SCC.312 Languages and Compilation (Week 15)

Paul Rayson

p.rayson@lancaster.ac.uk

Last Week

- “*uvwxy*” Theorem

Reminder

- Context sensitive & unrestricted grammars

- The Turing Machine
  – Linear Bounded

- The Complete Chomsky Hierarchy

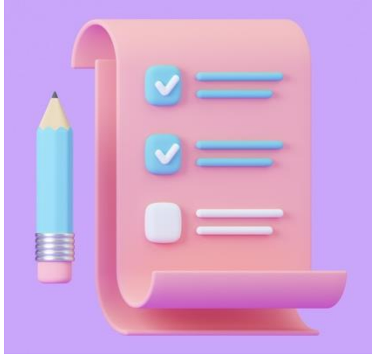
This week's Topics

- Machines and Computation

- Abstract Machines
  - Finite State Transducers
  - More on Turing Machines
    - Universal Turing Machine

- The Halting Problem

Learning Objectives



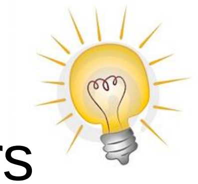
1. appreciate the connection between formal grammars their computational implications
2. appreciate the implications and importance of Turing’s Thesis
3. understand the concept of a Universal Turing Machine
4. understand that there are some (well-specified) problems that cannot be solved (for example the halting problem)

This is a good time to review all 24 learning objectives so far

Computation

Machines and Computation

- We can regard phase structure grammars as *computational*, as well as *linguistic* devices
- **Computable languages** are those that can be processed by computers
  - or abstract machines that represent computers
- For example consider the following:
  - $\{a^i b^j c^{i * j} : i, j \geq 1\}$ (G_{11}) is essentially a model of a process for multiplying two arbitrary length numbers



Machines and Computation

- A compiler is essentially a machine that will decide if a program is a valid based on its understanding of the syntax
- A programming language, specified by a grammar, is therefore a computable one
  - Note: the compiler does not understand what a program does! It will compile any program so long as it is syntactically correct

Machines and Computation

- Here, we consider the types of computation that can be carried out by two abstract machines
  - Finite State Transducers

  - Turing Machines 

Finite State Transducers

- A **finite state transducer** (FST) is essentially a finite state recogniser that also produces output.
- So a finite state recogniser is a restricted finite state transducer
- It may also be called a **finite state machine with output**

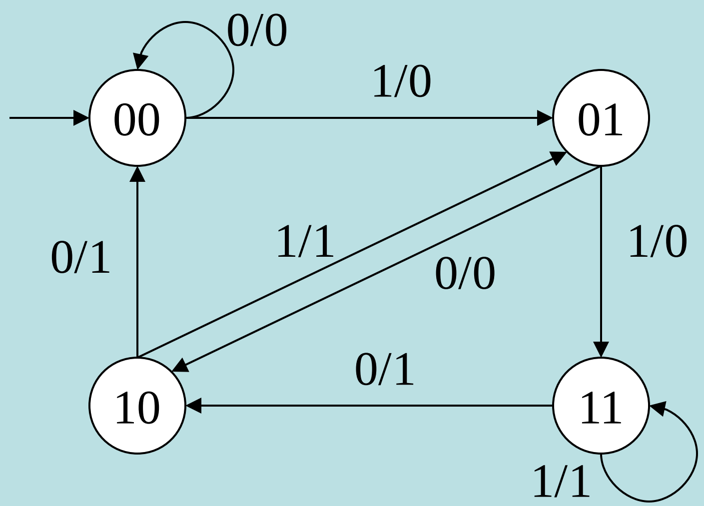
Finite State Transducer Output

- Each time the finite state transducer traverses an arc it reads an input symbol as normal
- For every input symbol it writes an output symbol
- The output symbol goes to an output string that cannot be read by the finite state transducer
  - e.g. *a*/*b* is read an *a* and output a *b*
- Because it produces output a FST can be used to perform functions other than language accepting

FSTs as “Memory” Devices

- FSTs can be used to model simple memory tasks, such as a “shift” operations

A binary “two digit shift” machine



Example:

Input:

1	1	0	1	1	1	0

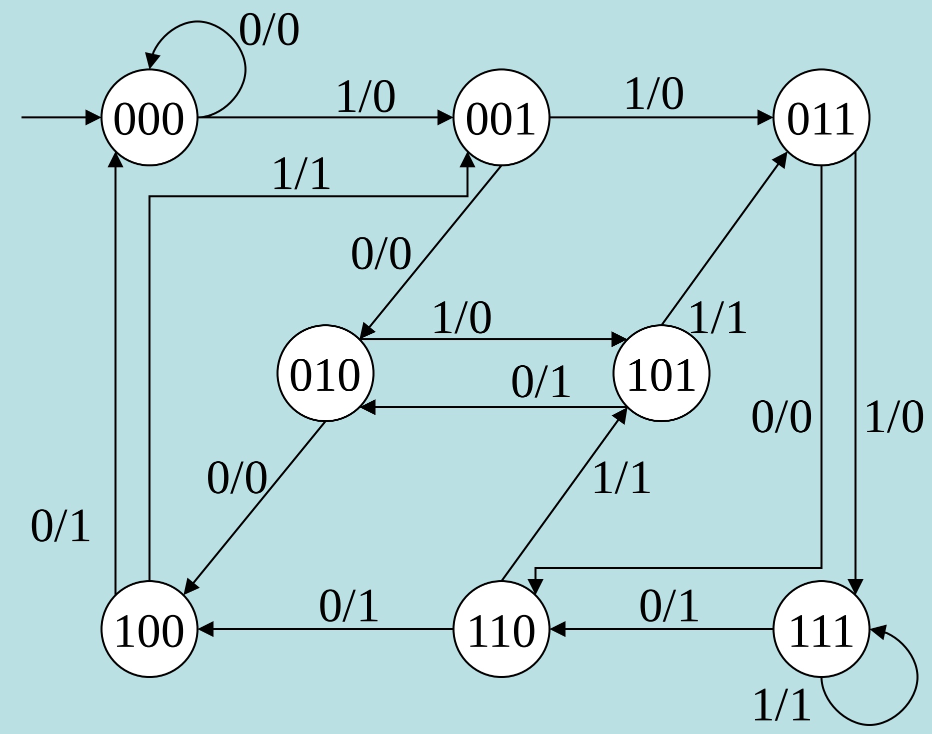
Output:

0	0	1	1	0	1	1

Start of shifted input sequence

FSTs as “Memory” Devices

A binary “three digit shift” machine



Example:

Input:

1	1	0	1	1	1	0	0	1	1	0

Output:

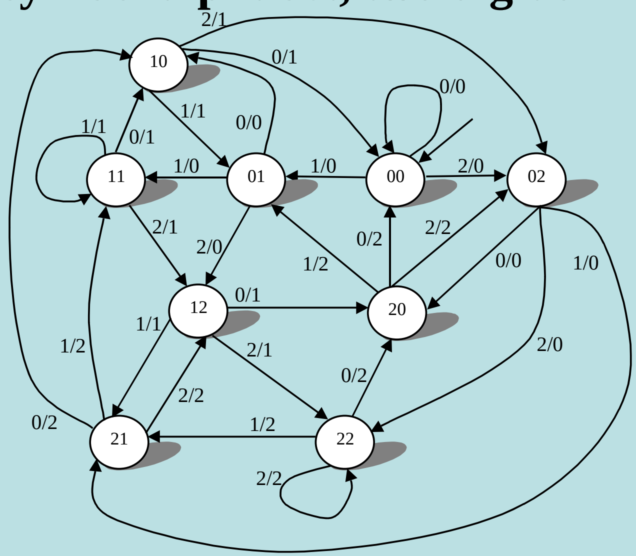
0	0	0	1	1	0	1	1	1	0	0

Start of shifted input sequence

FSTs as “Memory” Devices

- Available for any size alphabet and any sized shift, but both must be a fixed size

Three symbol alphabet, two digit shift



Example:

Input:

1	2	2	1	1	0	0	2	1

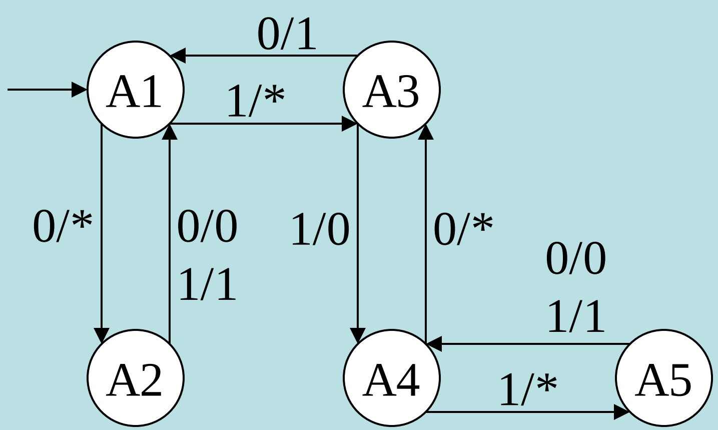
Output:

0	0	1	2	2	1	1	0	0

Start of shifted input sequence

FSTs For Computation – Add

- Adding two arbitrarily long binary numbers



Example:

01101 + 00110 (i.e.13 + 6)

The numbers are reversed and presented one digit of each number in turn. Doesn't matter which one first

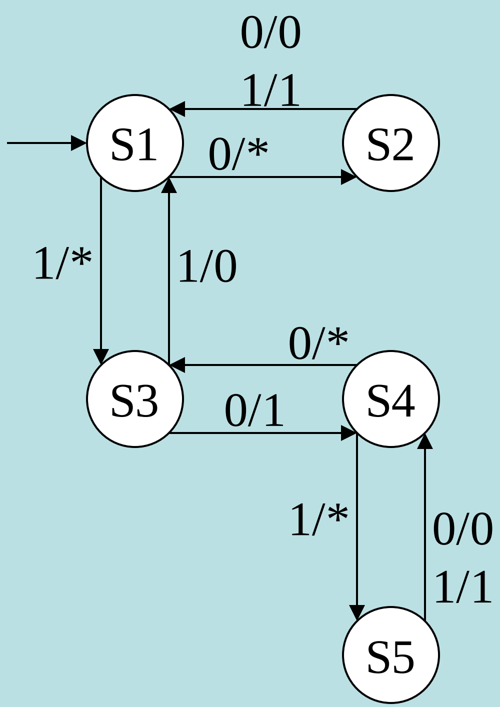
Input: **0110110100**

Output: ***1*1*0*0*1**

When the output is reversed we get: **10011** (i.e. 19)

FSTs For Computation – Subtract

- Subtract 2 arbitrarily long binary numbers



Example:

1101 - 0110 (i.e.13 - 6)

The numbers are reversed and presented one digit of each number in turn. Begins with the second number

Input: **01101101**

Output: ***1*1*1*0**

When the output is reversed we get: **0111** (i.e. 7)

A FSTs For Multiplication?

- So we can design a finite state transducer for addition and subtraction where two arbitrarily long binary numbers can be handled
- But can we design an FST to multiply two arbitrarily long binary numbers?
- No!
- Proof?
  - Lets assume it is possible create a multiplication finite state transducer…

A FSTs For Multiplication?

- We assume that the FST, which we shall call M, can multiply arbitrary length binary numbers
  - Lets suppose M has k states and we ask it to do $2^k \times 2^k$
  - $2^k \times 2^k = 2^{2k}$ = a 1 followed by 2k 0's
    - $2^3 = 1000$, $2^3 \times 2^3 = 2^6 = 1000000$ (i.e. 8 x 8 = 64)
  - Lets assume M only uses one state to process corresponding digits of both numbers simultaneously, print out the 1 and the first k zeros of the answer
  - M now only has k-1 states to print out the remaining k zeros of the answer

Modelling a Computer

- A computer at a given moment in time is essentially a very large FST
- Consider a very very small machine such as 32k
  - Such a machine has $2^{262,144}$ states
  - Suppose we waited for the machine to repeat a state
  - If the machine executed at 10^{12} state changes/second
  - It could take 10^{4000} years for this to happen
  - Waiting for our tiny computer to repeat a state because it happens to be an FST is a silly thing to do!

Beyond Finite State Transducers

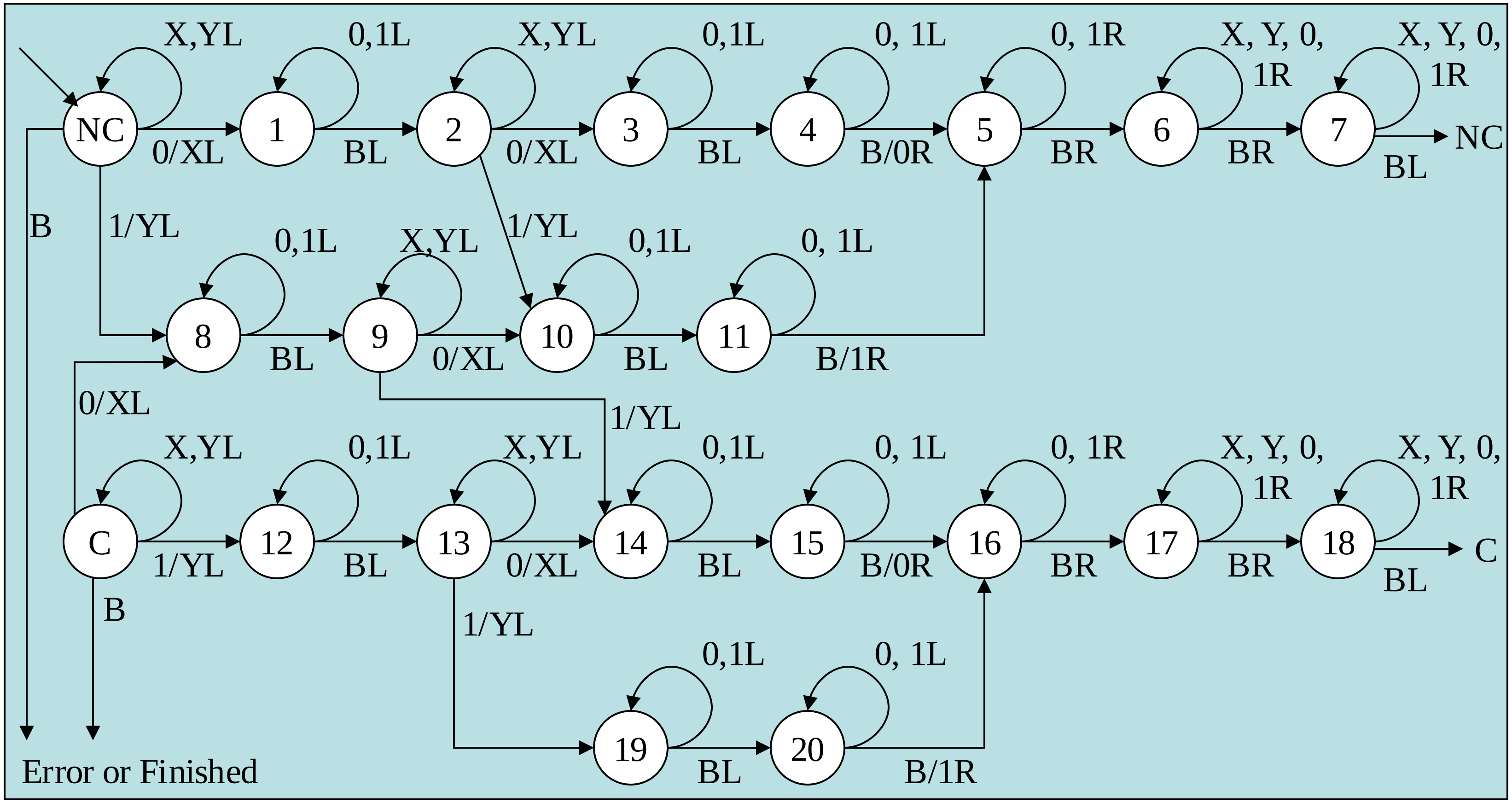
- As a computational device it has its limitations
  - A FST cannot do multiplication
  - Not a useful practical model of a digital computer
- It does share many properties with real computers
- However, a sufficiently powerful abstract machine is not that dissimilar from an FST
  - It moves back and forth through it input
  - Instead of separate output, it overwrites its input
  - Any guesses?

Turing Machine Reminder

Reminder

- A Turing machine is a FST equipped with a one dimensional tape
- The tape is divided into squares that can be occupied by a symbol or a blank (B) and extends infinitely in both directions
- The machine has a read/write head that at any time points to a given square on the tape

A TM For Binary Addition



A TM For Binary Addition

- The abbreviation “X,YL” means “X/XL, Y/YL”
- In the diagram the return arc to the NC (no carry) or C (carry) states are shown as arcs to the right
- The machine changes the bits it has read so that they are not considered again
  - X for a 0 and Y for a 1

A TM For Binary Addition

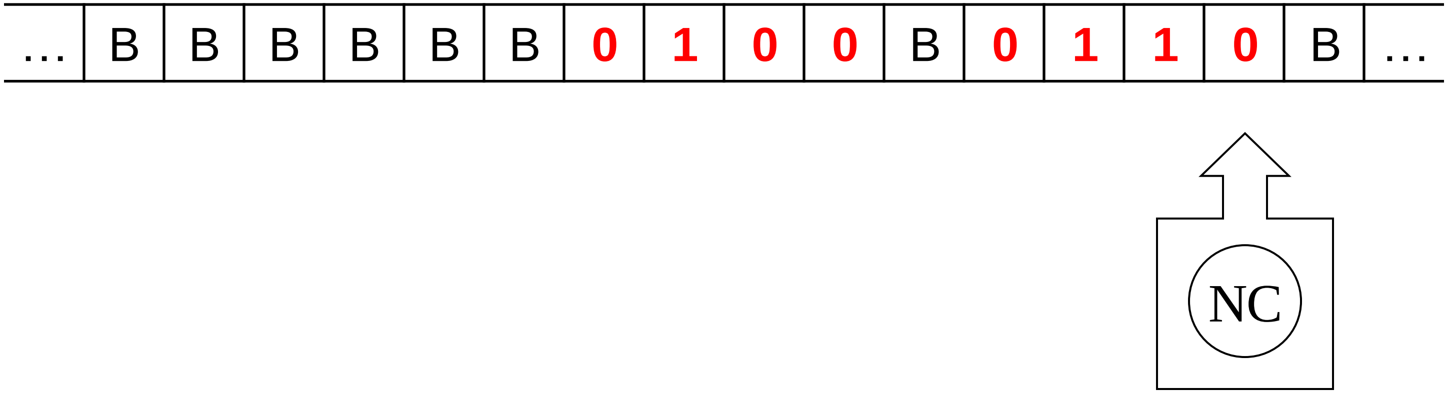
- The machine is incomplete!
  - Needs to tidy up (change X, Y back to 0, 1), then halt
  - It also needs to write any final carry
- We have not bothered with any error checking
  - e.g. that the binary numbers are both the same length
  - Usually ignore error-checking as we are trying to show what calculations are possible theoretically

TM Addition – Start

- Start with 2 binary numbers (assumed to be the same length) separated by a blank
  - e.g. 0100 + 0110 (i.e. 4 + 6)
  - Note: the numbers have not been reversed
- The sum will be written at the left-hand end of the tape separated by a blank

TM Addition – Start

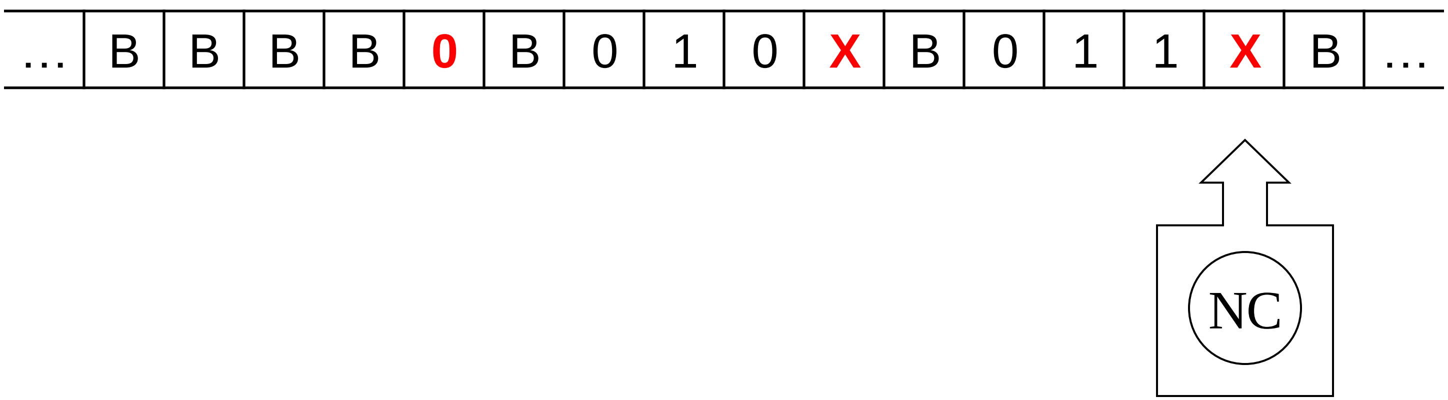
- Example: 0100 + 0110 (i.e. 4 + 6)
- We start in the state NC (no carry)



- The r/w head is over the right hand (least significant) digit of the right hand number

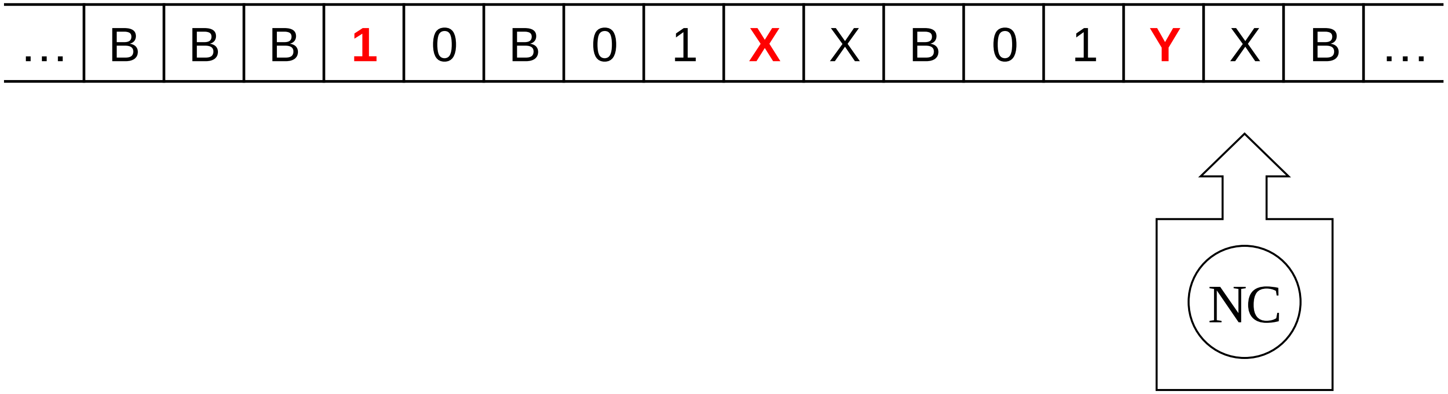
TM Addition – Example

- After the 1st round (bits 0 and 0) we have:



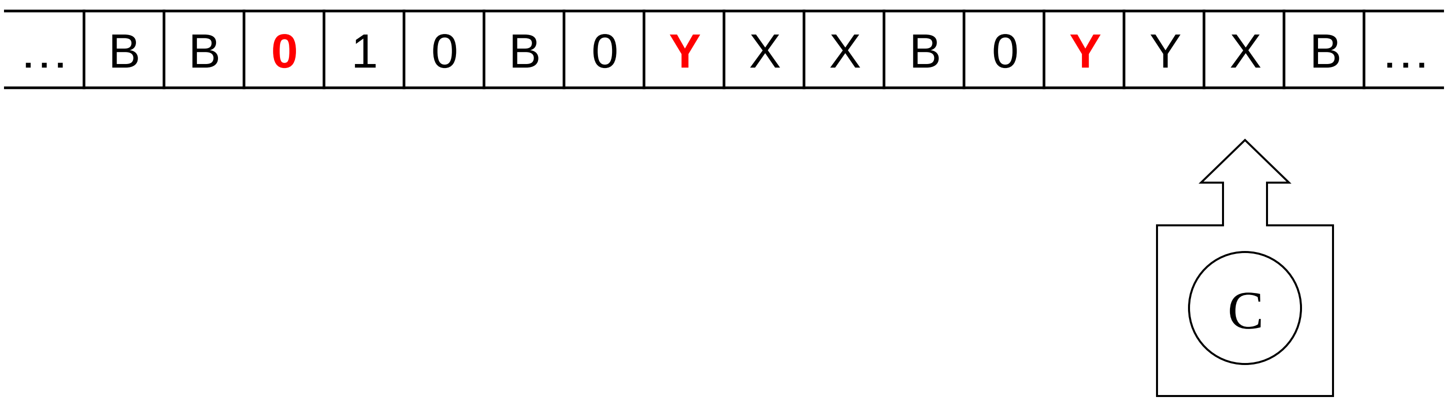
TM Addition – Example

- After the 2nd round (bits 1 and 0) we have:



TM Addition – Example

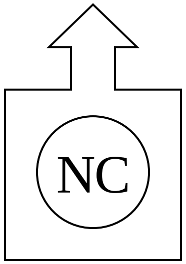
- After the 3rd round (bits 1 and 1) we have:



TM Addition – Example

- After the 4th round (bits 0 and 0, and a carry) we have:

…	B	**1**	0	1	0	B	**X**	Y	X	X	B	**X**	Y	Y	X	B	…



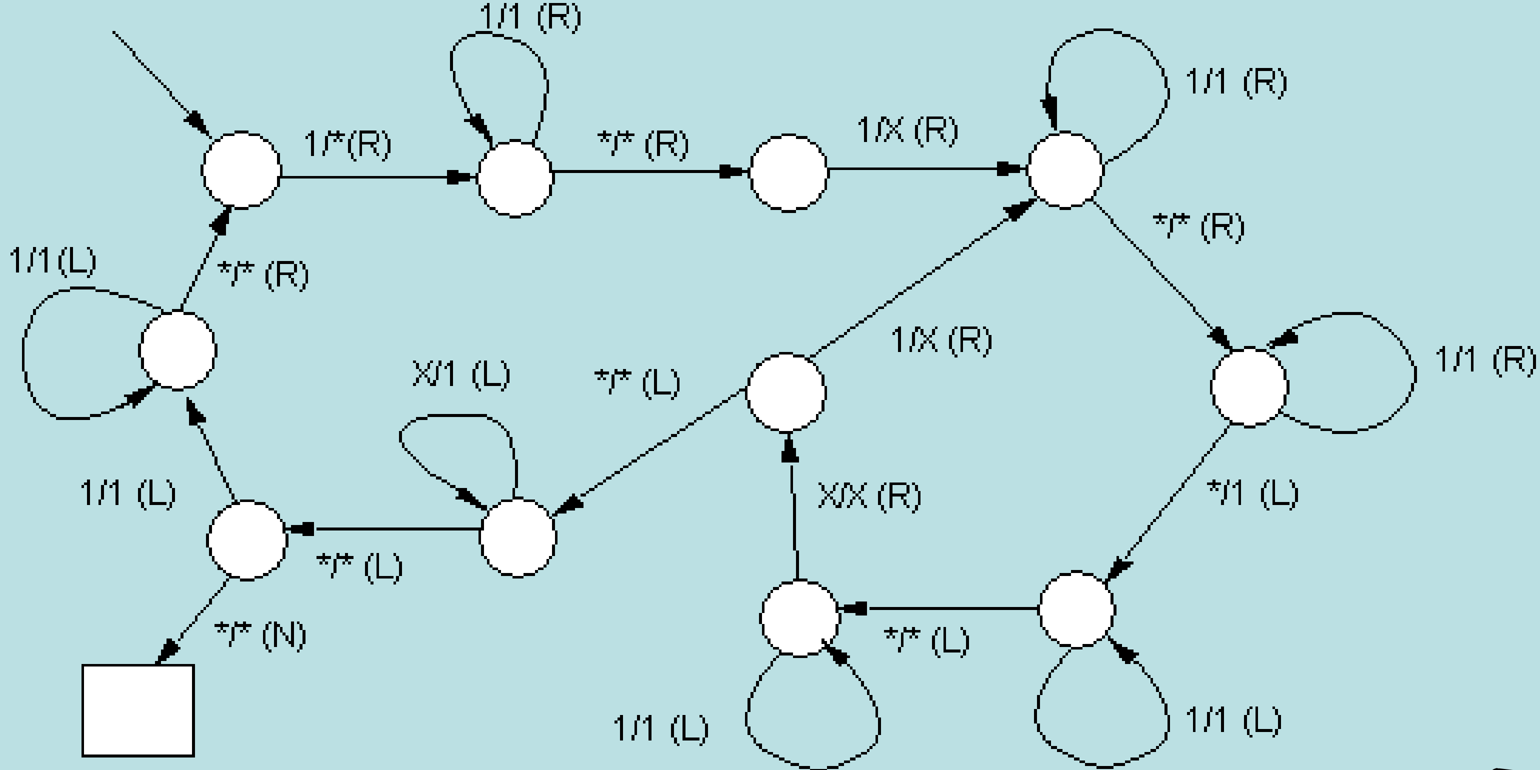
- *Finished!*
  - – 0100 + 0110 = 1010 (i.e. 4 + 6 = 10)
- Need to perform tidying up tasks

Turing Machines as Computers

- It is possible to design a Turing machine to do all the standard arithmetic operations and (in principle) any other well-defined task
  - e.g. to check that a given number is prime
- A Turing machine can perform computational tasks that are beyond the power of the finite state transducer
  - e.g. multiplication

A TM for Multiplication

A Unary Multiplier (unary numbers are of the form 11111 = 5, * = B)



A TM for Multiplication

S	*	1	1	1	*	1	1	1	*	*	*	*	*	*	*	*	*	*	*	*
•	*	*	1	1	*	X	X	X	*	1	1	1	*	*	*	*	*	*	*	*
•	*	*	1	1	*	1	1	1	*	1	1	1	*	*	*	*	*	*	*	*
•	*	*	*	1	*	X	X	X	*	1	1	1	1	1	1	*	*	*	*	*
•	*	*	*	1	*	1	1	1	*	1	1	1	1	1	1	*	*	*	*	*
•	*	*	*	*	*	X	X	X	*	1	1	1	1	1	1	1	1	1	*	*
H	*	*	*	*	*	1	1	1	*	1	1	1	1	1	1	1	1	1	*	*

The Power of Turing Machines

- Any well-defined computational task can be modelled by some Turing machine
- This includes any computational task that can be carried out by a computer
- Turing machines are more powerful than any given computer, as they effectively have an infinite amount of memory

Universal Turing Machines

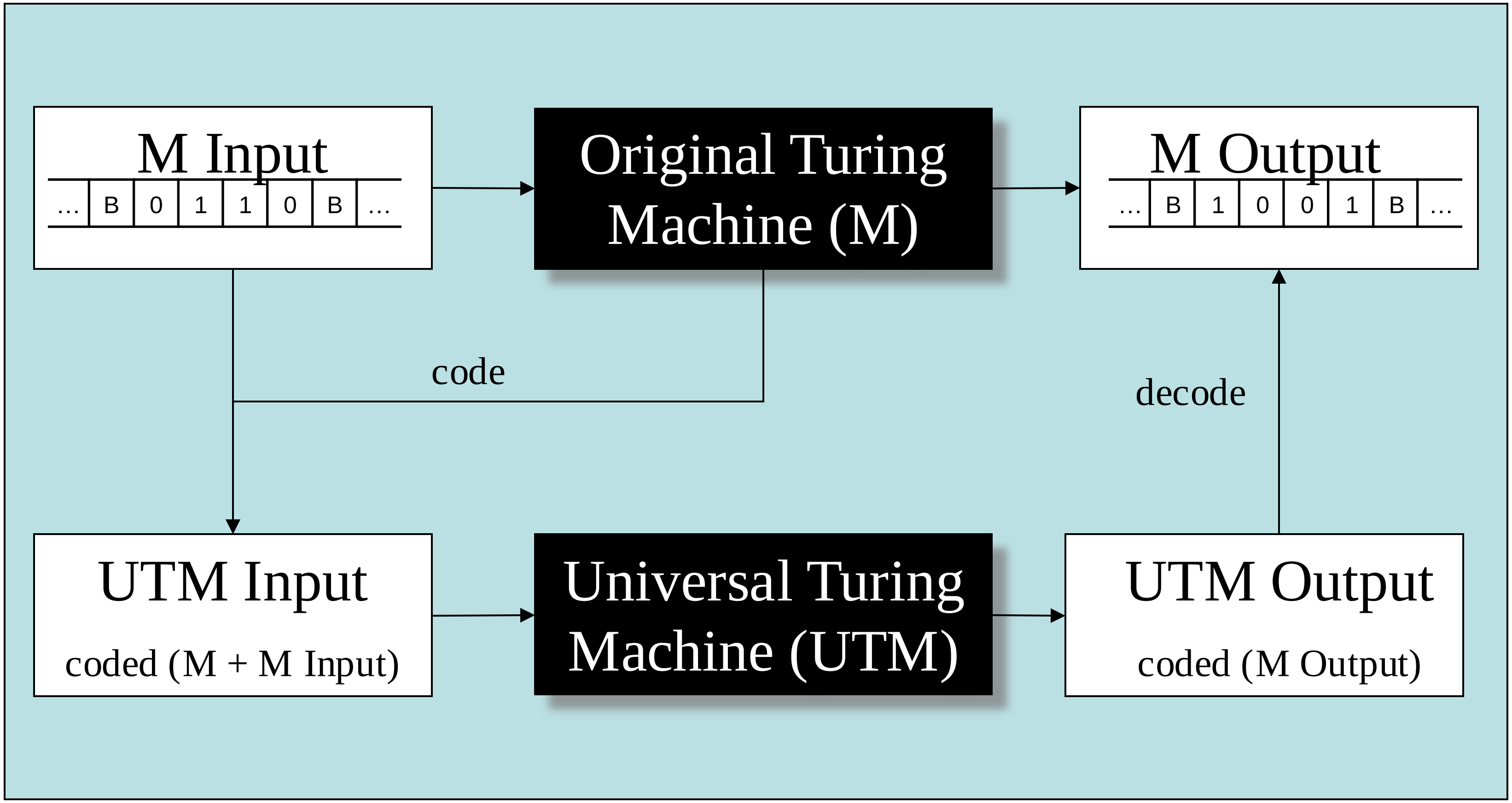
Universal Turing Machines

- So far we have designed a separate Turing machine for each problem
- A **Universal Turing Machine** (UTM) effectively runs any TM as if it were a program
- In a sense this UTM is the counterpart of a computer, but is of course more powerful than any computer could ever be

Universal Turing Machines (UTM)

- A UTM is an ordinary Turing machine that models the behaviour of a TM
- It receives as input a coded version of the TM and a coded version of the input string
- It produces as output a coded version of the output that would be produced by the original machine

The Basic Principle Of A UTM

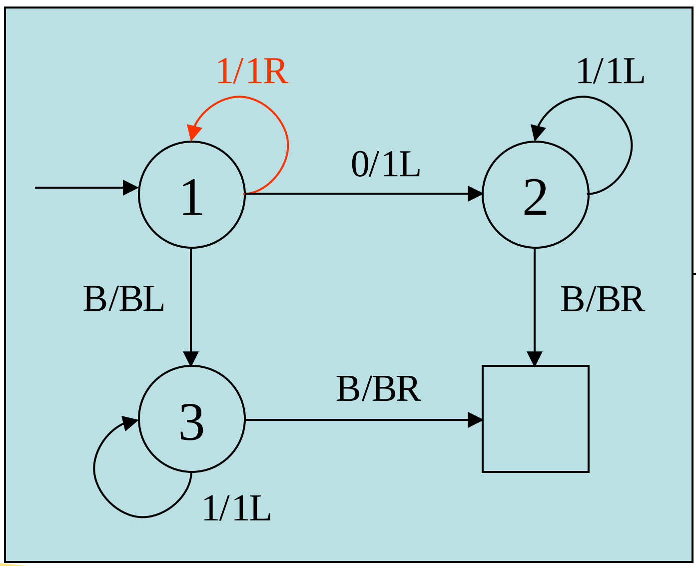


Coding A Turing Machine

- How do we code a UTM?
- We can represent any Turing machine as a set of **quintuples** (groups of 5 things)
  1. Current state
  2. Symbol being read
  3. Symbol to write
  4. Move (L or R)
  5. New State

Coding a UTM

- This example looks like:
  - Start state (1) is the first specified
  - Halt state (4) has no quintuples



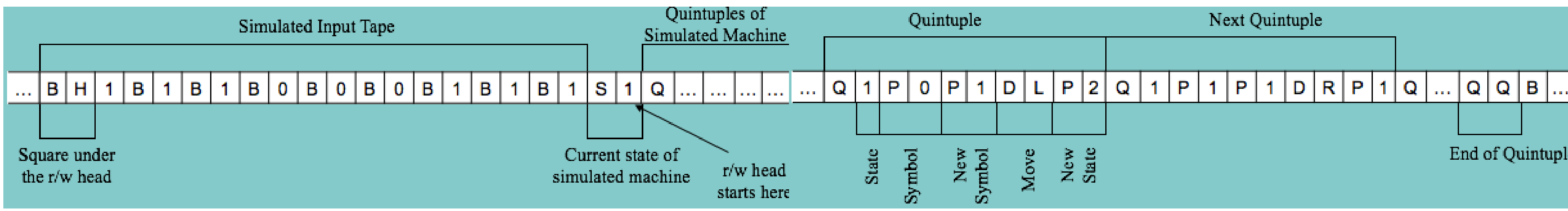
Current State S	Read Symbol R	Write Symbol W	Move M	New State S
1	0	1	L	2
1	1	1	R	1
1	B	B	L	3
2	1	1	L	2
2	B	B	R	4
3	1	1	L	3
3	B	B	R	4

Each arc is a quintuple

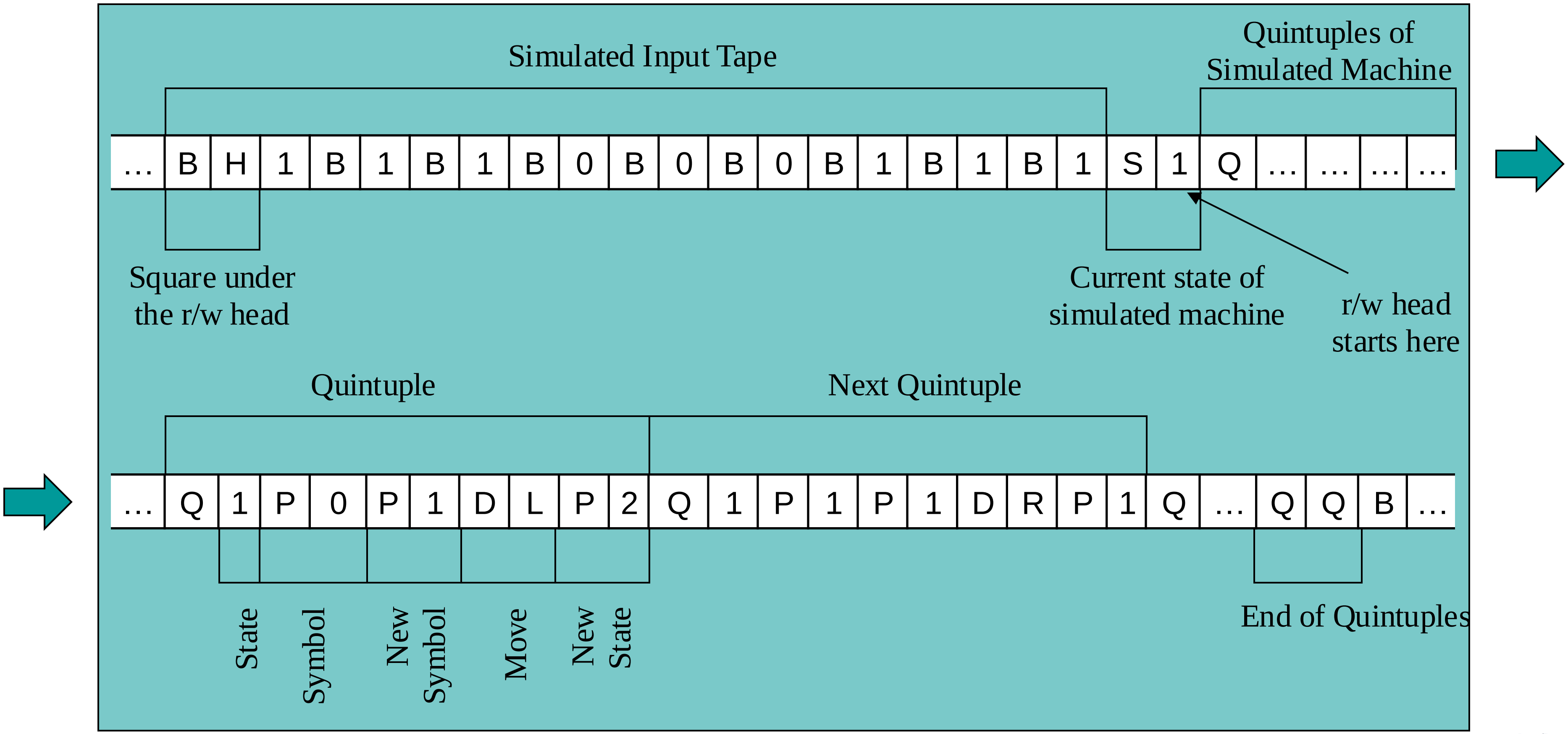
Setting Up A UTM

- On the UTM tape there is:
  - the quintuples of the machine we want to simulate
  - the start state of the simulated machine (could be 0)
  - the contents of the (non-blank) part of the data
  - a mark on the simulated tape to show where the simulated read-write head is
- We start the UTM going in state 1 with the UTM r/w head positioned over the left-most bit of the current state

The Universal TM Tape



The Universal TM Tape



Outline of the UTM Process

1. Find an arc
2. Match the arc
3. Follow the arc

Outline of the Process – Step 1

Find an arc

- Find the first quintuple matching the current state of the simulated machine
- If the quintuple …
  - Fails – change the Q to F and look for next Q
  - Succeeds – go onto step 2
- If the first Q is followed immediately by a second Q we are at the end of the quintuples and there was no match – so halt

Outline of the Process – Step 2

Match the arc

- Compare the data symbol under the simulated machine r/w head with the quintuple found in step 1
  - Matches – go to step 3
  - Does not match - change the quintuple's Q to F and go back to part 1 to find another quintuple
    - Have to undo any changes made from steps 1 and 2

Outline of the Process – Step 3

Follow the arc

- Copy the new data symbol from the matching quintuple to the data square under the read-write head
- Move the simulated r/w head as appropriate
- Copy the new state from the matching quintuple to the current state on the tape
- Tidy up – i.e. rewrite F as Q to match them again
- Start from step 1 and repeat…

Other Representation Techniques

- The example in the Parkes book uses 3 tapes
  - Tape 1: stores the coded machine M followed by the coded input to M
  - Tape 2: used to work on a copy of coded input
  - Tape 3: holds a sequence of 0’s representing M’s current state
- Each character is represented as a number of 0’s separated by a single 1
  - Could also be represented in any form of binary
  - Use the X, Y replacement technique to check matches

Implications of the UTM

- It is possible to code a UTM to be presented to another UTM for simulation
- A UTM is a close analogy to the stored program computer
- A computer can only be as powerful as a UTM given unlimited storage
  - It cannot be more powerful

Implications of the UTM

- If M produces a solution given a particular input then the UTM produces a coded version of the same solution
- The UTM may or may not halt, depending on the machine and the data it is simulating
  - If M does halt, we can read off the result from the simulated tape
  - If M does not halt, by going into an infinite loop, then the UTM will do the same

The Halting Problem

The Halting Problem

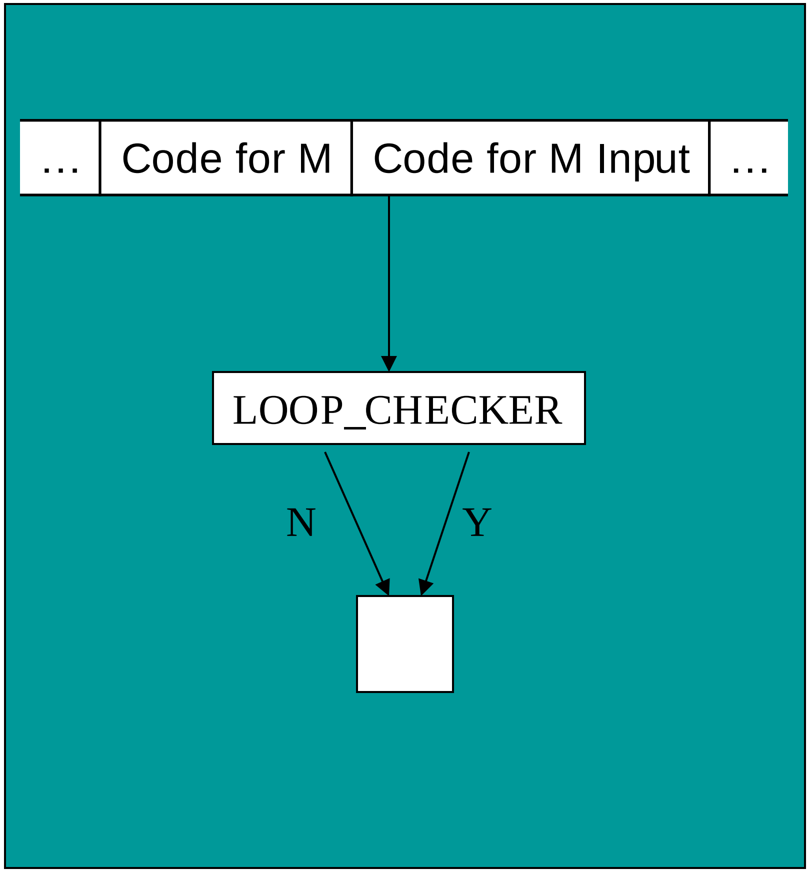
- There are problems which, though well-specified, are inherently unsolvable
- One of these is called the **halting problem**

The Halting Problem Defined

- A useful program would be one that detects infinite loops in other programs
  - i.e. Does running program P on data D stop or loop forever?
  - It would form a useful part of a compiler
- It is possible to prove that we cannot write a program that can always answer this question
- Proof?
  - Once again, we start by assuming that we can write such a program for a Turing machine…

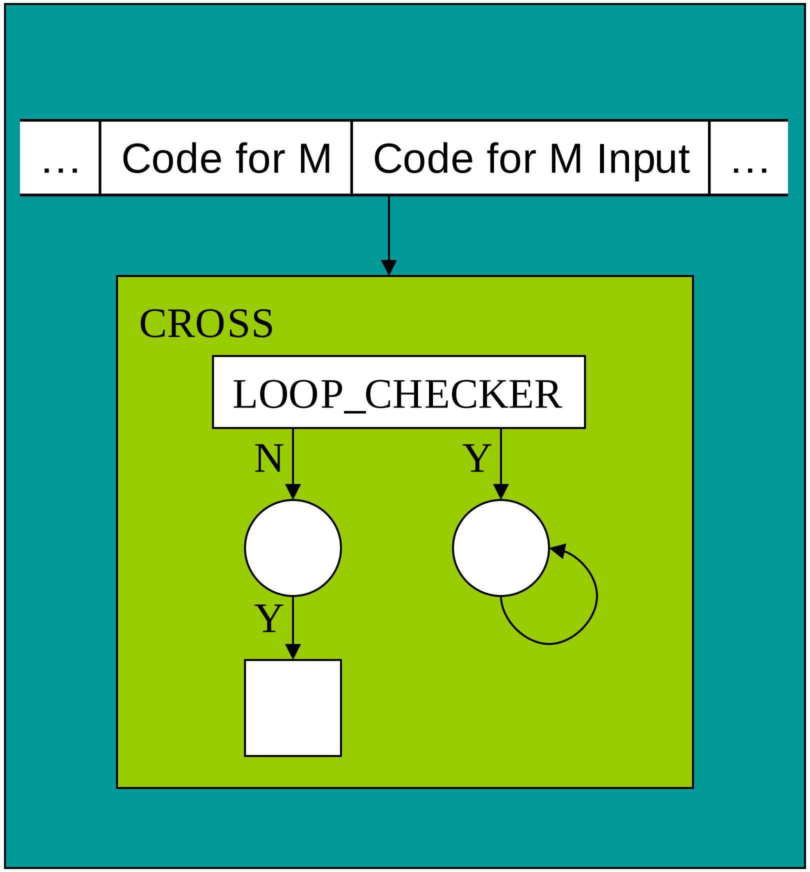
The Halting Problem – Phase 1

- We assume we have a TM, called M, and a UTM called LOOP_CHECKER that:
  - Prints Y if M would halt on its input
  - Prints N if M goes into an infinite loop
  - Both of these halt



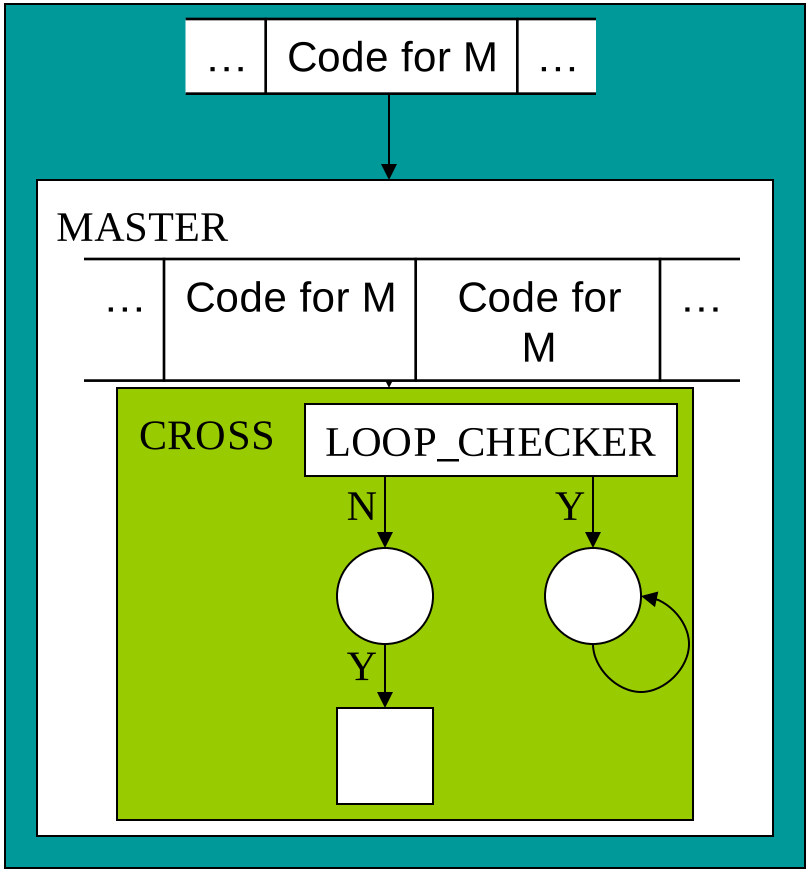
The Halting Problem – Phase 2

- Next we create a UTM called CROSS that
  - Prints Y and halts if M goes into an infinite loop (i.e. didn’t halt)
  - Loops indefinitely if M would halt on the input



The Halting Problem – Phase 3

- Then we create a UTM called MASTER
  - Copies the machine code and presents both copies to CROSS
  - MASTER now simulates the M on a description of M



The Halting Problem – Phase 4

- Finally, we give MASTER a copy of its own code
- MASTER now simulates its own behaviour on an input tape that is a coded version of itself.
- Does MASTER halt?
  - MASTER halts (given a description of itself as input) if it does not halt (given a description of itself as input)!
  - MASTER does not halt (given a description of itself as input) if it halts (given a description of itself as input)!

The Halting Problem

- The whole thing is a contradiction
- Our original assumption is obviously false
- The Halting Problem is only partially solvable
  - i.e. on those cases where M would halt
- We can prove that other problems are unsolvable, by showing that a solution would provide a solution to the halting problem
- This problem was first proved by Turing in 1936

Turing’s Thesis

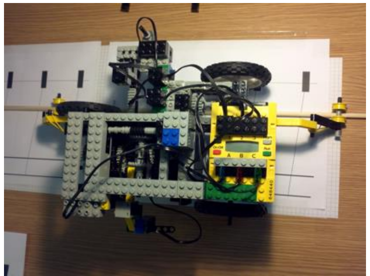
- Turing’s Thesis was originally posited in the 1930s and has yet to be disproved
- Essentially, Turing’s thesis says:
  - Any well defined information processing task can be carried out by some Turing machine

A *thesis* is something asserted as being true

Turing’s Thesis

- Turing’s thesis tells us that:
  - There is no machine (either real or abstract) that is more powerful, in terms of what it can compute, than the Turing machine.
  - The computational limitations of the Turing machine are the least limitations of any real computer.
- So, there is nothing more powerful than a TM
  - In terms of whether or not a task can be carried out
  - Not necessarily in terms of how many steps need to be carried out or how much storage is required

You’ve witnessed this inefficiency in the palindrome problems



Turing completeness

- Any system of rules is "Turing complete" if it can be used to simulate *any* Turing Machine
  - e.g. computer instruction set, imperative programming language with conditional branching, lambda calculus, Charles Babbage's analytical engine
- Two such systems are "Turing equivalent" if each one can be used to simulate the other

Further reading: see the Church-Turing thesis

Week 15 summary

- Phrase structure grammars are computational and linguistic devices
- Finite state transducers have their uses as models of computations, but are restricted
- Universal Turing machines can run other TMs and are a close analogy to the stored program computer
- The halting problem is only partially solvable
- There is nothing more powerful than a TM

The final drop-in practical labs for the first half of the module will continue this week

Summary “Languages” (weeks 11-15)

- In this part of the module have covered:

  - Phrase Structure Grammars
    - Regular, context free/sensitive, unrestricted

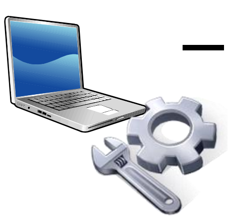


  - Their equivalent abstract machines
    - Finite state, pushdown recognisers, Turing machines

  - Their uses and implications

    - Equivalence, syntax, semantics and ambiguity

  - Chomsky Hierarchy



  - Abstract machines as a model of the computer

https://pixabay.com/photos/the-end-sand-end-beach-text-1544913/