Functional Programming (COMP0020)

Description

Aims:
This course explores the functional programming paradigm and the implementation technology for functional programming languages. It aims to develop a broad understanding of the functional programming style and recursive programming techniques using the language Miranda, together with an understanding of implementation issues that are relevant not only to functional languages but also to other systems that require automatic dynamic memory management. The course explores the underlying mechanics of strict and lazy functional languages; it does not use Haskell or F# or OOCAML and does not aim to provide training in such languages, though an introduction to Miranda is provided and students are expected to improve their functional programming skills through independent study. In the second half of the course students are expected to use independent study to read extensively about implementation issues which are then discussed in the lectures.

Learning outcomes:
On successful completion of the module, a student will be able to do the following at a level suitable for a Year 3 undergraduate:
1. understand the basics of the lambda calculus and combinators and how they are used in the implementation of functional languages;
2. understand the main features of a lazy functional language;
3. understand type checking, type-inference and the operation of the Miranda (Hindley-Milner) type system;
4. write and understand non-trivial functional programs in Miranda;
5. understand the computation and memory management issues affecting the sequential implementation of lazy functional languages;
6. solve problems relating to all of the above, under examination conditions.

Key information

<table>
<thead>
<tr>
<th>Year</th>
<th>2019/20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Credit value</td>
<td>15 (150 study hours)</td>
</tr>
<tr>
<td>Delivery</td>
<td>UG L6, Campus-based</td>
</tr>
<tr>
<td>Reading List</td>
<td>View on UCL website</td>
</tr>
<tr>
<td>Tutor</td>
<td>Dr Chris Clack</td>
</tr>
<tr>
<td>Term</td>
<td>Term 2</td>
</tr>
<tr>
<td>Timetable</td>
<td>View on UCL website</td>
</tr>
</tbody>
</table>

Assessment

- Written examination (main exam period): 100%

Find out more

For more information about the department, programmes, relevant open days and to browse other modules, visit ucl.ac.uk

Disclaimer: All information correct as of June 2019. Please note that aspects of the module may be subject to change. UCL will make best efforts to inform applicants of major changes.
Content:
Context:
- Classification of Programming Languages;
- Distinctive Features of Functional Programming Languages;
The Lambda Calculus and Combinators:
- Versions of the Lambda Calculus
- Syntax and semantics;
- Reduction orders, strong normalisation;
- Combinators
- Computationally complete sets;
Introduction to Miranda:
- Programming Environment;
- Types;
- Recursion;
- Pattern-Matching;
- Lists;
- Higher-Order Functions;
- User-Defined Types; Types:
Recursive Programming Techniques:
Introduction to Implementation Techniques:
- Strict Evaluation and Lazy Evaluation;
- The Need for Automatic Memory Management;
Automatic Memory Management:
- Memory Allocation and Garbage Collection;
- Garbage Collection Techniques.

Requisites:
In order to be eligible to select this module, a student must be registered on a programme for which it is a formally-approved option or elective choice AND must have EITHER (i) taken Years 1 and 2 of BSc or MEng Computer Science at UCL; OR (ii) taken all Term 1 modules of the MSc Computer Science programme at UCL (and should currently be taking Term 2 modules for the MSc Computer Science programme at UCL); OR (iii) undertaken the following prior study at university undergraduate (Majors) or postgraduate level;
- Programming in one high-level programming language and one assembly language;
- Formal systems of logic such as Boolean algebra, propositional logic or predicate calculus;
- Virtual machines, virtual memory and memory paging;
- Compilers, including lexical analysis, parsing and code generation;
- Dynamic data structures and abstract data types;
- Models of storage in computer systems;
- Algorithmic complexity.
Computer Science

Functional Programming (COMP0020)

**Description**

**Aims:**
This course explores the functional programming paradigm and the implementation technology for functional programming languages. It aims to develop a broad understanding of the functional programming style and recursive programming techniques using the language Miranda, together with an understanding of implementation issues that are relevant not only to functional languages but also to other systems that require automatic dynamic memory management. The course explores the underlying mechanics of strict and lazy functional languages; it does not use Haskell or F# or OOCAML and does not aim to provide training in such languages, though an introduction to Miranda is provided and students are expected to improve their functional programming skills through independent study. In the second half of the course students are expected to use independent study to read extensively about implementation issues which are then discussed in the lectures.

**Learning outcomes:**
On successful completion of the module, a student will be able to do the following at a level suitable for a Year 3 undergraduate:
1. understand the basics of the lambda calculus and combinators and how they are used in the implementation of functional languages;
2. understand the main features of a lazy functional language;
3. understand type checking, type-inference and the operation of the Miranda (Hindley-Milner) type system;
4. write and understand non-trivial functional programs in Miranda;
5. understand the computation and memory management issues affecting the sequential implementation of lazy functional languages;
6. solve problems relating to all of the above, under examination conditions.

**Key information**

- **Year:** 2019/20
- **Credit value:** 15 (150 study hours)
- **Delivery:** PGT L7, Campus-based
- **Reading List:** [View on UCL website](https://www.ucl.ac.uk)
- **Tutor:** Dr Chris Clack
- **Term:** Term 2
- **Timetable:** [View on UCL website](https://www.ucl.ac.uk)

**Assessment**

- Written examination (main exam period): 100%

**Find out more**

For more information about the department, programmes, relevant open days and to browse other modules, visit [ucl.ac.uk](https://www.ucl.ac.uk)
Content:

Context:
- Classification of Programming Languages;
- Distinctive Features of Functional Programming Languages;

The Lambda Calculus and Combinators:
- Versions of the Lambda Calculus
- Syntax and semantics;
- Reduction orders, strong normalisation;
- Combinators
- Computationally complete sets;

Introduction to Miranda:
- Programming Environment;
- Types;
- Recursion;
- Pattern-Matching;
- Lists;
- Higher-Order Functions;
- User-Defined Types; Types:

Recursive Programming Techniques:

Introduction to Implementation Techniques:
- Strict Evaluation and Lazy Evaluation;
- The Need for Automatic Memory Management;

Automatic Memory Management:
- Memory Allocation and Garbage Collection;
- Garbage Collection Techniques.

Requisites:

In order to be eligible to select this module, a student must be registered on a programme for which it is a formally-approved option or elective choice AND must have EITHER (i) taken Years 1 and 2 of BSc or MEng Computer Science at UCL; OR (ii) taken all Term 1 modules of the MSc Computer Science programme at UCL (and should currently be taking Term 2 modules for the MSc Computer Science programme at UCL); OR (iii) undertaken the following prior study at university undergraduate (Majors) or postgraduate level;
- Programming in one high-level programming language and one assembly language;
- Formal systems of logic such as Boolean algebra, propositional logic or predicate calculus;
- Virtual machines, virtual memory and memory paging;
- Compilers, including lexical analysis, parsing and code generation;
- Dynamic data structures and abstract data types;
- Models of storage in computer systems;
- Algorithmic complexity.