Sample coursework specification

(The following is a generic description that will have to be fleshed out with code and implementation details depending on the programming language used for the course).

The two coursework assignments could have the following form:

1. **Coursework 1:** Students are provided with some pre-written code that implements a front-end for a basic interpreter for a small programming language. They are asked to first implement the key components of the interpreter (reinforcing lectures on working with abstract syntax and environments) and then add a nontrivial feature to the language, such as simple exception-handling or a simple kind of object, including augmenting the parser and interpreter stages.

2. **Coursework 2:** Students are given a specification of a small domain-specific language and an outline of how it is to be implemented in the course programming language (building on lectures on DSLs). This includes a specification of “valid” programs and of the behaviour of the DSL programs, either via interpretation or translation to some other language (or both). Examples might include:
   - Source language: a small language for describing geometric scenes; target language: a representation of a scene decomposed into triangles for rendering by a ray tracer.
   - Source language: a small markdown-like language. Target language: HTML (for rendering on the Web) or LaTeX (for translation to PDF).
3.3 Sample tutorial/lab sheet question

1. Parameterized types
Some types, such as lists, are naturally thought of as parameterized. For example, in Scala, the type `List[A]` takes a parameter `A`, the type of elements of the lists. Many programming languages now have this capability, although usually with different syntax:

- **Java**: `List<A>
- **Scala**: `List[A]
- **ML**: `'a list
- **Haskell**: `[a]

(a) Define a type (i.e. a class) in Scala called `Tree[A]` for binary trees parameterized by the type `A` of the data associated with nodes. There should be two constructors for such trees: `Leaf()` constructing a leaf with no data, and `Node(a,t1,t2)` taking an `A`-value and two trees and constructing a tree.

(b) Define two values of type `Tree[A]` for different instantiations of `A`, for example, integers and strings.

2. Parametric polymorphism
Type parameters are useful because they allow us to describe the behaviour of operations on parameterized types. For example, instead of having one length function for lists of integers and another for lists of strings, and so on, we can define a single list length function:

```scala
def length(l : List[A]) : Int = l match {
  case List() => 0
  case x :: xs => length xs + 1
}
```

This is called **parametric polymorphism**. In parametric polymorphism, the function’s behaviour is the same for any type instantiated as the parameter — we cannot give one behaviour for one type and a “different” behaviour for another type.

(a) Write a function that adds up all of the integers in value of type `List[Int]`, returning an `Int`. Generalise its type signature by replacing each occurrence of `Int` with a parameter `A`. What goes wrong?

(b) Add a method `size` to `Tree` that computes the size of a binary tree, where the size of a leaf is 1 and that of a node `Node(a,t1,t2)` is the sum of the sizes of `t1` and `t2` plus one.

3. Ad hoc polymorphism
Some languages allow for a form of overloading of operations, sometimes called **ad hoc polymorphism**. For example, common arithmetic operations such as addition (+) might be overloaded for different numeric types. More generally, in languages like C++, certain built-in operators can be overloaded to provide special behaviour when used with specific classes. As an example, the C++ standard library overloads the operators `<<` and `>>` to perform bit shifting operators when used with integers and to perform stream input and output.

In Scala, such overloading is accommodated in a somewhat more principled way using a feature called **traits**. An operation such as `size` can be defined as part of a trait as follows:

```scala
trait HasSize {
  def size() : Int
}
```
A trait can be included in a class definition as follows:

class Tree[A] extends HasSize {
    ...
}

(a) Define a subclass Tree1 of Tree that extends HasSize, so that the inherited definition of size matches the required size component of HasSize.

(b) Define a second subclass Tree2 of Tree that extends HasSize and redefine its size operation to always return -1.

(c) Write a function sameSize that takes two values of type HasSize and checks whether they have the same size.

(d) Call this function on values of the different subclasses of Tree to verify that the correct implementations of size are called for different types.

4. Parameterized traits

Traits can themselves be parameterized by types. The built-in trait Ordered[T] is an example:

trait Ordered[T] {
    def compare(that : T) : Int
    def < (that: T) : Boolean = ???
    def <= (that: T) : Boolean = ???
    def > (that: T) : Boolean = ???
    def >= (that: T) : Boolean = ???
    ...
}

Here, the type parameter T is needed to name the type of other elements to which this will be compared. The this.compare(that) operation returns a positive integer if this is less than that, zero if they are equal and a negative integer if this is greater than that.

Based on this specification, fill in the ??? regions in the above code snippet with code that defines standard comparison operators such as < in terms of compare. (This definition is already part of the standard library definition of Ordered, hence, to implement the trait, one only needs to define compare and the other standard comparison operators are defined automatically.)

3.4 Any other relevant materials

Illustrative lecture / tutorial plan  As requested by earlier Board of Studies discussion, here is a plan of lectures showing what material could be covered in 18-20 lectures (with 2 lectures being introductory/review and another 2 lectures on special topics, leaving 16 lectures of examinable material). (As noted in the syllabus, other selections of topics are possible.)
<table>
<thead>
<tr>
<th>Week</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>L4</th>
<th>Lab</th>
<th>Warm-up exercises due</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>L5</td>
<td>L6</td>
<td>L7</td>
<td>L8</td>
<td>T1</td>
<td>First coursework due</td>
</tr>
<tr>
<td>3</td>
<td>Interpreters and implementing operational semantics</td>
<td>Function types and recursive functions</td>
<td>Objects, classes, and interfaces</td>
<td>Variant types, pattern matching; lists</td>
<td>Variable binding, substitution, and their pitfalls</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>L9</td>
<td>L10</td>
<td>L11</td>
<td>L12</td>
<td>T2</td>
<td>First coursework due</td>
</tr>
<tr>
<td>5</td>
<td>Parameterized types and polymorphism</td>
<td>Inheritance, subtyping</td>
<td>Domain-specific language examples: parsing, simple graphics</td>
<td>Domain-specific language implementation and translation</td>
<td>Lists and general recursive types</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>L13</td>
<td>L14</td>
<td>T3</td>
<td>T4</td>
<td>Second coursework due</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>State; assignment and imperative control; other side-effects</td>
<td>Control structures: goto, procedure call/return, exceptions</td>
<td>Generics and overloading</td>
<td>Overview of second coursework</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>L15</td>
<td>L16</td>
<td>T5</td>
<td>T6</td>
<td>Second coursework due</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Special topic guest lecture: verification or language-based security</td>
<td>Special topic guest lecture: parallel or distributed programming</td>
<td>Pragmatics of language processing</td>
<td>Operational semantics of stateful computation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>L17</td>
<td>L18</td>
<td>T7</td>
<td>T8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pure vs. stateful computation; call by name vs. call by value</td>
<td>Reactive programming; events and handlers</td>
<td>Stream programming examples</td>
<td>Control abstractions, continuations</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L19</td>
<td>L20</td>
<td>T8</td>
<td></td>
<td>Course review</td>
<td>Fun with continuations; coroutines</td>
</tr>
</tbody>
</table>
4 Course management

[This information can be compiled in parallel to the elicitation of comments for section 5.]

4.1 Course information and publicity

For a quick start in the first year of delivery, the following could be provided at the start of the year:

- A lecture schedule (see section 3.4) listing topics to be covered in each lecture, together with references to web resources or textbooks
- Pointers to web-based resources on the programming language to be used for the course (e.g. Scala, F#)
- Two sample exam questions

During the first delivery of the course itself, the lecture schedule would be turned into slides. (Producing a complete set of materials before the beginning of classes would require essentially the same work as delivery of the course itself; moreover, the planned course material is similar to that covered by similar courses mentioned in the introduction and so existing course slides or lecture notes can be adapted relatively easily.)

4.2 Feedback

Feedback mechanisms will follow University best practice:

Formative feedback: will consist of feedback on an initial (not for credit) formal lab-based programming exercise, as well as other informal interaction with course staff through office hours and the course mailing list/forum

Summative feedback: will consist of feedback on the two assessed courseworks.

Monitoring and responding to student feedback In addition, to monitor the delivery of the course in its first few years, it is planned to ask tutors and lab demonstrators to record and report on student participation and exercise suitability, so that difficulty level and effectiveness of these activities can be fine-tuned. It is also planned to give exit interviews to a sample of students to collect targeted feedback on specific course issues that may not be captured by generic course feedback collected by ITO. Of course, comments from course feedback and SSLC will also be taken into account.

4.3 Management of teaching delivery

The course management will follow the standard practice. After the first year, the course may benefit from being co-taught by two lecturers. The ITO will support allocation of students to tutorials, collection, recording, and return of coursework, and so on. Computing Services may be asked to support the course by ensuring installations of specific programming languages or libraries.
5 Comments

All comments received to date have been incorporated into the proposal.

5.1 Year Organiser Comments
5.2 Degree Programme Coordinators
5.3 BoS Academic Secretary