COMP90053 Program Analysis and Transformation

Credit Points:	12.5
Level:	9 (Graduate/Postgraduate)
Dates & Locations:	This subject is not offered in 2015.
Time Commitment:	Contact Hours: 36 hours, comprising of two 1-hour lectures and one 1-hour workshop per week Total Time Commitment: 200 hours
Prerequisites:	**One of the following:** **Subject** / **Study Period Commencement:** / **Credit Points:** COMP30020 Declarative Programming — Semester 2 — 12.50 COMP90048 Declarative Programming — Semester 2 — 12.50
Corequisites:	None
Recommended Background Knowledge:	Basic proficiency in discrete mathematics.
Non Allowed Subjects:	None
Core Participation Requirements:	For the purposes of considering request for Reasonable Adjustments under the Disability Standards for Education (Cwth 2005), and Student Support and Engagement Policy, academic requirements for this subject are articulated in the Subject Overview, Learning Outcomes, Assessment and Generic Skills sections of this entry. It is University policy to take all reasonable steps to minimise the impact of disability upon academic study, and reasonable adjustments will be made to enhance a student's participation in the University's programs. Students who feel their disability may impact on meeting the requirements of this subject are encouraged to discuss this matter with a Faculty Student Adviser and Student Equity and Disability Support: http://services.unimelb.edu.au/disability
Contact:	email: **harald@unimelb.edu.au (mailto:harald@unimelb.edu.au)**
Subject Overview:	**AIMS** In the 1930s, Alan Turing and Konrad Zuse independently proposed designs of computing machines based on the idea that storage used for data and storage used for instructions be indistinguishable. This “stored-program” model formed the blueprint for all modern computers. The ability to treat programs as data turned out to be very powerful, as it meant that a program could be designed to read, generate, analyse and/or transform other programs, and even modify itself while running. This subject is concerned with meta-programs - programs that work on other programs, possibly generating programs as output. People routinely read, generate, analyse, test, and transform programs. For example, a programmer may look through code for potential buffer overruns, and may add runtime tests to avoid the security problems that could result. It is preferable, however, to automate such activity as far as we can, partly because that makes programmers more productive, and partly because computers generally are better at these tasks, avoiding human oversights and mistakes. This subject introduces the main techniques and applications of program analysis and transformation, including methods used by modern optimizing compilers and allied tools. **INDICATIVE CONTENT** # Syntax and semantics: Program representations, operational and denotational semantics. # Fixed point theory: Order, lattices, functions and fixed points # Program analysis: The monotone framework, constraint-based analysis, collecting semantics, abstract interpretation, widening, inter-procedural analysis, analysis of functional and logic programs

	# Meta-programming: Interpreters, meta-interpreters, program instrumentation, source-to-source program transformation, including fold/unfold and partial evaluation # Other topics may be covered via the project, for example, analysis for violations of safety and/or security policies, or analysis and transformation for finding and implementing parallelism.
Learning Outcomes:	**INTENDED LEARNING OUTCOMES (ILO)** On completion of this subject, students should have the following skills: 1 Describe standard approaches to program analysis and program transformation 2 Solve mathematical problems relevant to reasoning about program runtime properties 3 Design and build non-trivial program analysis/transformation tools 4 Adapt and apply existing program analysis tools to the needs of a project 5 Explain the limits of program analysis as applied to specific languages, and use this to inform decisions about which languages to use in programming projects
Assessment:	A 45-minute mid-semester written test around Week 7, requiring approximately 13 - 15 hours of work (10%), addressing ILOs 1 and 2. A programming project during semester (30%), requiring approximately 35 - 40 hours of work, addressing ILOs 3 and 4. A 2-hour end-of-semester written examination (60%), addressing ILOs 1, 2, and 5. Hurdle requirement: To pass the subject, students must obtain: 15/30 in project work 35/70 in the mid-semester test and end-of-semester written examination combined.
Prescribed Texts:	None
Breadth Options:	This subject is not available as a breadth subject.
Fees Information:	Subject EFTSL, Level, Discipline & Census Date, http://enrolment.unimelb.edu.au/fees
Generic Skills:	On completion of the subject, students should have the following skills: # Analytical skills # Reasoning and problem-solving skills # Ability to apply knowledge of science and engineering fundamentals # Capacity for creativity and innovation # Ability to undertake problem identification, formulation and solution # Ability to utilise a systems approach to complex problems and to design for performance.
Notes:	**LEARNING AND TEACHING METHODS** The subject involves two 1-hour lectures per week followed by a 1-hour workshop. Weekly tutorial problems are assigned and discussed in class. The programming project is pivotal, as the subject relies heavily on learning-by-doing. Students work in small groups to implement a sophisticated program analysis tool of practical value. Students are encouraged to share ideas and artefacts, such as developed test data. More generally, “contributing student pedagogy” is utilised as a community-building tool. Accordingly, in-plenum discussion of the project is allowed, and encouraged. **INDICATIVE KEY LEARNING RESOURCES** The subject uses online reading materials and an online discussion forum. It offers advance access to teaching materials, including slides used in lectures. **CAREERS / INDUSTRY LINKS** Semantics-based formal approaches are increasingly important in software engineering and software development, in particular for security- and safety-critical applications.
Related Course(s):	Master of Information Technology Master of Information Technology Master of Science (Computer Science) Master of Software Systems Engineering

Related Majors/Minors/ Specialisations:	B-ENG Software Engineering stream MIT Computing Specialisation MIT Distributed Computing Specialisation Master of Engineering (Software)