Turing Machines

The Ultimate Computational Model

The Foundation of Computer Science

230 A. M. TURING [Nov. 12,

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

Alan Turing's 1936 paper: "On Computable Numbers, with an Application to the Entscheidungsproblem"

This single paper laid the foundation for modern computer science

Learning Objectives

By the end of this lecture, you will be able to:

- **Define** what a Turing Machine is and how it works
- **Trace** TM execution on sample inputs
- **Design** simple TMs for basic computations
- **Understand** why TMs are the theoretical model of computation
- **Implement** virtual TMs in Java

Recall: Finite Automata Limitations

DFAs and NFAs:

- Fixed, finite memory (states)
- Can only read input left-to-right once
- Cannot write or modify input

Cannot recognize: 0^n1^n and other non-regular languages

We need something more powerful...

The Turing Machine Model

Only slightly more complex than DFAs, but infinitely more powerful!

Three key enhancements to DFAs:

1. **Enhanced Tape**
2. **Reject States**
3. **Halting Behavior**

Enhancement 1: The Tape

Capabilities:

- **Move both left and right**
- **Read and write** symbols
- **Infinite** in both directions
- Special blank symbol: ⊔

State Transitions Include:

- Input symbol read
- Output symbol to write
- Direction: L or R

Example: `0:1,R`

- Read 0, write 1, move Right

Enhancement 2: Reject States

DFAs: Only accept states (implicit rejection)

TMs: Explicit accept AND reject states

- Needed because TMs can run indefinitely
- Must explicitly specify rejection

Enhancement 3: Halting

Critical Property:

When a TM reaches an **accept** or **reject** state:

- It **stops immediately**
- No further processing

This defines the computational output

TM State Diagram Notation

Transition format: `input:output,direction`

Example: `0:0,R`

- Read: 0
- Write: 0
- Move: Right

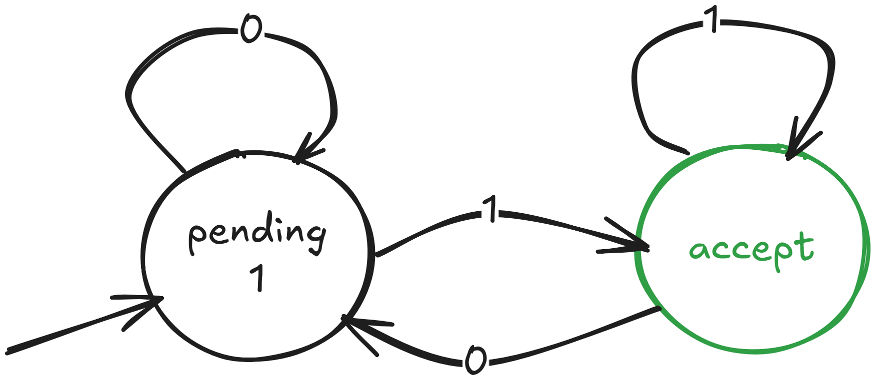
Shorthand:

- Unlabeled input → any other symbol e.g. `:1,L`
- Unlabeled output → same as input e.g. `1:L`
- No transition defined → reject

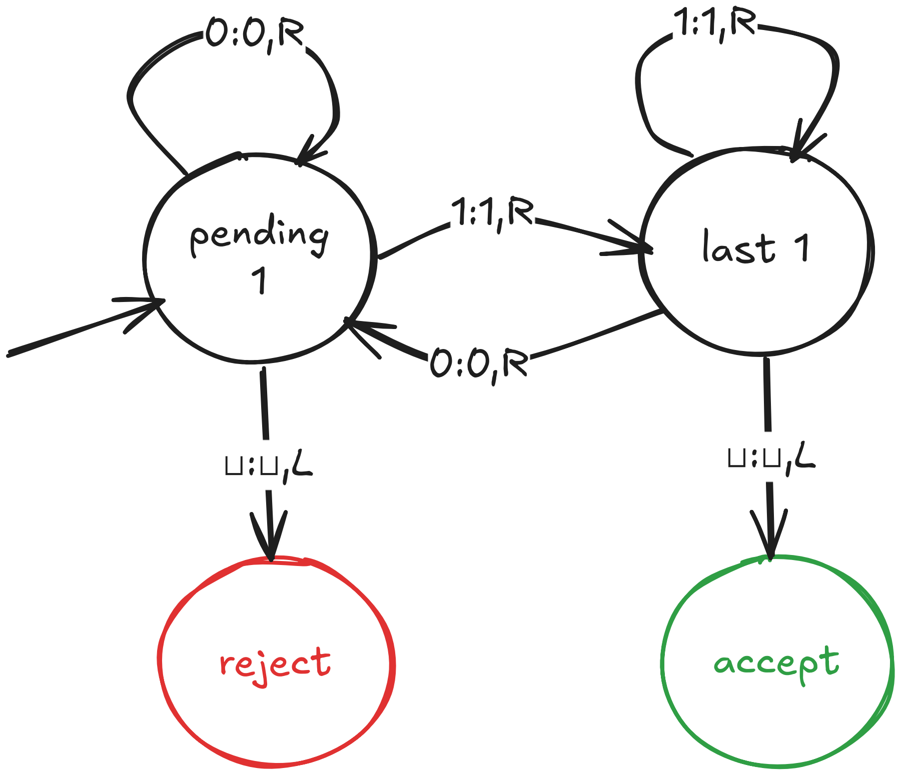
Example: DFA as TM

Language of binary strings ending in 1

DFA

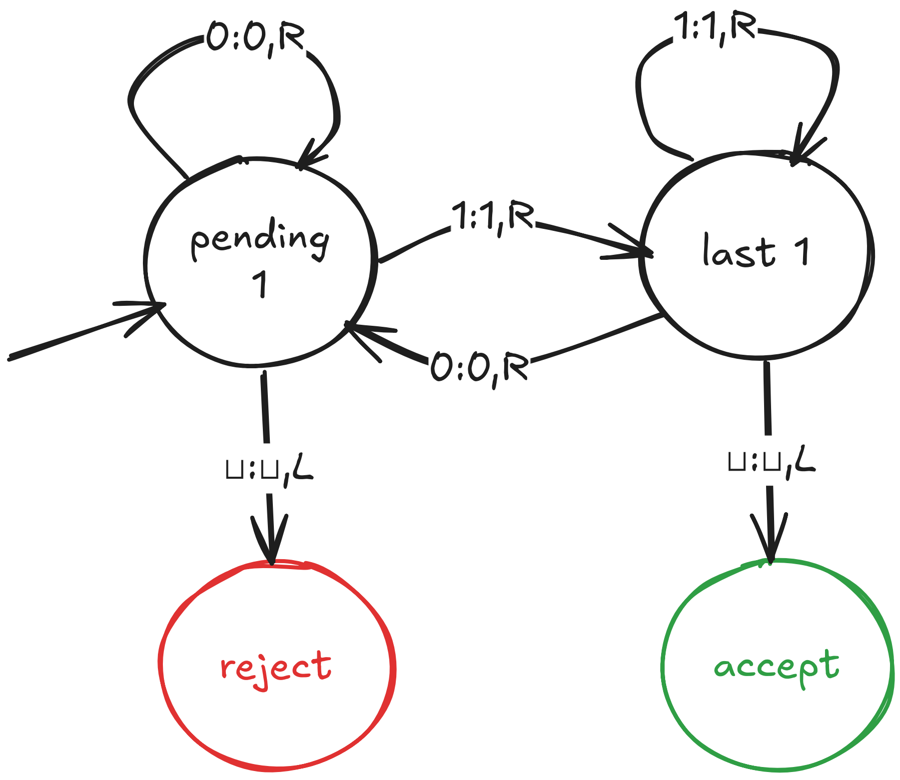


TM

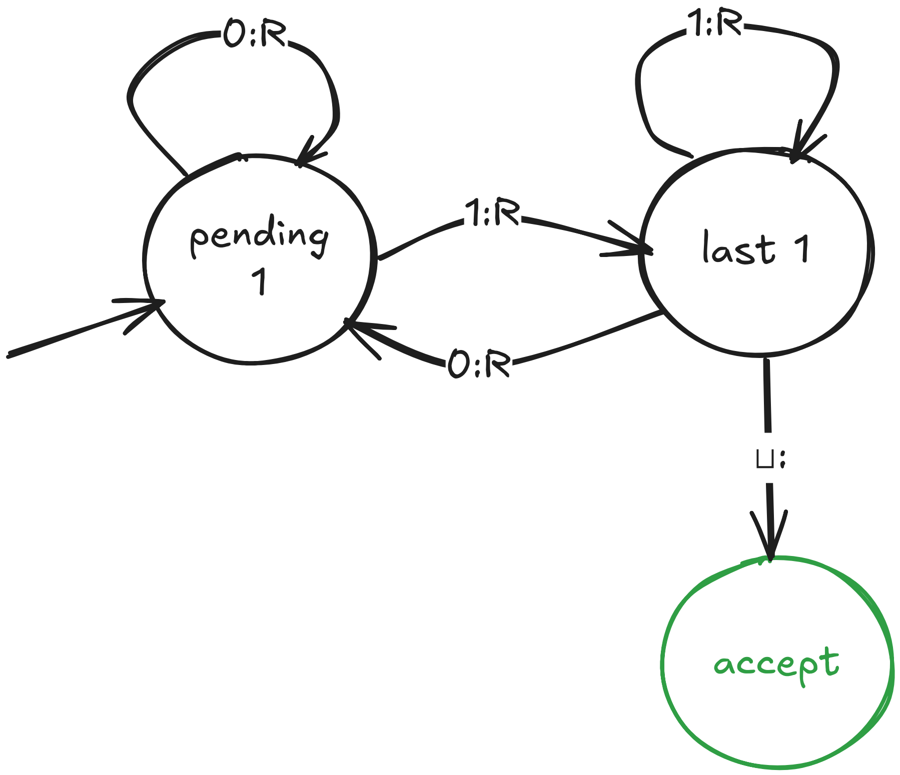


TM Shorthand

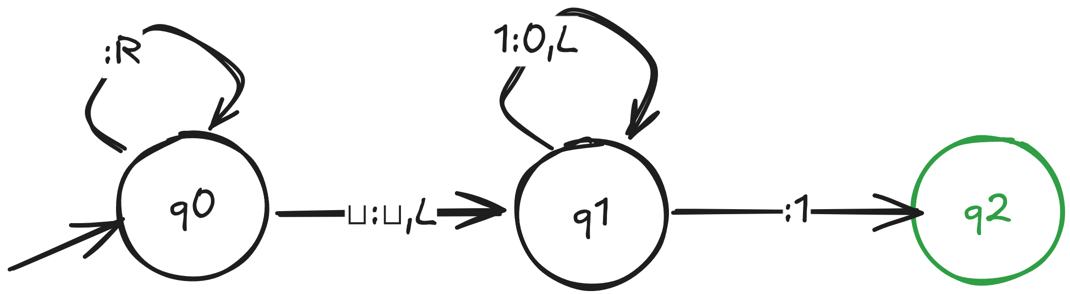
TM 1



TM 2



Example w/ output: Binary Incrementer



Example trace for 101 (5_{10}):

State	Tape Position	Action
q0	**1** 0 1 ⊔	R
q0	1 **0** 1 ⊔	R
q0	1 0 **1** ⊔	R
q0	1 0 1 **⊔**	L, q1
q1	1 0 **1** ⊔	0, L
q1	1 **0** 0 ⊔	1, q2
q2	110	accept

Strategy:

1. Scan right to end of input
2. Move left, flipping 1s to 0s
3. When you hit a 0 or ⊔, flip to 1 and stop

🎯

Active Learning: Trace the Incrementer

Given input: `111`

Questions:

1. What will the final output be?
2. How many state transitions occur?
3. What if input is `1111111` (all 1s)?

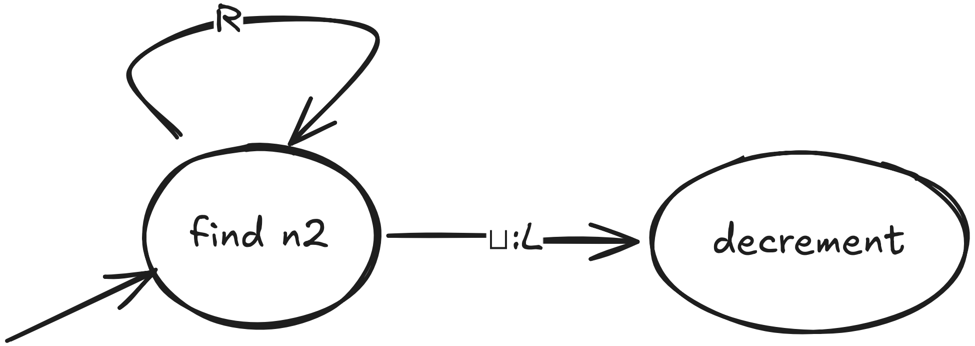
Example: Binary Adder

Input format: `␣101+10␣`, **Output format:** `␣111␣` (5 + 2 = 7)

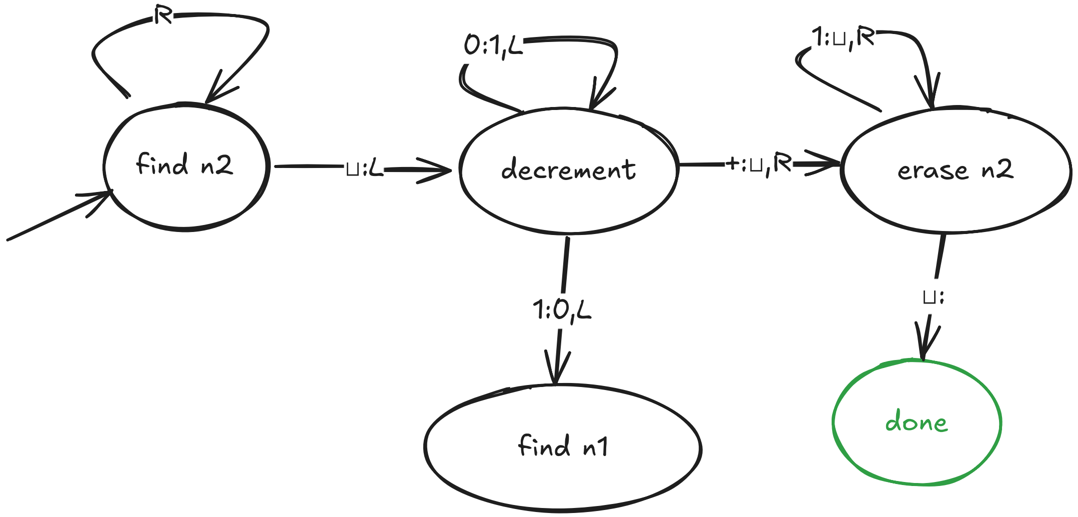
Strategy:

1. Scan right to the end of n_2
2. Decrement n_2
3. If n_2 was all 0s before the decrement (resulting in all 1s after the decrement):
    i. Replace `+111...␣` with `␣␣␣...`
    ii. Accept
4. Scan left to the end of n_1
5. Increment n_1
6. Repeat from step 1

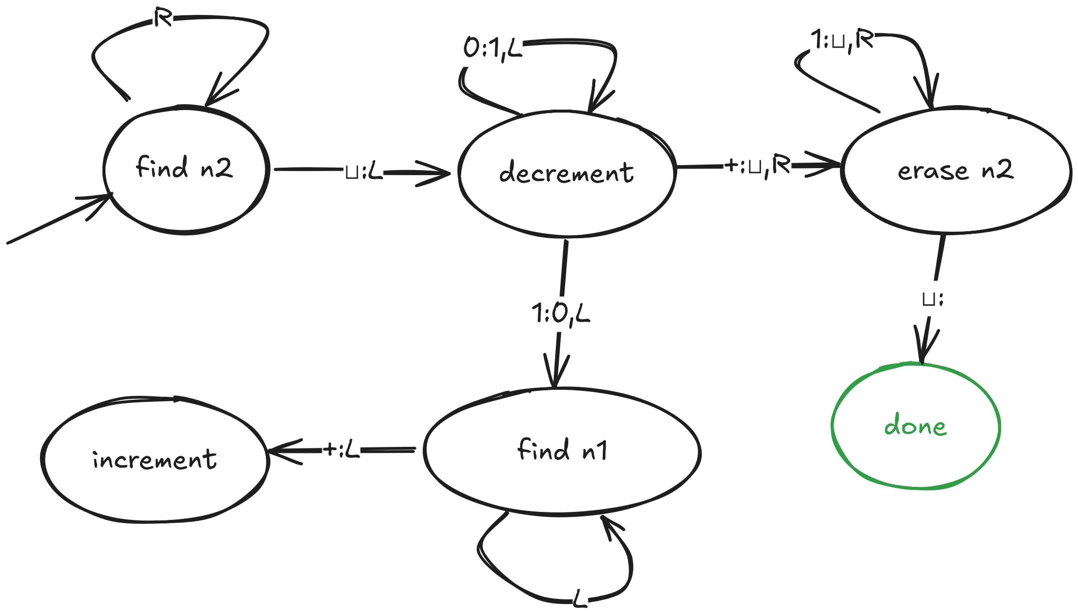
1. Scan right to the end of n_2



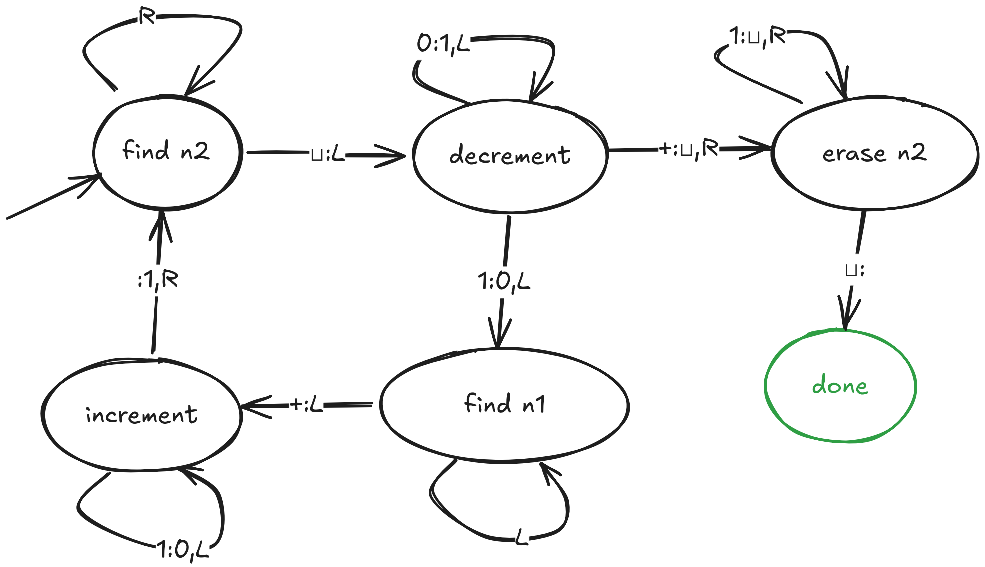
1. Scan right to the end of n_2
2. Decrement n_2
3. If n_2 was all 0s before the decrement (resulting in all 1s after the decrement):
    i. Replace `+111...␣` with `␣␣␣...`
    ii. Accept



1. Scan right to the end of n_2
2. Decrement n_2
3. If n_2 was all 0s before the decrement (resulting in all 1s after the decrement):
    i. Replace `+111...⊔` with `⊔⊔⊔...`
    ii. Accept
4. Scan left to the end of n_1



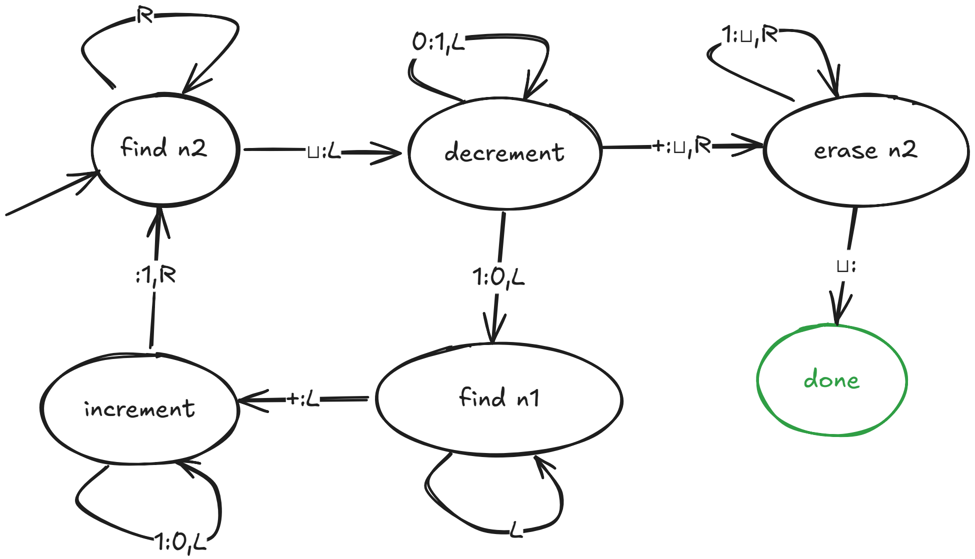
1. Scan right to the end of n_2
2. Decrement n_2
3. If n_2 was all 0s before the decrement (resulting in all 1s after the decrement):
   i. Replace `+111...␣` with `␣␣␣...`
   ii. Accept
4. Scan left to the end of n_1
5. Increment n_1
6. Repeat from step 1



Adder: Trace Example

Input: `101+10`

State	Tape	Action
find n2	⊔ **1** 0 1 + 1 0 ⊔	R
…	…	…
find n2	⊔ 1 0 1 + 1 0 **⊔**	L, decrement
decrement	⊔ 1 0 1 + 1 **0** ⊔	1, L
decrement	⊔ 1 0 1 + **1** 1 ⊔	0, L, find n1
find n1	⊔ 1 0 1 **+** 0 1 ⊔	L, increment
increment	⊔ 1 0 **1** + 0 1 ⊔	0, L
increment	⊔ 1 **0** 0 + 0 1 ⊔	1, R, find n2
find n2	⊔ 1 1 **0** + 0 1 ⊔	R
…	…	…



Implementing TMs in Java

Just as we implemented virtual DFAs and NFAs, we can implement virtual TMs!

Key difference: The tape structure

Java Implementation: Tape Class

```java
public class Tape {
  public Tape(String input) {
    right.push(' ');
    for (int i = input.length() - 1; i >= 0; i--) {
      right.push(input.charAt(i));
    }
    currentSymbol = right.pop();
  }

  public char read() {...}
  public void write(char symbol) {...}

  public void moveLeft() {
    right.push(currentSymbol);
    if (left.isEmpty()) {
      left.push(' ');
    }
    currentSymbol = left.pop();
  }
  public void moveRight() {...}

  private char currentSymbol;
  private final Stack<Character> left = new Stack<>();
  private final Stack<Character> right = new Stack<>();
}
```

Tape: Two-Stack Strategy

Problem: Tape is infinite

Solution: Use two stacks

- **Left stack:** Symbols to the left
- **Right stack:** Symbols to the right
- **Current symbol:** Between them

Example: For input `101`

```
Left: []
Current: 1
Right: [0, 1]

After moveRight():
Left: [1]
Current: 0
Right: [1]
```

Java Implementation: Transition Class

```java
public class Transition {
  public enum Direction { L, R }

  public Transition(State nextState,
                    Character writeSymbol,
                    Direction direction) {
    this.nextState = nextState;
    this.writeSymbol = writeSymbol;
    this.direction = direction;
  }

  public State getNextState() {...}
  public Character getWriteSymbol() {...}
  public Direction getDirection() {...}

  private final State nextState;
  private final Character writeSymbol;
  private final Direction direction;
}
```

Java Implementation: State Class

```java
public class State {
  public void addTransition(Character inputSymbol,
                            Transition transition) {
    transitions.put(inputSymbol, transition);
  }

  public Transition getTransition(Character inputSymbol) {
    return transitions.get(inputSymbol);
  }

  private final Map<Character, Transition> transitions
    = new HashMap<>();
}
```

Java Implementation: TM Class

```
public class TM {
  public TM() {...}

  public void setStartState(State state) {...}
  public void addAcceptState(State state) {...}
  public void addRejectState(State state) {...}

  /** Returns final state (accept/reject) and tape contents */
  public String run(String input) {
    Tape tape = new Tape(input);
    State current = startState;

    while (!isAcceptState(current) && !isRejectState(current)) {
      char symbol = tape.read();
      Transition t = current.getTransition(symbol);

      tape.write(t.getWriteSymbol());
      if (t.getDirection() == Direction.L) {
        tape.moveLeft();
      } else {
        tape.moveRight();
      }
      current = t.getNextState();
    }

    return formatResult(current, tape);
  }
}
```

Universal Turing Machine (UTM)

Key Insight: A TM can be specified as data (a string)

Notation: `<TM>` = string representation of TM

Universal TM: A TM that can simulate any other TM

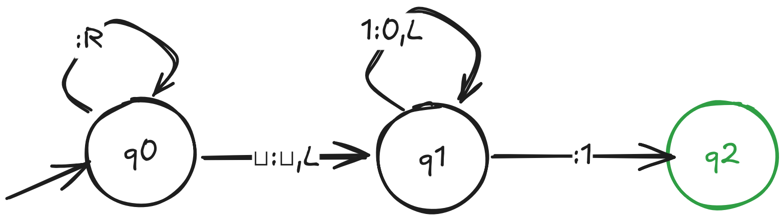
```
public class UTM {
  public UTM(String tmDescription) {...}

  /** Simulates the TM on the input */
  public String simulate(String input) {...}
}
```

Revolutionary Idea: Programs as data!

- **Stored program concept** (von Neumann architecture)
- Leads to **general-purpose computers**
- Turing conceived this before computers existed!

Encoding a TM as a String



Many possible formats

Graphviz:

```
digraph {
    start -> q0;
    q0 -> q0 [label="0:0,R\n1:1,R"];
    q0 -> q1 [label="⊔:⊔,L"];
    q1 -> q1 [label="1:0,L"];
    q1 -> q2 [label="0:1\n⊔:1"];
    q2 -> accept;
}
```

Custom Encoding Format:

```
start q0
accept q2
q0 q0 0:0,R
q0 q0 1:1,R
q0 q1 ⊔:⊔,L
q1 q1 1:0,L
q1 q2 0:1
q1 q2 ⊔:1
```

The exact format doesn't matter - what matters is that a TM CAN be encoded as a string!

Programs Processing Programs

Does this seem strange?

It shouldn't! You encounter it constantly:

- **App stores** - process app programs
- **Compilers** - process high-level programs → assembly
- **Java compiler** (`javac`) - Java → bytecode
- **JVM** (`java`) - bytecode program → execution
- **Interpreters** - Python, JavaScript, etc.

TMs formalized this concept decades before real computers!

TM Variants (All Equivalent)

Many variations of TMs exist:

1. **Multiple tapes**
2. **Nondeterministic TMs**
3. **Two-way infinite tape vs. one-way**
4. **Different halt conditions**

Remarkable Fact: All variants can simulate each other!

Why Turing Machines Matter

Three Fundamental Reasons:

1. **Theoretical Foundation**
    - Precise model of computation
    - Enables mathematical proofs
2. **Universal Model**
    - Church-Turing Thesis (next lecture)
    - As powerful as any physical computer
3. **Practical Impact**
    - Inspired von Neumann architecture
    - Foundation for compiler theory
    - Basis for computability theory

Key Takeaways

✓ TMs add tape read/write and bidirectional movement to DFAs

✓ TMs can recognize non-regular languages like 0^n1^n

✓ TMs can be implemented in Java using two stacks for the tape

✓ Universal TMs can simulate any TM (programs as data!)

✓ All TM variants are equivalent in computational power

✓ TMs are the theoretical model for all computation

Looking Ahead

Next Topics:

1. **Church-Turing Thesis** - TMs = maximal computational power
2. **Decidability** - What can TMs compute?
3. **The Halting Problem** - What CAN'T TMs compute?
4. **Complexity Theory** - What's practical vs. impractical?