

# Course Outline & Introduction

## COMP3610 — Principles of Programming Languages

Ben Lippmeier

Australian National University

Semester 2, 2009

## About 27 Lectures:

11 – 12am	Monday	Chem T2
11 – 12am	Tuesday	Phys G5
9 – 10am	Wednesday	Phys G5

Not all the lecture slots will be used. Check the lecture page on the course website for details. No lectures in Weeks 7-8.

Most lectures will include short comprehension exercises at the end. You should do these as soon after the lecture as practical (or think about them on the way home) to test what parts you did/didn't understand. Most should only take 10-15 min.

## About 8 lab/tutorial classes:

- Commencing *Week 2*
- Tuesday 1pm in N116 (weeks 2, 4, 5, 6) (2 hours)
- Tuesday 1pm in N108 (weeks 9, 10, 11, 12) (1-2 hours)

These will be longer exercises and practical programming tasks. They may take longer than the lab session to complete, and you are expected to finish them in your own time if need be.

It is important, and expected, that you will have made a genuine attempt at the exercises *before* the laboratory class or tutorial.

## Assessment

- There will be two assignments with a combined weight of 30%. Each will have a fixed deadline and they will be due in about weeks 9 and 12.
- There will be a 3 hour final exam with a weight of 70%. It will be open-book.
- Your final mark will be the sum of these two components.
- Consistent scaling may occur across all students.
- Final marks are also moderated by the Department of Computer Science examiners meeting.

## What is this subject about?

“Principles of programming languages...”

- Features and foundations of a *functional programming language*.
- The *recognition* and *translation* of programming languages.
- The *semantics* of programming languages.

Programs and programming languages as objects of study in their own right.

### Unifying theme:

“The unreasonable effectiveness of logic in computer science” – Phil Wadler.

## Prerequisites

- That you can already write programs. This course does not teach basic programming skills.
- That you can write functions over recursive data structures. If you have fond memories of inserting nodes into binary search trees, then that's what we want.

## Highly recommended:

- At least a passing familiarity with a functional programming language.  
Any of: Haskell, ML, Scheme, O’Caml, Scala ...
- If you haven’t used a functional language before, then *start today*.  
The COMP1100 course notes would be a good place to start.  
<http://cs.anu.edu.au/student/comp1100.2009>
- Familiarity with elementary logic and natural deduction.  
There is a list of logic textbooks at:  
<http://cs.anu.edu.au/student/comp2600.2008/reference.php>  
Also check out the COMP2600 lecture notes:  
<http://cs.anu.edu.au/student/comp2600.2009>

# Proposed Syllabus

- Introduction.
- What a programming language is.
- Types and functional programming languages.
  - Haskell by example.
  - The typed lambda calculus.
  - Reduction strategies.
  - System-F and the lambda-cube.
  - Program optimisation.
  - Polymorphic type inference.

- Recognition and translation of languages.
  - Lexical analysis.
  - Top-down parsers.
  - Bottom-up parsers.
  - Scanner and parser generators.
- Semantics of programming languages.
  - Operational semantics.
  - Denotational semantics.
  - Soundness of type systems.
- Bringing it together.
  - Strictness analysis.
  - Effect typing and the DDC.

## Useful Books

*Haskell – the Craft of Functional Programming (2e)*. Simon Thompson, 1999.

**Lambda calculus, types, soundness, System-F.**

*Types and Programming Languages (TAPL)*. Pierce, 2002.

(Works up from simple to advanced. Highly recommended.)

**Lambda calculus, simple types, semantics, fixpoint theory.**

*Semantics with Applications*, Nielson & Nielson, 1992.

(Probably the most basic.)

*The Formal Semantics of Programming Languages*, Winskell, 1993.

(Better, but mathematically more sophisticated.)

*Semantics of Programming Languages*, Gunter, 1992.

(Excellent but advanced.)

## Syntax, Parsing and Translation:

*Compilers: Principles, Techniques, and Tools*. Aho, Sethi and Ullman. 1986.  
(“The dragon book”. An absolute classic. There is a new 2007 edition.)

Perhaps also something like:

*lex & yacc*, Levine, Mason and Brown, 992.

## What to get?

TAPL is considered the standard reference in its field. If you like functional programming then it will keep on giving for years to come.

I read “The Dragon Book” as an undergrad and assimilated it into my world view. (or was it the other way around?)

## Some inspired gibberish

*“I am the eggman, you are the egg-men, I am the walrus, goo goo ga’joo.”*

– John Lennon.

- Written in English (sort of)
- An artistic/musical statement, written while the author was hallucinating.
- What does it mean?
  - Unlikely to be a literal statement.
  - Someone cannot be both the eggman, and a walrus.
  - Walruses (walri?) can’t play piano.
- How can we know if we understand this statement?



## A philosophical / religious statement

*“With our thoughts we make the world.”* – Gautama Buddha.

- Deeply meaningful to > 200 million people worldwide (though not all in the English version).
- Meaning is intuitive. We could discuss the interpretation of this statement, but I’m not sure we could ever fully agree.
- How can we know if we understand it?
  - Philosophy is supposed to make us feel better about ourselves..
  - Is it working yet?

## A scientific statement

*“The sky is blue.”*

- The sky is not blue at night time, it's mostly black.
- Sometimes in the afternoon it's red or orange.
- When it's snowing it's white.
- Are you looking at it now? Where are you?
- Seemingly clear statements of fact often rely on unstated assumptions.

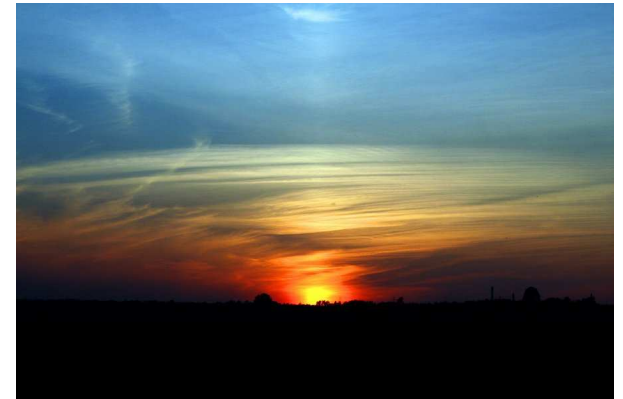


photo: Adam Ziaja CCSA 2.5

## Two logical statements

$$\neg(A \wedge B) \Rightarrow \neg A \vee \neg B$$

$$\text{I am a Giraffe} \wedge \text{You are Texas} \Rightarrow \text{I am a Giraffe}$$

- In a logical statement, all assumptions are explicit.
- The second statement is no less logical than the first.
- Verification of a logical statement requires no intuition, it can be done mechanically by a theorem prover.

## Gibberish again.

Oh HAI, I made you a cookie, but I ate it.

⇒ I made you a cookie.

Oh NOES, I made you a cookie, but I ate it.

⇒ I ate it.



## What if we treat these statements as programs?

**if** *True* **then** *print* “cookie” **else** *print* “eated”  
⇒ *print* “cookie”

**if** *False* **then** *print* “cookie” **else** *print* “eated”  
⇒ *print* “eated”

- When we substitute symbols for familiar ones, a statement can appear to gain meaning, yet we have no more information than before.

## A meaningless dance of symbols.

69 54 74 50 63 61 6b 65 45 65 61 74 65 64  
⇒ 65 61 74 65 64

69 46 74 50 63 61 6b 65 45 65 61 74 65 64  
⇒ 63 61 6b 65

- To the **computer**, our programs are a meaningless dance of symbols.
- It is not necessary for it to “understand” the symbols, just to **compute** with them.

## Formal semantics gives precise meaning to gibberish.

- It is not necessary for *us* to “understand” the symbols either.
- To perform a computation the only things that matter are:
  - What symbols we have.
  - What symbols we can replace them with.
- Computation is purely mechanical. This course covers how to specify, and to reason about the mechanism.
- As humans, when we read strings of symbols we tend to interpret them, always trying to impose our own meaning. However, when computing by hand we must *avoid this at all cost*.
- This helps us separate the mechanism from the symbols it is working on.

## A program with no meaning

```
#include <stdio.h>
int thing = 0;
int main(int argc, char** argv) {
    printf("%d\n", thing++ + ++thing);
    printf("%d\n", thing);
    return 0;
}
```

- Compile and run this with optimisations turned on, and then with optimisations turned off (with and without the -O flag).
- Compare the results. “Tested” with GCC 4.0.1 and 4.1.2. No, it’s not a bug.
- Compile again with `-Wall` for a laugh.

## Exercises

1. What do you think the program on the previous slide should do?
2. Why do you think it gives different results when compiled with optimisations on and off?
3. Modify the program so that it returns the “correct” answer each time. You should only need to cut and paste.