innovate achieve lead

BITS Pilani
Pilani Campus

CS/IS F214
Logic in Computer Science

MODULE: INTRODUCTION

Logicians laid the foundation for Computing – Part II
- Turing Machines and Undecidability

TURING MACHINES: COMPUTABILITY, CHURCH-TURING EQUIVALENCE

RECALL: Recursive Functions and Lambda Calculus

- Godel proposed Recursive Functions as standard for "calculability".
- Church proposed Lambda Calculus as standard for "calculability" problems.
  - Church also proved that Recursive Functions are equivalent to Lambda Calculus

Recursive Functions and Lambda Calculus were not *acceptable* enough!

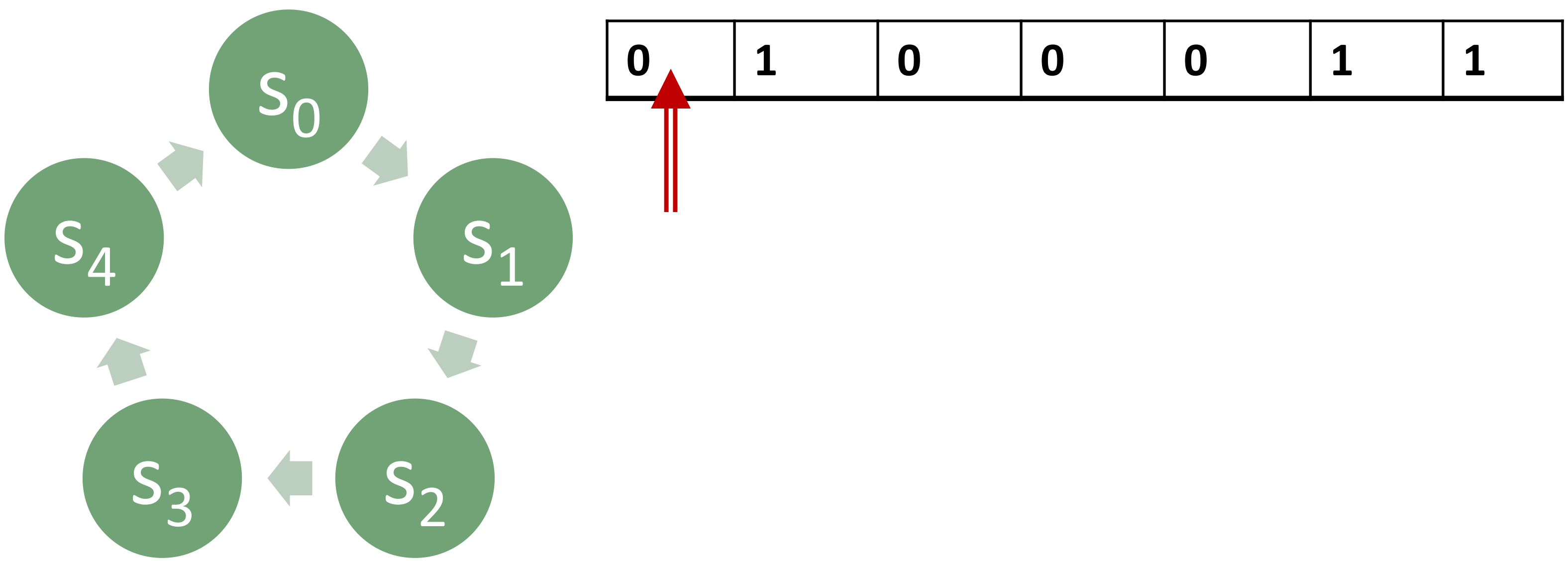
- There was one caveat with these definitions of "calculability":
  - These systems (**recursive functions** and **Lambda Calculus**) did not capture *the effort required to calculate* in their definition:
    - i.e. while it was clear one can use these functions to calculate, it was not clear what "calculate" means!

Alan Turing: Computability

- Alan Turing came up with – what are now known as – ***Turing machines*** (circa 1935):
  - yet another scheme for "calculability".
- But Turing also stated what he means by "calculable" or "computable":
  - Something is ***computable*** (i.e. calculable) if
    - it can be computed in a *finite number of steps* where
      - each step takes a *finite amount of time and resources*.

Turing Machines

- A Turing machine has:
  - a finite number of states
    - in which it can be;
  - and a tape (that may be arbitrarily long in one direction)
    - divided into cells each of which can contain one symbol



Turing Machines - Operation

- In a Turing machine, a basic step was defined as follows:
  i. the machine *reads a value from a single cell in a tape*
  ii. the machine *goes from one state to another state*
  iii. and *it may write (or overwrite) into a single cell* in the tape

Universal Turing Machines

- Turing also defined a **Universal Turing Machine (UTM)**
  - which can take the *description of any Turing machine M* and *an input I* and **run M on I**.

- The **UTM** was the first (abstract) definition of a "computing" machine.
  - This happened before the first modern (physical) computers came into existence:
    - simple versions of which were built in the late 1930s;
    - more sophisticated versions in 1940s - one of them built by a team including Turing.

Church-Turing Equivalence

- Turing worked under Church after his definition of Turing machines and:
  - proved that **Lambda Calculus** and **Turing machines** are ***equivalent***
    - i.e. what can be done with one can be done with the other

Church-Turing (Hypo)Thesis

- This resulted in ***Church-Turing Thesis***:
  - anything computed using a Turing machine is *computable* and
  - anything *computable* can be computed using a Turing machine
    - or equivalently
      - defined in Lambda calculus or
      - defined as a recursive function

- Several other definitions of computability have been proposed
  - and they have all been proved equivalent to Turing machines.

ARE THERE NON-COMPUTABLE FUNCTIONS?

Computability – General Purpose Computers

- Turning machines are often used as the foundation for proving something computable.
  - But general purpose computers are equivalent to Turing machines:
    - they have "instructions" which can be simulated by (one or more steps of) a Turing machine
    - they use ***Random Access Memory***
      - but this can be simulated on the tape in a Turing machine
        - which requires ***Sequential Access***

Computability – Algorithms and Languages

- Often, instead of defining a Turing Machine to prove computability of a given function, say **f**,
  - one may write down an algorithm computing **f**
    - as long as this algorithm can be implemented on a general purpose computer
  - or write a program in a programming language
    - as long as this program can be run on a general purpose computer

Are there functions that cannot be computed?

- Are all functions computable?
  - i.e. can any function - that one can define – be computed?
- Before providing an example of a function that is not computable we need a few definitions:
- Decision problems:
  - A problem that requires a Boolean answer (i.e. TRUE or FALSE) is said to be ***a decision problem***.
- Decidability
  - A decision problem that is computable is said to be ***decidable***.
  - A decision problem that is not computable is said to be ***undecidable***.

Undecidable Problem - Example

- Definition **(Halting Problem):**
  - *Decide whether a(n arbitrary) program will – eventually – halt (i.e. terminate).*
- **Halting Theorem:**
  - ***Halting Problem is undecidable***
    - *i.e. no program can decide whether an arbitrary program will – eventually – halt (i.e. terminate).*

Halting Problem

- (Re-Statement of the) **Halting problem:**
  - Let **H** be a program with arguments **P** and **x** such that
    - **P** is a(n arbitrary) program and
    - **x** is the input to **P** and
    - **H(P, x)** produces as output

      1 if program **P** (eventually) terminates on input **x** ;

      0 otherwise.

- Now, to prove the Halting Theorem:
  - we need to prove that *such a program H cannot exist*.

Halting Theorem - Proof

- Proof by Contradiction:
  - Assume **H**, the *halting tester*, exists.
  - Define a procedure **Diag** as follows:

```
Diag(Q) {
    while (H(Q,Q) == 1)   /* Do nothing */   ;
}
```

  - What does **H(Diag,Diag) return?**

Halting Theorem – Proof by Contradiction (continued)

<u>Assume **H(Diag,Diag)** returns 0</u>

- Then **Diag(Diag)** must not terminate
  - by definition of **H**
- The only way **Diag** would not terminate:
  - the loop in **Diag** would not terminate
    - i.e. the condition remains true
    - i.e **H(Diag, Diag) == 1**

This is a contradiction

Halting Theorem – Proof by Contradiction (continued)

Assume **H(Diag,Diag)** returns 1

- Then **Diag(Diag)** must terminate
  - by definition of **H**
- The only way **Diag** would terminate:
  - the loop in **Diag** would terminate
    - i.e. the condition is false
    - i.e **H(Diag, Diag) == 0**

This is a contradiction

Halting Theorem – Proof by Contradiction (continued)

<u>Assume H(Diag,Diag) returns 0</u>

- Then Diag(Diag) must not terminate
  - by definition of H
- The only way Diag would not terminate:
  - the loop in Diag would not terminate
    - i.e. the condition remains true
    - i.e H(Diag, Diag) == 1

This is a contradiction

<u>Assume H(Diag,Diag) returns 1</u>

- Then Diag(Diag) must terminate
  - by definition of H
- The only way Diag would terminate:
  - the loop in Diag would terminate
    - i.e. the condition is false
    - i.e H(Diag, Diag) == 0

This is a contradiction

- By **LEM**, H(Diag, Diag) must return 0 or 1; either answer results in a contradiction
  - ***i.e. our assumption that H exists must be incorrect!***