

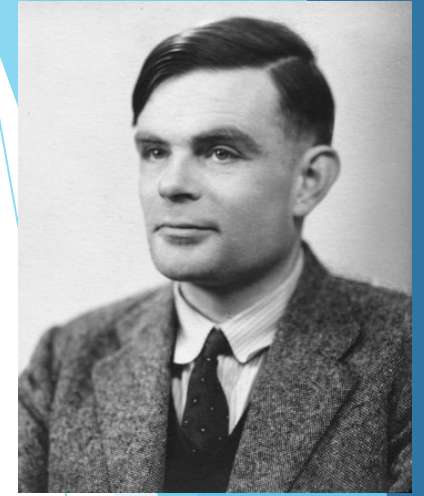
CAB402: Programming Paradigms

Administration

- ▶ **Unit Coordinator:** Dr Wayne Kelly (3138 9336, w.kelly@qut.edu.au)
- ▶ **Prerequisites:** CAB201 Programming Principles and CAB203 Discrete Structures
- ▶ **Contact:** 2 hour lecture + 1 hour practical session per week
- ▶ **Assessment:**
 - ▶ Functional vs Imperative Programming Assignment: 30% (individual), due 24th April
 - ▶ Research Report: 30% (individual), due 31st May
 - ▶ Final Theory Exam: 40%

Theoretical Background

- ▶ Theoretical Models of Computation
 - ▶ Turing Machines
 - ▶ Lambda Calculus
- ▶ Real Computing Machines
 - ▶ Von Neumann Architecture



Alan Turing 1912-54



John von Neumann 1903-57



Alonzo Church 1903-95

Turing Machines

Alan Turing

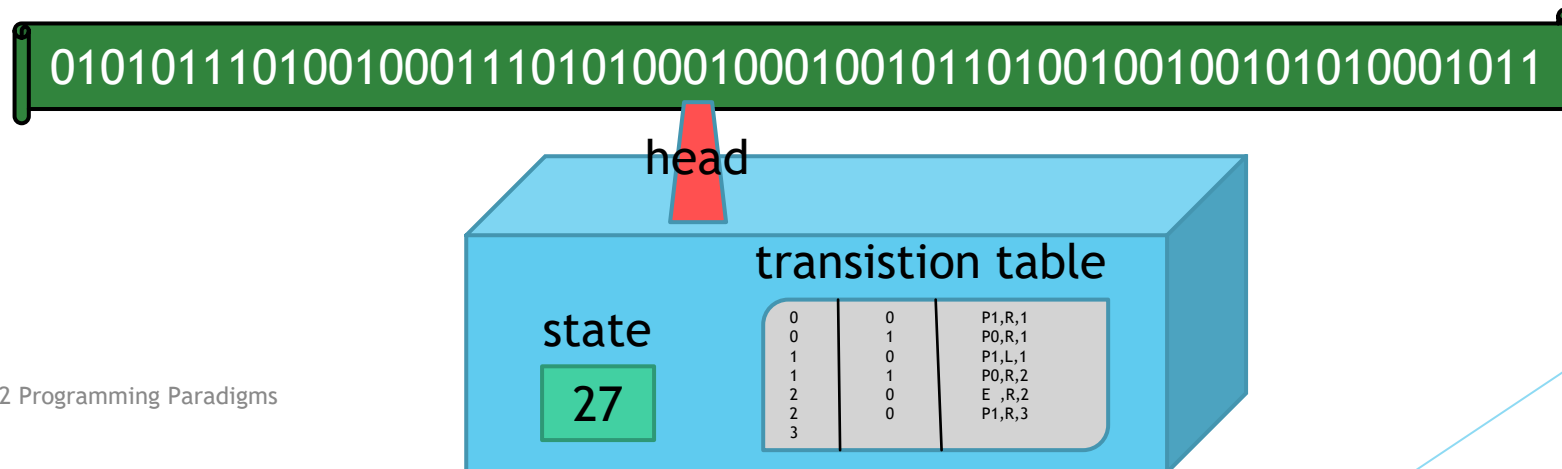


Alan Turing 1912-54



Turing Machines

- ▶ Tape: a sequence of cells, each containing a symbol from some finite alphabet
- ▶ Head: can read and write symbols on the tape and move left or right
- ▶ State Register: used to remember the current state (finite number of states)
- ▶ Transition Table
 - ▶ Current symbol + Current state -> Erase or write a symbol in the current cell +
 - ▶ Move the head left or right +
 - ▶ Change the current state



Example Turing Machine

Current State	Input Symbol	Write Symbol	Move Tape	New State
State 1	0		Right	
State 1	1	0	Right	State 2
State 2	0		Right	State 3
State 2	1		Right	
State 3	0	1	Left	State 4
State 3	1		Right	
State 4	0		Left	State 5
State 4	1		Left	
State 5	0	1	Right	State 6
State 5	1		Left	
State 6	0			Halt
State 6	1	0	Right	State 2

Universal Turing Machine

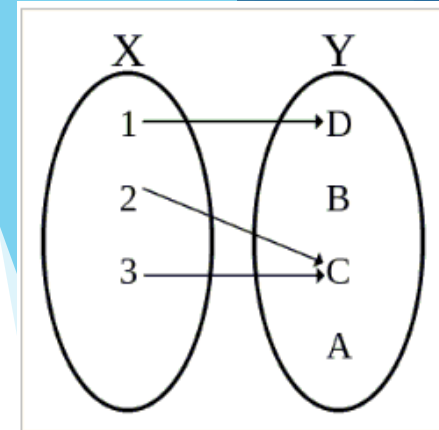
- ▶ Can Simulate any Turing Machine
- ▶ Input Tape contains the “Program” (encoding of the transition table) + Input Data
- ▶ Transition table is finite, so any program can be encoded as a single integer number.
- ▶ Developed by Alan Turing 1936-37.
- ▶ Origin of the “Stored Program Computer” used by John von Neumann (1946)

Mathematical Functions

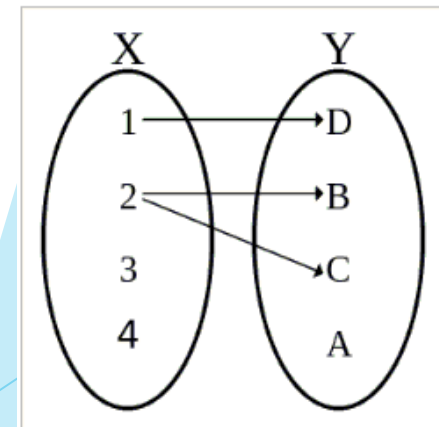
- ▶ [http://en.wikipedia.org/wiki/Function_\(mathematics\)](http://en.wikipedia.org/wiki/Function_(mathematics))
- ▶ In mathematics, a **function** is a relation between
 - ▶ a set of inputs (*it's domain*) and
 - ▶ a set of permissible outputs (it's co-domain)
 - ▶ with the property that each input is related to exactly one output.
- ▶ The concept of function can be extended by allowing more than one input parameter (and/or output values)
 - ▶ This intuitive concept is formalized by a function whose domain (or co-domain) is the Cartesian product of two or more sets.
 - ▶ E.g., consider the function that associates two integers to their product:

$$\text{mult}(x, y) = x \cdot y$$

- ▶ $\text{Sin}(x)$, $\text{Abs}(y)$



The above diagram represents a function with domain $\{1, 2, 3\}$, codomain $\{A, B, C, D\}$ and set of ordered pairs $\{(1,D), (2,C), (3,C)\}$. The image is $\{C,D\}$.



However, this second diagram does *not* represent a function. One reason is that 2 is the first element in more than one ordered pair. In particular, $(2, B)$ and $(2, C)$ are both elements of the set of ordered pairs. Another reason, sufficient by itself, is that 3 is not the first element (input) for any ordered pair. A third reason, likewise, is that 4 is not the first element of any ordered pair.

Partial Functions

- ▶ A function is a partial function if its value is undefined for some elements in its domain.
 - ▶ E.g. $f(x) = 1/x$ is undefined for $x = 0$
- ▶ In the context of computer programs, a computer program that computes a partial function will fail to terminate (i.e. go into a finite loop) for some inputs in its domain.
 - ▶ Such non-termination in this context is not a bug - it is the required behaviour.
 - ▶ A programming language is not Turing complete if it can't generate programs that will sometimes not terminate, as such a language cannot be used to implement partial functions (which a Turing machine can do).

Lambda Calculus



Alonzo Church 1903-95

Lambda Calculus

- ▶ Lambda Calculus expressions are defined recursively as follows:

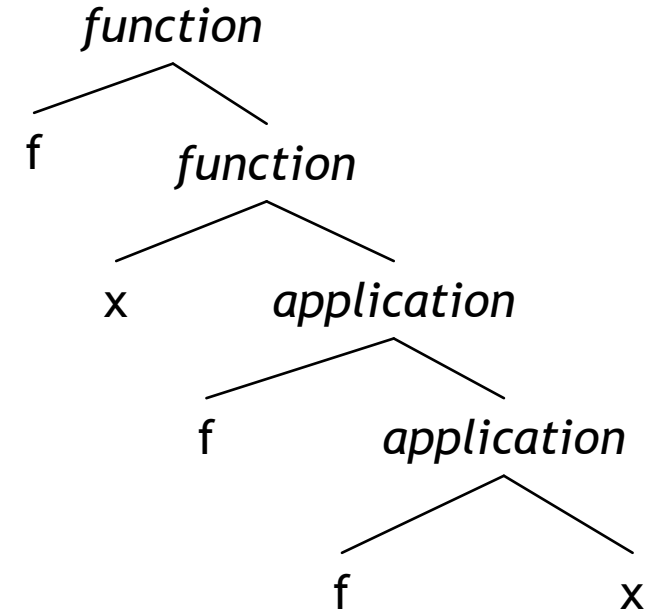
$\langle \text{expression} \rangle := \langle \text{name} \rangle \mid \langle \text{function} \rangle \mid \langle \text{application} \rangle$

$\langle \text{function} \rangle := \lambda \langle \text{name} \rangle . \langle \text{expression} \rangle$

$\langle \text{application} \rangle := \langle \text{expression} \rangle \langle \text{expression} \rangle$

E.g. $\lambda f . \lambda x . (f (f x))$

$(\lambda f . (\lambda x . f x \text{ a}) (\lambda g . (\lambda x . \lambda g . f g x) (f g x)))$

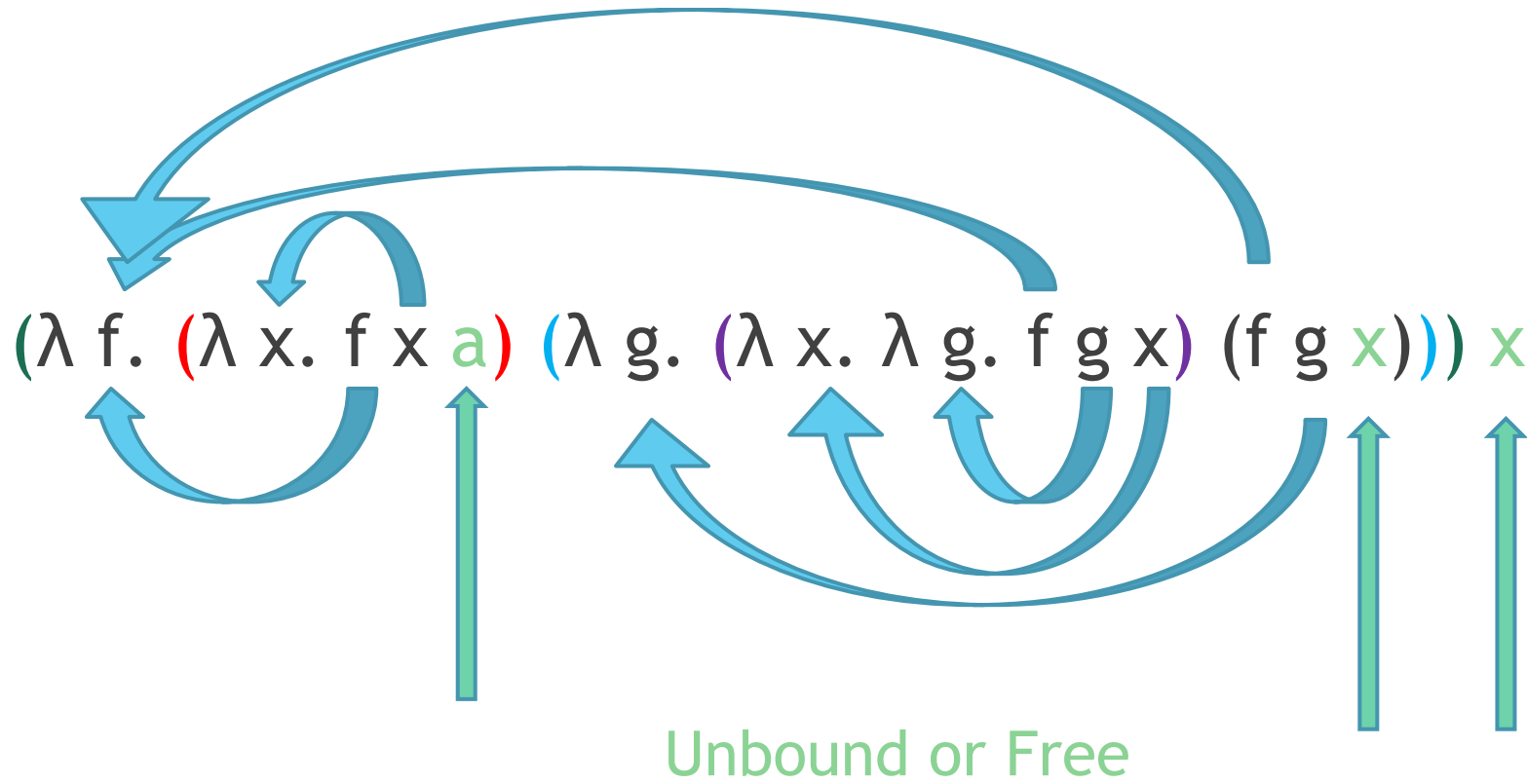


Nested Lexical Scopes in C, C# and Java

```
class Foo
{
    int i;

    void Bar(int j)
    {
        int k = j + 3;
        while (k < 33)
        {
            int p;
            p = k + j + i;
        }
    }
}
```

Bound vs Free names



Evaluating Function Calls in C, C# and Java

```
int foo(int j)
```

```
{
```

```
    return j + 3;
```

```
}
```

```
void main(string args[])
```

```
{
```

```
    int i = 21;
```

```
    int x = foo(i*2);
```

```
}
```

Step 2: substitute formal parameter `j` by argument value 42

Step 1: Evaluate argument `i*2` = 42

Variable renaming in C, C# and Java

```
int foo(int j)
{
    return j + 3;
}
```



```
int foo(int k)
{
    return k + 3;
}
```


Evaluating Lambda Expressions

- ▶ α -reduction (rename parameters):
 - ▶ $\lambda x. E \Rightarrow \lambda y. E[x/y]$ (provided that y is not already used in E)
 - ▶ E.g. $\lambda x. \lambda y. x y \Rightarrow \lambda z. \lambda y. z y$
- ▶ β -reduction (substitution rule):
 - ▶ $(\lambda x. E) F \Rightarrow E[x/F]$ (provided E does not use parameter names used in F)
 - ▶ E.g. $(\lambda x. (\lambda y. (x y))) (\lambda z. z) \Rightarrow \lambda y. ((\lambda z. z) y) \Rightarrow \lambda y. y$
- ▶ where $E[a/b]$ denotes the result of replacing all free occurrences of a in E by b .

Precedence in Lambda Expressions

- ▶ Lambda abstraction has lower precedence than an application, so:
 - ▶ $\lambda x. y z \iff \lambda x. (y z)$
- ▶ Applications are left associative, so:
 - ▶ $x y z \iff (x y) z$

Reduction Examples

$(\lambda x. \lambda y. (x y)) y$

$\Rightarrow (\lambda x. \lambda z. (x z)) y$

(α -reduction renaming y to z)

$\Rightarrow \lambda z. (y z)$

(β -reduction replacing x by y)

$(\lambda f. \lambda x. f (f x)) (\lambda x. x)$

$\Rightarrow \lambda x. (\lambda x. x) ((\lambda x. x) x)$

(β -reduction replacing f by $(\lambda x. x)$)

$\Rightarrow \lambda x. (\lambda x. x) ((\lambda y. y) x)$

(α -reduction renaming x to y)

$\Rightarrow \lambda x. (\lambda x. x) x$

(β -reduction replacing y by x)

$\Rightarrow \lambda x. (\lambda y. y) x$

(α -reduction renaming x to y)



$\Rightarrow \lambda x. x$

(β -reduction replacing y by x)

Higher Order Functions

- ▶ Functions where the input or the output is itself a function
- ▶ e.g. Input is a function:
 - ▶ $\lambda f. \lambda x. f(f\ x)$
- ▶ e.g. Output is a function:
 - ▶ $\lambda x. (\lambda y. x + y)$

Currying

- ▶ All functions in lambda calculus take only one argument.
 - ▶ $\lambda x. E$
- ▶ Functions that require two (or more) arguments are simulated by *currying*.
 - ▶ $\lambda (x,y). E$ 
 - ▶ $\lambda x. (\lambda y. E)$ 
 - ▶ A function of two arguments is simulated by a function of one argument that returns a function of one argument that returns the result.
 - ▶ E.g:
 - ▶ $\text{plus} = \lambda x. (\lambda y. x + y)$
 - ▶ This allows partial application:
 - ▶ E.g. $\text{plus } 4$ is a function which adds 4 to its argument.

Is Lambda Calculus Rich Enough to Program Arithmetic?

- ▶ Lambda calculus does not have built in support for numeric literals or numeric operations!
- ▶ On most computers, numbers are encoded as a sequence of 0's and 1's
- ▶ In lambda calculus we can encode or represent numbers as functions:

$0 = \lambda f. \lambda x. x$

$1 = \lambda f. \lambda x. f(x)$

$2 = \lambda f. \lambda x. f(f(x))$

$3 = \lambda f. \lambda x. f(f(f(x)))$

$\text{succ} = \lambda n. \lambda f. \lambda x. f(n f x)$

$\text{plus} = \lambda n. \lambda m. (n \text{ succ } m)$

or without named functions: $(\lambda \text{succ}. \lambda n. \lambda m. (n \text{ succ } m)) (\lambda n. \lambda f. \lambda x. s(n f x))$

$\text{mult} = \lambda n. \lambda m. n (\text{plus } m) 0$

$\text{True} = \lambda a. \lambda b. a$

$\text{False} = \lambda a. \lambda b. b$

$\text{ifthenelse} = \lambda p. \lambda a. \lambda b. p a b$

- ▶ <http://www.cs.unc.edu/~stotts/723/Lambda/overview.html>

Recursion in Lambda Calculus

- ▶ With named functions:
 - ▶ $\text{Factorial} = \lambda n . \text{if } n == 0 \text{ then } 1 \text{ else } (\text{mult } n (\text{Factorial } (n-1)))$
- ▶ Without named function:
 - ▶ $\text{Factorial} = Y (\lambda r. \lambda n . \text{if } n == 0 \text{ then } 1 \text{ else } (\text{mult } n (r (n-1))))$
 - ▶ Where Y is a special function called the *fixed point combinator*:
 - ▶ $Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$
 - ▶ This function has the property that: $Y f = f(Y f)$

Church-Turing Thesis

- ▶ **Proved** that Turing Machines and Lambda Calculus are equivalent in computation power, i.e.
 - ▶ If a Turing machine exists to solve a particular problem then an equivalent solution can be expressed using Lambda calculus.
 - ▶ If a function can be expressed using Lambda calculus, then an equivalent Turing machine exists to compute the same function.
- ▶ **Hypothesised** that Turing Machines and Lambda Calculus effectively define the set of functions that can be computed or are “effectively calculable”.
 - ▶ If a Turing computer can’t solve a problem, then:
 - ▶ no computer that we have today* (Dell, MacBook Air, Cray Supercomputer, etc.)
 - ▶ nor any computer that we might invent in the future (even Quantum computers).could solve that problem.

**Technically, today’s real computers are less powerful than a Turing Machine as they do not have infinite memory.*

Computability

- ▶ Need to Distinguish between:
 - ▶ Computability
 - ▶ Computational Complexity
- ▶ For a given problem, there are typically many algorithms (approaches) to solve that problem.
- ▶ If a problem is undecidable, then no algorithm exists that will solve that problem (no matter how hard we search or how clever we are).
- ▶ Known Undecidable Problems (http://en.wikipedia.org/wiki/List_of_undecidable_problems)
 - ▶ Halting problem
 - ▶ Program Equivalence

Turing Complete

- ▶ A Programming Language is said to be “*Turing Complete*” if it can be used to simulate a Turing machine.
- ▶ If a Language is not Turing Complete then there are some problems that it cannot be used to solve, that are solvable using other languages.

Why Does Choice of Language Matter?

- ▶ Special purpose languages such as SQL, HTML, XML are not Turing Complete, but the vast majority of general purpose programming languages are Turing Complete.
- ▶ So, why would we choose one General Purpose Programming Language over another, if they are all Turing Equivalent?

Von Neumann Architecture

- ▶ [First Draft of a Report on the EDVAC, 1946](#)
- ▶ Our next aim is to go deeper into the analysis of CC. Such an analysis, however, is dependent upon a precise knowledge of the system of orders used in controlling the device, since the function of CC is to receive these orders, to interpret them, and then either to carry them out, or to stimulate properly those organs which will carry them out. It is therefore our immediate task to provide a list of the orders which control the device, i.e. to describe the code to be used in the device, and to define the mathematical and logical meaning and the operational significance of its code words.
- ▶ The orders which are received by CC come from M, i.e. from the same place where the numerical material is stored.
- ▶ The orders which CC receives fall naturally into these four classes:
 - a) orders for CC to instruct CA to carry out one of its ten specific operations
 - b) orders for CC to cause the transfer of a standard number from one place to another;
 - c) orders for CC to transfer its own connection with M to a different point in M, with the purpose of getting its next order from there;
 - d) orders controlling the operation of the input and the output of the device

▶ **Modern Computers still have this basic architecture**

