

# Programming Languages: Functional Programming

## 0. Introduction

Shin-Cheng Mu

Autumn 2025

### So, what is this course about?

- Some people refer to this course as “programming language theory”, that is, theories about the language and tools we use to program.
- Why does it matter to us?

## 1 The Isle of Knights and Knaves

- On a remote isle there live two kinds of people:
  - the *knights* always tell the truth, while
  - the *knaves* always lie.
  - Everyone on the isle is either a knight or a knave.
- You are at the entrance of a cave. Legend has it that the deep in the cave there buries a huge amount of gold... or a dragon that may swallow you alive. You see an old man. How do you form a question to know which is the case?

### Warming Up

- With two islanders, A and B:
- A says: “if you ask B whether he is a knight, he would say ‘Yes’.”
- What can you infer about A and B?

### Exhaustive Enumeration?

- What matters more is how you solved the problem.
- Most people would exhaustively enumerate all possibilities.
  - “Suppose that A is a knight...”

### Equivalence

- Abbreviate “A is a knight” to  $A$ .
- As a convention (among certain circles), we write logical equivalence, that is, “if and only if”, or equality on booleans, as  $\equiv$ .
- Suppose that A said some sentence P. If A is a knight, P must be *True*. Otherwise P must be *False*.
- Thus, “A said P” can be denoted by  $A \equiv P$ .

### Warming Up...

- $A \equiv A$  is always *True*.
  - Indeed, any person would say he/she is a knight.
- “A says: ‘B is a knight’.”
  - $A \equiv B$ .
  - A and B are of the same kind.
- A says: “if you ask B whether he is a knight, he would say ‘Yes’.”

$$\begin{aligned} A &\equiv (B \equiv B) \\ &\equiv A \equiv \text{True} \\ &\equiv A. \end{aligned}$$

- Thus we know that A is a knight. Nothing can be said about B.

- A says: “B and I are of the same kind!”

$$\begin{aligned} A &\equiv (A \equiv B) \\ &\equiv \{ \equiv \text{ is associative} \} \\ (A \equiv A) &\equiv B \\ &\equiv \text{True} \equiv B \\ &\equiv B. \end{aligned}$$

- Thus we know that B is a knight. Nothing can be said about A.
- In fact, not many people know that  $\equiv$  is associative.

## Back to the Cave...

- Goal: design a question  $Q$  such that  $A$  answers Yes iff. there is gold in the cave.
- “ $A$  answers Yes to question  $Q$ ” is also written  $A \equiv Q$ .
- Let  $G$  denote “there is gold in the cave.”
- “ $A$  answers Yes to  $Q$  iff. there is gold in the cave.”

$$\begin{aligned}(A \equiv Q) &\equiv G \\ \equiv (Q \equiv A) &\equiv G \\ \equiv Q &\equiv (A \equiv G).\end{aligned}$$

- So the question is “Is ‘You are a knight’ equivalent to ‘there is gold in the cave’?”

## 2 Abstraction

### How Was the Problem Solved?

1. Turn the problem into mathematical formulae.
  2. And then calculate, using the rules associated with the operators.
- The first step, called “abstraction”, is harder.
  - The second step is much easier, because we *let the symbols do the work!*
    - Well-designed symbols relieve us of the mental burden.
    - Recall how you calculate, say  $17 \times 24$ ?
  - Why does that concern us?

### A Programming Language is a Symbolic, Formal System

- Because *a programming language is an abstract model, and a collections of symbols and their related rules, to relieve us of the mental burden of programming.*
- Abstraction: a programming language models the real world, while throws away some “unimportant parts”.
- A formal system: a collection of symbols, and some rules to manipulate them.
  - We hope that a programming language is well-designed, such that it helps us to program.

## Abstraction

- “What are the three most important factors in real estate?”
  - Location, location, and location.
- “What are the three most important factors in a programming language?”
  - Abstraction, abstraction, and abstraction — Paul Hudak.
- Abstraction: the process of
  - extracting the underlying essence of a mathematical concept,
  - removing any dependence on real world objects with which it might originally have been connected, and
  - generalizing it so that it has wider applications or matching among other abstract descriptions of equivalent phenomena.

## Algebra

- “Mary had twice as many apples as John had. Mary found that half of her apples are rotten and thus throws them away. John ate one of his apples. Still, Mary has twice as many apples as John has. How many apples did they originally have?”

$$\begin{aligned}m &= 2 \cdot j, \\ m/2 &= 2 \cdot (j - 1).\end{aligned}$$

## Abstraction

- From “Mary had twice as many apples as John...” to “ $m = 2 \cdot j$ ”:
  - extracted: values, and their relationships.
  - dropped: time, causality, ...
- What if time and causality turn out to be important? We need another abstraction.
  - Perhaps a stronger logic/algebra.

## Not One, but Many Logics

- Propositional logic.
- (First-order) predicate logic: for all, exists...
- Modal logic: describing time and order.
- Separation logic: sharing of resources.
- Descriptive logic: concepts, and relationship between concepts.
- Each (or, some) logic corresponds to a type system in a programming language.

## Abstraction in Imperative Programming Languages

- Abstraction of control structures: for-loops, while-loops...
- Procedure abstraction.
- Data abstraction: user-defined datatypes, instead of bits and bytes...
- What algebraic laws do they satisfy? Hmm... not many, unfortunately.

## Abstractions of Other Paradigms

- Object-oriented programming: everything is an object!
- Functional programming: everything is a function!
- Logic programming: “computation = controlled deduction!” “algorithm = logic + control!”

## A Language is an Abstraction

- A programming language is an abstract view toward computation, with attention on aspects the designers care about.
- To learn a language is to learn its view.
- Alan Perlis: “A language that doesn’t affect the way you think about programming, is not worth knowing.”
- In this term I hope you will see something that affects the way you think about programming.


## 3 Algebraic Manipulation

- What qualifies as a good abstraction?
- Our point of view: one that gives us more properties to manipulate with.

### Greek Alphabetical Numerals

- See Figure 1 for Greek alphabetical numerals.
- 11, 12, 13 are written ια, ιβ, ιγ.
- 21, 22, 23 are written κα, κβ, κγ.
- Natural in a way. Not very suitable for calculation.
- Why can we not denote 23 by βγ? What about 203 and 2003?

### Maya Numerals

- See Figure 2.
- Orders vertically stack, bottom to top. 20-base, apart from the second order, since  $18 \times 20 = 360$  is closer to the number of days in a year.
- Relatively easy arithmetic calculation.
- Zero is represented by .

### Algebraic Properties of Programs?

The following two programs are equivalent.

- ```
s = 0; m = 0;
for (i=0; i<=N; i++) s = a[i] + s;
for (i=0; i<=N; i++) m = a[i] + m;
```
- ```
s = 0; m = 0;
for (i=0; i<=N; i++) {
    s = a[i] + s;
    m = a[i] + m;
}
```

Is that easily seen? Can we transform one to another? Does the equivalence still hold if we replace the assignment by other statements?

α	β	γ	δ	ε	Ϝ	ζ	η	θ	ι	κ	λ	μ	ν	ξ	ο	π	ρ
1	2	3	4	5	6	7	8	9	10	20	30	40	50	60	70	80	90
	ρ	σ	τ	υ	φ	χ	ψ	ω	ϗ	Ϡ	ϡ	Ϣ	ϣ	Ϥ	ϥ	Ϧ	ϧ
	100	200	300	400	500	600	700	800	900	1000	2000	3000					

Figure 1: Greek alphabetical numerals.

·	1 × 18 × 20 × 20 × 20 × 20 = 2880000	⋮				
··	2 × 18 × 20 × 20 × 20 = 288000	⋮	⋮	⋮	⋮	⋮
⋮	6 × 18 × 20 × 20 = 43200	⋮	⋮	⋮	⋮	⋮
··	2 × 18 × 20 = 720	⋮	⋮	⋮	⋮	⋮
⋮	13 × 20 = 260	⋮	⋮	⋮	⋮	⋮
⋮	19	⋮	⋮	⋮	⋮	⋮
	total: 3212199					
	(a) Representating 3212199.					

Figure 2: The Maya arithmetic.

## Maximum Segment Sum

- The specification:  $\max \{ \text{sum}(i, j) :: 0 \leq i \leq j \leq N \}$ , where  $\text{sum}(i, j) = a[i] + a[i + 1] + \dots + a[j - 1]$ .

- What we want the program to do.
- One can imagine a program using three nested loops.

- The program:

```
s = 0; m = 0;
for (i=0; i<=N; i++) {
  s = max(0, a[j]+s);
  m = max(m, s);
}
```

- How to do it.

- They do not look like each other at all!
- Moral: programs that appear “simple” might not be that simple after all!

“...the designer of the program had better regard the program as a sophisticated formula. And we also know that there is only one trustworthy way of designing a sophisticated formula, viz., derivation by means of symbol manipulation. We have to let the symbols do the work.”

— E.W.Dijkstra, The next forty years. 14 June 1989.

## Programming, and Programming Languages

- Correctness: that the behaviour of a program is allowed by the specification.
- Semantics: defining “behaviours” of a program.
- Programming: to code up a correct program!
- Thus the job of a programming language is to help the programmer to program,
  - either by making it easy to check that whether a program is correct,
  - or by ensuring that programmers may only construct correct programs, that is, disallowing the very construction of incorrect programs!

## 4 Plans for this Term

### Plans for this Term

- We will start with learning a functional language, Haskell.
  - We can learn something new since it is so different from what you are used to.
- Much emphasis will be on
  - How to construct programs in a disciplined manner.

- How to show that programs are correct.
- Haskell will be used as a tool to learn semantics.
- *When the time comes*, we will use a dependently typed language, Agda, to talk about relationship between programs and proofs.

### **Textbook and Homepage**

- Unfortunately, there is not a completely suitable textbook.
  - For the functional programming part, I am currently working on a draft textbook! It is available on the course website. Comments welcomed.
- There are a number of good Haskell tutorials.
- Course homepage on NTU COOL <https://cool.ntu.edu.tw/courses/51303>. More info will be updated there.

I wish you enjoy this course.