Programming Turing Machines
### Turing Machines are Hard

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>×</th>
<th>=</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_s$</td>
<td>B</td>
<td>R</td>
<td>$q_1$</td>
<td>B</td>
</tr>
<tr>
<td>$q_1$</td>
<td>1</td>
<td>R</td>
<td>$q_1$</td>
<td>1</td>
</tr>
<tr>
<td>$q_x$</td>
<td>×</td>
<td>R</td>
<td>$q_1$</td>
<td>×</td>
</tr>
<tr>
<td>$q_=$</td>
<td>=</td>
<td>R</td>
<td>$q_1$</td>
<td>=</td>
</tr>
<tr>
<td>$q_R$</td>
<td>1</td>
<td>L</td>
<td>$q_R$</td>
<td>×</td>
</tr>
<tr>
<td>$q_{s2}$</td>
<td>1</td>
<td>R</td>
<td>$q_{s2}$</td>
<td>×</td>
</tr>
<tr>
<td>$q_{x2}$</td>
<td>1</td>
<td>R</td>
<td>$q_{x2}$</td>
<td>Reject</td>
</tr>
<tr>
<td>$q_{=2}$</td>
<td>1</td>
<td>R</td>
<td>$q_{=2}$</td>
<td>Reject</td>
</tr>
<tr>
<td>$q_{L2}$</td>
<td>1</td>
<td>L</td>
<td>$q_{L2}$</td>
<td>×</td>
</tr>
</tbody>
</table>
Outline for Today

• **A programming language for Turing machines.**

• Design a simple programming language that “compiles” down to Turing machines.

• Keep extending our language to see just how powerful the Turing machine is.
Our Initial Language: \textbf{WB}

- Programming language \textbf{WB} ("Wang B-machine") controls a tape head over a singly-infinite tape, as in a normal Turing machine.

- Language has six commands:
  - Move \textit{direction}  
    - Moves the tape head the specified direction (either left or right)
  - Write \textit{s}  
    - Writes symbol \textit{s} to the tape.
  - Go to \textit{N}  
    - Jumps to instruction number \textit{N} (all instructions are numbered)
  - If reading \textit{s}, go to \textit{N}  
    - If the current tape symbol is \textit{s}, jump to the instruction numbered \textit{N}.
  - Accept and Reject  
    - Ends the program.

- Statements in \textbf{WB} are executed in the order in which they appear, unless control flow changes.
A Simple Program in WB

0: If reading B, go to 4.
1: If reading 1, go to 5.
2: Move right.
3: Go to 0.
4: Accept.
5: Reject.
A **WB** Program for Even Palindromes

- Suppose we want to test if a string is an even-length palindrome.
- Idea: Cross off the first symbol and match it with the symbol on the far side of the tape.
- If it matches, great! Repeat.
- Otherwise, we should reject.
A WB Program for Even Palindromes

// Start
0: If reading 0, go to M0.
1: If reading 1, go to M1.
2: Accept

// M0
3: Write B.
4: Move right.
5: If reading 0, go to 4.
6: If reading 1, go to 4.
7: Move left.
8: If reading 0, go to Next.
9: Reject.

// M1
10: Write B.
11: Move right.
12: If reading 0, go to 11.
13: If reading 1, go to 11.
14: Move left.
15: If reading 1, go to Next.
16: Reject.

// Next
17: Write B.
18: Move left.
19: If reading 0, go to 18
20: If reading 1, go to 18
21: Move right
22: Go to Start.
WB and Turing Machines

- **Recall:** A language $L$ is **recursively enumerable** iff there is a TM for it.

- **Theorem:** A language $L$ is recursively enumerable iff there is a **WB** program for it.

- Need to show the following:
  - Any TM can be converted into an equivalent **WB** program.
  - Any **WB** program can be converted into an equivalent TM.
From Turing Machines to \textbf{WB}

- Basic idea: Construct a small \textbf{WB} program for each state that simulates that state.

- Combine all programs together to get an overall \textbf{WB} program that simulates the Turing machine.
A State in a Turing Machine

• There are three kinds of states in a Turing machine:
  • Accepting states,
  • Rejecting states, and
  • “Working” states.

• We can easily build WB programs for the first two:
  
  ```
  // q_{acc}  // q_{rej}
  0: Accept  0: Reject
  ```
Working States

• At a given working state in a Turing machine, we will do exactly the following, in this order:
  • Read the current symbol.
  • Write back a new symbol based on this choice of symbol.
  • Transition to some destination state.
• Could we build a WB program for this?
# Working States

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₀</td>
<td>B</td>
<td>R</td>
<td>q₁</td>
</tr>
<tr>
<td>q₀</td>
<td>0</td>
<td>L</td>
<td>q₀</td>
</tr>
<tr>
<td>q₀</td>
<td>B</td>
<td>R</td>
<td>q_{acc}</td>
</tr>
</tbody>
</table>

// q₀

0: If reading 0, go to 0q₀.
1: If reading 1, go to 1q₀.
2: If reading B, go to Bq₀.

// 0q₀

3: Write B.
4: Move right.
5: Go to q₁

// 1q₀

6: Write 0
7: Move left.
8: Go to q₀

// Bq₀

9: Write B
10: Move right.
11: Go to q_{acc}
A Complete Construction

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>0</td>
<td>R</td>
<td>( q_1 )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>0</td>
<td>R</td>
<td>( q_{\text{rej}} )</td>
</tr>
</tbody>
</table>

// \( q_0 \)
0: If reading 0, go to 3.
1: If reading 1, go to 6.
2: If reading B, go to 9.
3: Write 0.
4: Move right.
5: Go to \( q_1 \).
6: Write 1.
7: Move right.
8: Go to \( q_{\text{rej}} \).
9: Write 1.
10: Move right.
11: Go to \( q_{\text{acc}} \).

// \( q_1 \)
13: If reading 0, go to 16.
14: If reading 1, go to 19.
15: If reading B, go to 22.
16: Write 0.
17: Move right.
18: Go to \( q_{\text{rej}} \).
19: Write 1.
20: Move right.
21: Go to \( q_0 \).
22: Write 1.
23: Move right.
24: Go to \( q_{\text{acc}} \).

// \( q_{\text{acc}} \)
12: Accept.

// \( q_{\text{rej}} \)
25: Reject.
From \textbf{WB} to Turing Machines

- We now need a way to convert a \textbf{WB} program into a Turing machine.

- Construction sketch:
  - Create a state in the TM for each line of the \textbf{WB} program.
  - Introduce extra “helper” states to implement some of the trickier instructions.
  - Connect the states by transitions that simulate the \textbf{WB} program.

- We will show how to translate each \textbf{WB} command into a collection of states plus transitions.
Refresher: Turing Machine Notation

B → B, R

0 → 0, R
1 → 1, R

0 → B, L

1 → 1, R

0 → 0, L
1 → 1, L

start

q_0

q_1

q_2

q_acc

q_rej

q_3

q_4

q_5

B → B, R

B → B, R

B → B, R

B → B, R

B → B, L

B → B, R

B → B, R

B → B, R

B → B, R

B → B, L

B → B, L

B → B, R

B → B, L
Refresher: Turing Machine Notation

- The accept and reject states are denoted

  \[ q_{\text{acc}} \quad q_{\text{rej}} \]

- A transition of the form

  \[ q_{a} \xrightarrow{x \rightarrow y, D} q_{b} \]

  means “on seeing \( x \), write \( y \) and move direction \( D \).”
Accept and Reject

- The **Accept** and **Reject** commands are the easiest to translate.

- To translate **N: Accept** into TM states, construct the following:

  \[
  q_n \rightarrow \Gamma \rightarrow \Gamma, R \rightarrow q_{acc}
  \]

- To translate **N: Reject** into TM states, construct the following:

  \[
  q_n \rightarrow \Gamma \rightarrow \Gamma, R \rightarrow q_{rej}
  \]
Move left and Move right

- We can translate **N: Move left** and **N: Move right** by having the TM do the following:
  - Write back the same symbol that was already on the tape (ensuring that we don't change the tape).
  - Move in the indicated direction.
  - Transition into the state representing line \(N + 1\).
Go to $L$

- The line $N$: Go to $M$ needs to change into the state for line $M$ without moving the tape head.
- All TM transitions move the tape head; how might we address this?
- Move right and change into a new state that then moves back to the left.

\[ q_n \xrightarrow{\Gamma \rightarrow \Gamma, R} q_{\text{temp}} \xrightarrow{\Gamma \rightarrow \Gamma, L} q_L \]
Write $S$

- The line $N$: Write $S$ needs to
  - Write the symbol $s$,
  - Leave the tape head where it is, and
  - Move to line $N + 1$.
- We use a similar trick as before:
If reading $s$, go to M

- The line $N$: If reading $s$, go to M either
  - Executes a “go to M” step as before if reading $s$, or
  - Does nothing and transitions to state $N + 1$.

\[
q_n \xrightarrow{s \to s, R} q_{\text{temp}} \xrightarrow{\Gamma \to \Gamma, L} q_m
\]

\[
\Gamma - s \to \Gamma - s, R
\]

\[
q_{\text{temp}2} \xrightarrow{\Gamma \to \Gamma, L} q_{n+1}
\]
A Complete Conversion

0: If reading B, go to 4.
1: If reading 1, go to 5.
2: Move right.
3: Go to 0.
4: Accept.
5: Reject.

0 → 0, R
1 → 1, R
Γ → Γ, L
B → B, R
Γ → Γ, L
Γ → Γ, R
Γ → Γ, L
Γ → Γ, R
Γ → Γ, R
Γ → Γ, R
Γ → Γ, R
start

0

1

2

3

4

5

a

r

0 → 0, R
1 → 1, R
Γ → Γ, L
B → B, R
Γ → Γ, L
Γ → Γ, R
The Story So Far

- We have just built a simple programming language that is equivalent in power to a Turing machine.
- This language, however, makes for some very complicated programs.
- Let's add some new features to our programming language to make it a bit easier to work with.
Revisiting Even Palindromes

// M0
3: Write B.
4: Move right.
5: If reading 0, go to 4.
6: If reading 1, go to 4.
7: Move left.
8: If reading 0, go to Next.
9: Reject.

// Next
17: Write B.
18: Move left.
19: If reading 0, go to 18
20: If reading 1, go to 18
21: Move right.
22: Go to Start.

• Steps 4 – 6 essentially say “move right, then move right until you read a blank.”

• Steps 18 – 20 essentially say “move left, then move left until you read a blank.”

• Is it really necessary to write this out each time?
Introducing WB2

• The programming language **WB2** is the language **WB** with two new commands:
  - **Move left until** \{s_1, s_2, ..., s_n\}.
    - Moves the tape head left until we read one of \(s_1, s_2, s_3, ..., s_n\).
  - **Move right until** \{s_1, s_2, ..., s_n\}.
    - Moves the tape head right until we read one of \(s_1, s_2, s_3, ..., s_n\).
• Both commands are no-ops if we're already reading one of the specified symbols.
• We can write programs in **WB2** that are much easier to read than in **WB**.
A **WB** Program for Even Palindromes

// Start
0: If reading 0, go to M0.
1: If reading 1, go to M1.
2: Accept

// M0
3: Write B.
4: Move right.
5: If reading 0, go to 4.
6: If reading 1, go to 4.
7: Move left.
8: If reading 0, go to Next.
9: Reject.

// M1
10: Write B.
11: Move right.
12: If reading 0, go to 11.
13: If reading 1, go to 11.
14: Move left.
15: If reading 1, go to Next.
16: Reject.

// Next
17: Write B.
18: Move left.
19: If reading 0, go to 18
20: If reading 1, go to 18
21: Move right
22: Go to Start.
A **WB2** Program for Even Palindromes

// **Start**
0: If reading 0, go to M0.
1: If reading 1, go to M1.
2: Accept

// **M0**
3: Write B.
4: Move right.
5: Move right until {B}.
6: Move left.
7: If reading 0, go to Next.
8: Reject.

// **M1**
9: Write B.
10: Move right.
11: Move right until {B}.
12: Move left.
13: If reading 1, go to Next.
14: Reject.

// **Next**
15: Write B.
16: Move left.
17: Move left until {B}.
18: Move right.
19: Go to Start.
A **WB2** Program for *BALANCE*

- Let $\Sigma = \{0, 1\}$ and consider the language *BALANCE*:
  \[
  \{ \, w \in \Sigma^* \mid w \text{ has the same number of } 0\text{s and } 1\text{s.} \, \}
  \]
- Let's write a **WB2** program for *BALANCE*. 
A WB2 Program for **BALANCE**

// Start
0: Move right until {0, 1, B}.  
1: If reading 0, go to Match0.  
2: If reading 1, go to Match1.  
3: Accept.

// Match0
4: Write B.  
5: Move right.  
6: Move right until {1, B}.  
7: If reading 1, go to Found.  
8: Reject.

// Match1
9: Write B.  
10: Move right.  
11: Move right until {0, B}.  
12: If reading 0, go to Found.  
13: Reject.

// Found
14: Write x.  
15: Move left until {B}.  
16: Move right.  
17: Go to Start.
Theorem: A language is recursively enumerable iff there is a **WB2** program for it.

We could directly prove this again by showing equivalence with Turing machines.

Instead, we'll connect it to **WB**: 

![Diagram](image_url)
From **WB2** to **WB**

- We will show how to turn any **WB2** program into an equivalent **WB** program.
- All old instructions are still valid.
- We need to show how to implement the new `Move ... until` commands using just **WB**.
Implementing \textbf{Move} \ldots until

- Replace $N$: \textbf{Move} $\textit{dir}$ until $\{s_1, \ldots, s_n\}$ as follows:
  
  $N+0$: If reading $s_1$, go to $N+n+2$.
  
  $N+1$: If reading $s_2$, go to $N+n+2$.
  
  $N+2$: If reading $s_3$, go to $N+n+2$.
  
  \ldots
  
  $N+(n-1)$: If reading $s_n$, go to $N+n+2$.
  
  $N+n$: \textbf{Move} $\textit{dir}$.
  
  $N+n+1$: Go to $N$

- Renumber other lines as appropriate.
Why This Matters

- We are starting to move more and more away from the Turing machine with from we started.
- The structure of our approach is
  - Find some simple programming language that can be directly translated into a Turing machine (and vice-versa).
  - Add new features to the language, and show how to implement those new features using the old language.
  - Add new features to \textit{that} language, and show how to implement those features using the previous language.
  - \textit{(etc.)}
  - Conclude that the final language is equivalent to a Turing machine.
A Repeating Pattern

// Match0
4: Write B.
5: Move right.
6: Move right until \{1, B\}.
7: If reading 1, go to Found.
8: Reject.

// Match1
9: Write B.
10: Move right.
11: Move right until \{0, B\}.
12: If reading 0, go to Found.
13: Reject.
A Simple Memory

• Right now, our programming language **WB2** has no variables in it.

• To solve larger classes of problems, let's invent a new language **WB3** that has support for variables.

• We will severely limit the scope of our variables:
  • Only **finitely many** total variables throughout the program.
  • Each variable can only hold a single tape symbol.
  • Each variable initially holds the blank symbol.
Our New Commands

• We will define **WB3** as **WB2** with the following extra commands:
  
  • **Load** $s$ into $v$.
    - Sets the variable $v$ equal to tape symbol $s$.
  
  • **Load current into** $v$.
    - Sets the variable $v$ equal to the currently-scanned tape symbol.
  
  • **If** $v_1 = v_2$, go to $L$.
    - If $v_1$ and $v_2$ have the same value, go to instruction $L$.
    - These may be constants or variables.

• Additionally, any command that referenced a tape symbol (for example, **write, if reading, move ... until**) can refer to variables in addition to constants.
A WB2 Program for Even Palindromes

// Start
0: If reading 0, go to M0.
1: If reading 1, go to M1.
2: Accept

// M0
3: Write B.
4: Move right.
5: Move right until {B}.
6: Move left.
7: If reading 0, go to Next.
8: Reject.

// M1
9: Write B.
10: Move right.
11: Move right until {B}.
12: Move left.
13: If reading 1, go to Next.
14: Reject.

// Next
15: Write B.
16: Move left.
17: Move left until {B}.
18: Move right.
19: Go to Start.
A **WB3** Program for Even Palindromes

// Start

0: Read current into X.
1: If X = B, go to Acc.
2: Write B.
3: Move right.
4: Move right until {B}.
5: Move left.
6: If reading X, go to Match.
7: Reject.

// Match

8: Write B.
9: Move left.
10: Move left until B.
11: Move right.
12: Go to Start.

// Acc:

13: Accept.
A WB2 Program for BALANCE

// Start
0: Move right until \{0, 1, B\}.  // Match1
1: If reading 0, go to Match0.
2: If reading 1, go to Match1.
3: Accept.

// Match0
4: Write B.
5: Move right.
6: Move right until \{1, B\}.
7: If reading 1, go to Found.
8: Reject.

// Match1
9: Write B.
10: Move right.
11: Move right until \{0, B\}.
12: If reading 0, go to Found.
13: Reject.

// Found
14: Write x.
15: Move left until \{B\}.
16: Move right.
17: Go to Start.
A **WB3** Program for *BALANCE*

// **Start**

0: Move right until \{0, 1, B\}. 8: Write B.
1: If reading B, go to Acc. 9: Move right.
2: If reading 0, go to 5. 10: Move right until \{Y, B\}
3: Load 0 into Y. 11: If reading Y, go to 13.
4: Go to Scan. 12: Reject.
5: Load 1 into Y. 13: Write x.
6: Go to Scan. 14: Move left until B.
7: 

// **Scan**

14: Move left until B.
15: Move right.
16: Go to Start.

// **Acc:**

17: Accept.
Equivalence of **WB2** and **WB3**

- **Theorem:** A language is recursively enumerable iff there is a **WB3** program for it.
- Adding in these sorts of variables adds *no power* to our model of computation!
- To prove the theorem, we will show
  - Any WB2 program can be converted to a WB3 program, and
  - Any WB3 program can be converted to a WB2 program.
The Proof: An Intuition

- Our programs allow only finitely many variables holding only one of finitely many different values (tape symbols).

- We could just replicate the program for each possible assignment to the variables, then hardcode in the behavior in each of these cases.

- Could make the program staggeringly huge, but it will still be finite!
The Transformation, Part I

- Let $V_1, V_2, \ldots, V_n$ be the variables referenced in the program.
  - We can just look at the source code to determine this.
- Make $|\Gamma|^n$ copies of the initial program, one for each possible assignment of tape symbols to the variables $V_i$.
- Order the copies arbitrarily, but make the version where all variables hold B come first.
The Transformation, Part II

• We now have a whole bunch of copies of \texttt{WB3} programs.
• We need to convert them into legal \texttt{WB2} programs.
• This works in two steps:
  • Removing variables from older \texttt{WB2} commands like \texttt{Write}, \texttt{If reading ...}, and \texttt{Move ... while}.
    – For example: “\texttt{Write X},” where \texttt{X} is a variable.
  • Rewriting all new \texttt{WB3} commands that reference variables to use only \texttt{WB2} commands.
    – For example: “\texttt{Load current into X}.”
Eliminating Variables from WB2

- Removing variables from purely WB2 statements is easy because we've copied the program so many times.
- For each copy, replace all variables in WB2 statements with the value that the variable has in that copy.

```
0: Load 0 into Y.
1: Write Y.
2: Accept

0: Load 0 into Y.
3: Load 0 into Y.
4: Write 0.
5: Accept

6: Load 0 into Y.
7: Write 1.
8: Accept

Y = B
Y = 0
Y = 1
```
Eliminating Variables from **WB3**

- We can eliminate commands that manipulate variables by replacing them with \texttt{Go to s}.
- There are three commands to eliminate:
  - Load \texttt{s} into \texttt{v}.
  - Load current into \texttt{v}.
  - If \( v_1 = v_2 \), go to \texttt{L}. 
If $v_1 = v_2$, go to $L$

- We can eliminate this statement by just hardcoding the jump in place.
- If in the current copy of the program $v_1$ and $v_2$ have the same values, replace with
  
  Go to $L$

  where $L$ is the corresponding version of $L$ in this copy.
- Otherwise, replace with
  
  Go to $N$

  where $N$ is the number of the next line in the program.
Load $s$ into $v$

- To simulate the effect of loading $s$ into $v$, we can jump out of the current copy of the program into the copy where $v$ has value $s$.

0: Load 0 into $Y$.
1: Write $Y$.
2: Accept

0: Go to 4.
1: Write $B$.
2: Accept

3: Go to 4.
4: Write 0.
5: Accept

6: Go to 4.
7: Write 1.
8: Accept

$Y = B$
$Y = 0$
$Y = 1$
Load current into $v$

- We can simulate this instruction using a similar trick to before.
- Replace this instruction as follows:

  If reading $s_1$, go to LoadS$_1$.
  If reading $s_2$, go to LoadS$_2$.
  ...
  If reading $s_n$, go to LoadS$_n$.

  // LoadS$_1$:
  Load $s_1$ into $v$.
  Go to Done.
  ...

  // LoadS$_n$
  Load $s_n$ into $v$.
  Go to Done.

  // Done:
Souping up our Tape

- Up to this point, we've been improving our WB programming language by adding in new ways of scanning over the tape.
- What if we made changes to the tape itself?
A Multitrack Tape

// Start

0: Read track 1 into X.
1: Move right.
2: Write X into track 2
3: If reading B on track 1, go to 5.
4: Go to 0
5: /* ... */
Introducing WB4

- Let's define **WB4** to be **WB3** with the introduction of finitely many *tracks* on the tape.
- The tape head still moves as a unit to the left or right, but we can now issue read and write commands to any cell in the current track.
- All previous commands updated to specify which track is to be read or written.
A Surprising Theorem

- **Theorem:** A language is recursively enumerable iff there is a **WB4** program for it.
- This is not obvious... it seems like adding in more tracks should increase the power of our programming language!
- As with before, will prove that all **WB4** programs are equivalent to **WB3** programs.
The Intuition

- Treat a single tape as a “fat tape” where each tape symbol encodes the contents of the cells of all four tracks.
- Each read or write to a specific location replaces the entire tape cell with a new symbol representing the change.
A Sketch of the Construction

• Replace each instruction that reads or writes a track with a huge cascading “if” that checks for every possible tape symbol and reacts accordingly.

• Can make the program enormously bigger, but it still ends up finite.

• I'm not even going to attempt to fit something like that onto these slides.
Where We Are Now

- Starting with **WB**, we have added
  - Loops to search for a value. (**WB2**)
  - Variables with finite storage. (**WB3**)
  - Multiple tracks. (**WB4**)
- Yet we still accept exactly the same set of languages.
- Every **WBn** program can be converted back to a TM.
Making Things Crazier

- What do you get when you combine a PDA and a **WB4** program?
- A program with an infinite tape, plus multiple stacks!

```
1 1 0 0 1 ...
1 1 0 0 ...
0 0 1 1
```

[Diagram showing tape and stacks]
Introducing WB5

- The programming language **WB5** is the programming language **WB4** with the addition of a finite number of stacks.

- We add three extra commands:
  - **Push s onto stack v.**
    - Pushes the symbol **s** onto the stack named **v**.
  - **If stack v is empty, go to L.**
    - If stack **v** is empty, go to instruction **L**.
  - **Pop stack v into w.**
    - If stack **v** is nonempty, pops **v** and puts the top into **w**.
The Multiplication Language

- Let \( \Sigma = \{ 0, 1, 2 \} \) and consider the language \( 01MULT \) defined as

  \[
  \{ w \in \Sigma^* \mid \text{the number of 2's in } w \text{ is the product of the number of 1's and the number of 0's. } \}
  \]

- For example:
  - \( 00112222 \in 01MULT \)
  - \( 22001122122 \in 01MULT \)
- This language is neither context-free nor regular.
- How could we write a \textbf{WB5} program for it?
WB5 Program for 01MULTI

// Start
0: If reading 0, go to Load0.
1: If reading 1, go to Load1.
2: If reading 2, go to Load2.
3: Go to Check.

// Load0
4: Push 0 onto Stack 0.
5: Move right.
6: Go to Start.

// Load1
7: Push 1 onto Stack 1.
8: Move right.
9: Go to Start.

// Load2
10: Push 2 onto Stack 2.
11: Move right.
12: Go to Start.
WB5 Program for 01MULTI

// Check:
13: If Stack 0 is empty, go to Ver.
14: Pop Stack 0.
15: If Stack 1 is empty, go to Fix.
16: Pop Stack 1.
17: Push 1 onto Stack 1T.
18: If Stack 2 is empty, go to Rej.
19: Pop Stack 2.
20: Go to 15.

// Fix:
22: If St 1T is empty, go to Check.
23: Pop Stack 1T.
24: Push 1 onto Stack 1.
25: Go to Fix.

// Ver:
26: If Stack 2 is empty, go to Acc.
27: Reject.

// Rej:
21: Reject.

// Acc:
28: Accept.
A Pretty Ridiculous Theorem

- **Theorem:** A language is recursively enumerable iff there is a WB\textsuperscript{5} program for it.
- So adding in finitely many infinite stacks doesn't give us any more expressive power!
- As with before, will prove that all WB\textsuperscript{5} programs are equivalent to WB\textsuperscript{4} programs.
From Stacks to Tracks

- The key idea behind the construction for converting \textbf{WB5} programs into \textbf{WB4} programs is to represent each stack with its own track.

- If there are $n$ stacks in the program, we will add $n + 1$ tracks:
  - One track for each of the $n$ stacks, and
  - One track for bookkeeping.

- If the \textbf{WB5} program was using any tracks, we'll keep them as well and add these new ones in separately.
0: Push 1 onto Stack 3.

0: Write × on track 5.
1: Move left until {>} on track 4.
2: Move right until {<} on track 4.
3: Write 1 on track 4.
4: Move right.
5: Write < on track 4
6: Move left until {>} on track 4.
7: Move right until {×} on track 5.
8: Write B on track 5.
1: If Stack 1 is empty, go to L

0: Write $\times$ on track 5.
1: Move left until $\{>\}$ on track 2.
2: Move right.
3: Load current on track 2 into $V$
4: Move left.
5: Move right until $\{\times\}$ on track 5.
6: Write $B$ on track 5.
7: If $V = <$, go to L.
2: Pop Stack 2 into X.

0: Write × on track 5.
1: Move left until {>} on track 3.
2: Move right until {<} on track 3.
3: Move left.
4: Load current on track 3 into X.
5: If X = >, go to 7.
6: Write < on track 3
7: Move left until {>} on track 3.
8: Move right until {×} on track 5.
9: Write B on track 5.
Completing the Construction

- We've seen how to convert the new **WB5** stack commands into **WB4** code.
- For this to work, the extra tracks must be set up correctly.
- Add preamble code to the generated **WB4** program to do this:
  
  Write > to track 2.
  
  ...
  
  Write > to track n.
  
  Move right.
  
  Write < to track 2.
  
  ...
  
  Write < to track n.
  
  Move left.
But Why Stop There?

- Adding finitely many stacks to \textbf{WB} doesn't increase its expressive power.
- What if we added finitely many \textbf{tapes} to \textbf{WB}?
- We now have a programming language controlling
  - Multiple tracks per tape,
  - Finitely many stacks, and
  - Finitely many tapes.
Introducing **WB6**

- The programming language **WB6** is **WB5** with the addition of multiple tapes.
- All tape commands have been updated to specify which tape they apply to.
- If tape unspecified, it's assumed that it's tape 1.
A **WB6** Program for *SEARCH*

- Recall from Problem Sets 5 and 6 that the language *SEARCH* over $\Sigma = \{0, 1, ?\}$ is the language
  $$\{ p?t \mid p, t \in \{0, 1\}^* \text{ and } p \text{ is a substring of } t \}$$

- How would we write a **WB6** program for *SEARCH*?

- (For simplicity, we'll assume that the input is properly formatted).
A **WB6** Program for **SEARCH**

```
// Start

0: Move tape 2 right.
1: If reading ? on tape 1.1, go to Match.
2: Load curr on tape 1.1 into X.
3: Write X to tape 2.
4: Move tape 1 right.
5: Move tape 2 right.
6: Go to 1.
```
A WB6 Program for SEARCH

// Match
7: Move tape 2 left until {B}
8: Move tape 2 right.
9: Move tape 1 right.
10: Write $ to tape 1, track 2.
11: If B on tape 2, go to Acc.
12: If B on tape 1, go to Rej.
13: Load tape 1, track 1 into X.
14: Load tape 2 into Y.
15: If X = Y, go to 17.
16: Go to Mismatch.
17: Move tape 1 right.
18: Move tape 2 right.
19: Go to 11.

// Mismatch
20: Move tape 1 left until {$}
21: Go to Match.
22: Accept.
23: Reject.

// Acc

// Rej
Oh, Come On Already...

• **Theorem:** A language is recursively enumerable iff there is a **WB6** program for it.

• We can really supercharge these languages without increasing our power!

• As with before, the construction will convert **WB6** programs into **WB5** programs.
The Key Idea

- Represent an infinite tape with two stacks.
A Sketch of the Construction

- At the start of the program, copy the contents of the initial tape into a pair of stacks that will henceforth represent the first tape.
- Convert all motion operations into stack manipulation operations to push and pop values from the appropriate stacks.
- Use variables to hold temporary values (for example, when moving the top of one stack to another).
0: Move tape 1 right.

0: If stack 1R is empty, go to 2.
1: Go to 3.
2: Push B onto stack 1R.
3: Pop stack 1R into X.
4: Push X onto stack 1L.
What Else Can We Add?

- Function call and return.
  - Have a stack to use as the call stack.
  - Calling a function pushes the index of the instruction to which it should return.
  - Returning pops the stack and jumps back.
- Named variables.
  - Have a tape storing a sequence of values of the form name: value.
  - Can read and write values from the tape.
- Pointers
  - Have variables hold the names of other variables.
- Primitive types and arithmetic.
  - Design subroutines for addition, subtraction, etc.
  - Apply them to named variables.
- **Pretty much any feature of any major programming language.**
The Conversion Back Down

• From **WB6** to **WB5**:  
  • Add in two stacks per tape used.  
  • Replace all tape operations with appropriate stack manipulations.

• From **WB5** to **WB4**:  
  • Add in one track per stack, plus one extra track.  
  • Replace all stack operations with appropriate manipulations of those tracks.
The Conversion Back Down

- **From WB4 to WB3:**
  - Expand the tape alphabet to include symbols for all track combinations.
  - Replace all references to track symbols with cascading if's for each possible case.

- **From WB3 to WB2:**
  - Replicate the code once for each possible assignment to variables.
  - Hardcode in statements referencing variables.
  - Replace variable manipulation code with code to jump to the appropriate copy.
The Conversion Back Down

• From **WB2** to **WB**:
  • Expand out *move ... until* statements by replacing them with cascading *if* statements.

• From **WB** to Turing machines:
  • Replace each statement with the appropriate Turing machine gadget.
The Conversion Back Down

- The total conversion of a WB6 program using variables, multiple tracks, multiple stacks, and multiple tapes might produce an enormous Turing machine!
- But that said, the result is still a Turing machine.
- **Turing machines are simple, yet have enormous computational power.**
Just how powerful are Turing machines?
Effective Computation

- An **effective method of computation** is a form of computation with the following properties:
  - The computation consists of a set of steps.
  - There are fixed rules governing how one step leads to the next.
  - Any computation that yields an answer does so in finitely many steps.
  - Any computation that yields an answer always yields the correct answer.
The **Church-Turing Thesis** states that

Every effective method of computation is either equivalent to or weaker than a Turing machine.

This statement cannot be proven or disproven, but is widely considered true.
Regular Languages

DCFLs

CFLs

All Languages

RE
Next Time

• **Encodings**
  • How do we do computations over arbitrary objects?

• **The Universal Turing Machine**
  • A Turing machine for running other Turing machines.

• **Nondeterministic Turing Machines**
  • What happens when we supercharge a TM? What does this even mean?

• **R and RE Languages**
  • A finer gradation within the RE languages.