Turing Machines, Part 2

CMPU 240 • Language Theory and Computation • Spring 2019

Now:

Assignment 5 graded
Optional programming assignment 1 (of 3)

Tonight:

Assignment 6

Soon:

Optional programming assignment 2 (of 3)
Exam 2 graded

On Assignment 5, Problem 5

An alarming number of people are making mistakes that show they don’t understand

1. what a language is,
2. how set operations like intersection work, or
3. both

Let’s review.
A **string** is a sequence of characters/symbols.

In a programming language, a string is written with quotation marks, e.g., "I already know this".

Punctuation goes outside the quotation marks because we're not animals.

In language theory, we omit the quotation marks and use visible characters for spaces, e.g., No, you don't.

The empty string, ε, is a string of length 0.

A **set** is an unordered collection of 0 or more objects of any type, e.g.,

- \( \emptyset \)
- \( \{\} \)
- \( \{0\} \)
- \( \{0, 1\} \)
- \( \mathbb{N} \)
- \( \mathbb{N}_0 \)

A **language** is a set of strings, e.g.,

- \( \emptyset \)
- \( \{\varepsilon\} \)
- \( \{a\} \)
- \( \{a, b\} \)

A language can be **finite**, i.e., only contain a fixed number of strings, even if that number is large.

Every finite language is regular.

It is also therefore, context-free because the context-free languages are more powerful.

A language can be **infinite**, i.e., contain an infinite number of strings.
If a language contains a very large number of strings – whether it is finite or infinite – we don’t write them all out.

We can use an ellipsis, e.g.,

\{a, aa, aaa, \ldots\}

but this is not very precise. E.g., we might mean the language consisting of strings composed of

one or more \textit{as}, or

either one \textit{a} or a prime number of \textit{as}.

Instead, we usually give a description of the set, e.g.,

\{w \mid w \in \{a, b\}^*\}

This means “Any string \(w\) such that \(w\) is an element of the Kleene closure of the set \(\{a, b\}\)”.

Or, more simply, “0 or more \textit{as} or \textit{bs} in any order”. 

\textbf{Note:} Kleene star (*) is an \textit{operator} that applies to languages (sets).

Sometimes we’ve written \{a^n b^n\} as a shorthand when discussing languages, but this is not a complete specification.

If you want to describe a language, you need to give the possible values for the variables:

\{a^n b^n \mid n \geq 0\}

or, to be more precise,

\{a^n b^n \mid n \in \mathbb{N}_0\}

since we can’t have non-integer numbers of a character in a string.

When we write \{… | …\}, this is \textit{set-builder notation}. It can involve variables like \textit{s}, \textit{w}, \textit{x}, or \textit{y} and characters like \textit{a}, \textit{b}, \textit{c}, …

You can tell these apart because we use letters from the end of the alphabet as variables for strings and substrings; we use letters from the beginning of the alphabet or numbers for characters.

To further distinguish them, both the textbook and I use different fonts for them.

Variables are in an \textit{italic font}.

Characters are in a \textit{fixed-width font}.
$L = \{a^i b^j c^k \mid i, j \geq 0\}$

This is a description of a language, $L$.

The language is a set containing all strings that are composed of an equal number of (0 or more) $a$s and $b$s followed by 0 or more $c$s. E.g.,

- $\varepsilon \in L$
- $ab \in L$
- $c \in L$
- $abcc \in L$
- $aabb \in L$
- $aaabbb \in L$
- $aaabbbccc \in L$

- $abb \notin L$
- $aab \notin L$
- $ac \notin L$
- $bc \notin L$
- $ba \notin L$
- $ababc \notin L$

If a language is regular, we might give the description as a regular expression, which is written without curly brackets, e.g.,

- $a^*$

This is not a string. This is a description of a language. I.e., it is a description of a set of strings:

$\{\varepsilon, a, aa, aaa, aaaa, aaaaa, \ldots\}$

Regular expressions are the only time we remove the curly brackets from the description of a language.

The regular expression $a$ means the language $\{a\}$.

The regular expression $a^*$ means that language $\{a\}^* = \{a^i \mid i \geq 0\}$

The regular expression $aub$ means the language $\{a\} \cup \{b\} = \{a, b\}$

Union and intersection are set operations. They can be applied to languages because languages are sets of strings.

They do not apply to strings. Never, ever, ever.
The intersection of two sets is a set containing the elements that are in both sets.

Therefore, the intersection of two languages is the language of strings that are in both of the original languages.

\[ \{a^i \mid i \geq 0\} \cap \{b^i \mid i \geq 0\} = \{\varepsilon\} \]

The only string in both languages is \( \varepsilon \)

\[ a^0 = \varepsilon \]
\[ b^0 = \varepsilon \]

\[ \{a^i \mid i \geq 1\} \cap \{b^i \mid i \geq 1\} = \emptyset \]

The languages (sets) are completely disjoint.

"But wait, I thought we do magic and make parts of strings go away!"

No magic. You never directly remove part of a string using operators like intersection or set difference (−). You never directly add to a string using union.

Rather, you can make a new language (set) by:

Putting all the strings from two languages into one new language (union).

Only including strings from language 1 that aren’t in language 2 (set difference).

Only including strings that are in both languages (intersection).

What is \( \{a^n b^n c^n \mid n \geq 0\} \cap \{a^i b^j c^i \mid i, j \geq 0\} \)?

\[ \{a^n b^n c^n \mid n \geq 0\} \]

These are the strings in both languages; the first language is a proper subset of the second.

If in doubt, draw a Venn/Euler diagram and consider what strings are in each part of it.
What is $\{a^n b^n c^n \mid n \geq 0\} \cap \{a^i b^i c^j d^j \mid i, j \geq 0\}$?

$\{\varepsilon\}$.

Strings in the first language can have
- as, bs, and cs
- no characters

Strings in the second languages can have
- as, bs, cs, and ds
- just as and bs
- just cs and ds
- no characters

What is $\{a^n b^n c^n \mid n \geq 1\} \cap \{a^i b^i c^j d^j \mid i, j \geq 1\}$?

$\emptyset$

Review from last time
To picture a Turing machine, imagine a mathematician performing calculations on a scroll of paper.

So that we don’t need to worry about running out of places to write things down, imagine the scroll is infinitely long.

The mathematician will be able to solve any computational problem that’s solvable, no matter how many operations are involved – though it might take him a very long time!

Alan Turing showed that any calculation that can be performed by a smart mathematician can also be performed by a stupid but meticulous clerk who follows a simple set of rules for reading and writing the information on the scroll.

If you’re familiar with Searle’s Chinese room argument, this may sound familiar, but we’ll leave that philosophical trap for CMPU 365 and the Cognitive Science department.

In fact, Turing showed that the human clerk can be replaced by a finite state machine.

The finite state machine looks at only one symbol on the scroll at a time, so it’s best thought of as a narrow paper tape, with a single symbol on each line.

Today we call the combination of a finite-state machine with an infinitely long tape a Turing machine.
This is the Turing machine's finite state control. It issues commands that drive the operation of the machine.

The machine is started with the input string written somewhere on the tape. The tape head initially points to the first symbol of the input string.

A physical model of a Turing machine by Mike Davey
youtube.com/watch?v=E3keLeMwflHY

This is the TM's infinite tape. Each tape cell holds a tape symbol. Initially all tape symbols are blank.
Our first Turing machine

Like other automata, TMs begin execution in their start state.

At each step, the TM only looks at the symbol immediately under the tape head.

These two transitions originate at the current state. We’re going to choose one of them to follow.

Each transition has the form \( \langle \text{read} \rangle \rightarrow \langle \text{write} \rangle, \langle \text{direction} \rangle \) and means “if symbol read is under the tape head, replace it with write and move the tape head in direction (left or right)”. The ☐ symbol denotes a blank cell.
Our first Turing machine

Each transition has the form
\( \langle \text{read} \rangle \rightarrow \langle \text{write} \rangle, \langle \text{direction} \rangle \)
and means “if symbol read is under the tape head, replace it with write and move the tape head in direction (left or right)”. The □ symbol denotes a blank cell.
Unlike a DFA or NFA, a TM doesn’t stop after reading all the input characters. We keep running until the machine explicitly says to stop.

This special state is an accepting state.
When a TM enters an accepting state, it immediately stops running and accepts whatever the original input string was.

This special state is a rejecting state.
When a TM enters a rejecting state, it immediately stops running and rejects whatever the original input string was.

A Turing machine has two alphabets:

An input alphabet, $\Sigma$. All input strings are written in the input alphabet.

A tape alphabet, $\Gamma$, where $\Sigma \subseteq \Gamma$. The tape alphabet contains all symbols that can be written onto the tape.

The tape alphabet $\Gamma$ can contain any number of symbols but always includes at least one blank symbol, $\square \notin \Sigma$.

The Turing machine begins with an infinite tape of $\square$ symbols with the input written beginning in the cell under the tape head.
We designed a Turing machine to recognize the context-free language

\[ L = \{0^n 1^n \mid n \in \mathbb{N}_0\} \]

by working its way from the outside in, erasing a 0 from the beginning of the string and a 1 from the end.

This is a \textit{recursive solution}, which turns an instance of the problem into a smaller instance of the same problem until it reaches a base case:

- empty string (accept)
- just 0s (reject)
- just 1s (reject).

Remember that all missing transitions implicitly reject.
Another Turing machine design

On Thursday, we began considering how we could design a Turing machine for a related context-free language over $\Sigma = \{0, 1\}$:

$L = \{w \in \Sigma^* \mid w \text{ has the same number of 0s and 1s}\}$
A caveat

0 0 1 1 1 1 0

0 0 1 1 1 0
A caveat

How do we know that this blank isn't one of the infinitely many blanks after our input string?

One solution
<table>
<thead>
<tr>
<th>One solution</th>
<th>One solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 0 0 1 1 1 1 0</td>
<td>x 0 0 1 1 1 1 0</td>
</tr>
<tr>
<td>One solution</td>
<td>One solution</td>
</tr>
<tr>
<td>x 0 0 1 1 1 1 0</td>
<td>x 0 0 1 1 1 1 0</td>
</tr>
</tbody>
</table>
One solution

\[ x \ 0 \ 0 \ x \ 1 \ 1 \ 1 \ 0 \]

One solution

\[ x \ x \ 0 \ x \ 1 \ 1 \ 1 \ 0 \]

One solution

\[ x \ x \ 0 \ x \ 1 \ 1 \ 1 \ 0 \]

One solution

\[ x \ x \ 0 \ x \ 1 \ 1 \ 1 \ 0 \]
One solution

One solution

One solution

One solution
One solution

[x x 0 x x 1 1 0]

One solution

[x x 0 x x 1 1 0]

One solution

[x x 0 x x 1 1 0]

One solution

[x x 0 x x 1 1 0]
<table>
<thead>
<tr>
<th>One solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x x x 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>One solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x x x 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>One solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x x x 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>One solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x x x 1 0</td>
</tr>
<tr>
<td>One solution</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>x x x x x x 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>One solution</th>
<th>One solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x x x 1 0</td>
<td>x x x x x x 1 0</td>
</tr>
</tbody>
</table>
Now the first non-\text{x} character we encounter is a \text{1}, so we want to cross it off but remember that we're looking for a matching \text{0}.
start → find 0/1

0 → x, R

find 1 → x, L

0 → 0, R

start → find 0/1

0 → x, R

find 1 → x, L

0 → 0, R

go home

0 → 0, L

1 → 1, L

x → x, L

start → find 0/1

0 → x, R

find 1 → x, L

0 → 0, R

go home

0 → 0, L

1 → 1, L

x → x, L
**Constant storage**

Sometimes a Turing machine needs to remember some additional information that can’t (at least conveniently) be put on the tape.

In this case, you can use the same techniques you used in designing DFAs and introduce extra states into the Turing machine's finite-state control.

The finite-state control can only remember one of finitely many things, but that might be all you need.

---

**An alternative approach**

Clearly the first language we built a TM to recognize,

\[ \{0^n1^n \mid n \in \mathbb{N}_0 \} \]

and this language,

\[ \{w \in \Sigma^* \mid w \text{ has the same number of } 0\text{s and } 1\text{s} \} \]

are related.

Could we use our Turing machine for the first language to recognize the second?
An alternative approach

To check if a string is in \( \{ w \in \Sigma^* \mid w \) has the same number of 0s and 1s}:

1. Sort the string so all 0s are before 1s.
2. Run the Turing machine for \( \{ 0^n1^n \mid n \in \mathbb{N} \} \) on the resulting tape.
3. Accept if it accepts; reject if it rejects.

You can think of this approach as using subroutines.

Turing machine subroutines

A TM subroutine is a Turing machine that, instead of accepting or rejecting an input, does some sort of processing job.

TM subroutines let us compose larger TMs out of smaller TMs, just as you’d use a variety of helper functions to write a complicated program.

Here, we could write a TM subroutine to sort a sequence of 0s and 1s into ascending order.

Turing machine subroutines

Note that subroutines are not a special part of the model for Turing machines.

They are a convenient way to think about building a single complex Turing machine out of simpler pieces.

Turing machine subroutines

Typically, when a subroutine is done running, you have it enter a state marked “done” with a dashed line around it.

When we're composing multiple subroutines together, the idea is that we'll snap in a real state for the “done” state.
We won’t go through implementing a Turing machine subroutine to sort the input string.

You can do it in very much the same way you would implement a sorting algorithm in a programming language.

Using subroutines to solve one problem using the solution to another is also a move toward the last proof technique we’ll cover in the course, reductions.

More on them later!

**Example: Turing machine arithmetic**

Let’s design a Turing machine that, given a tape that looks like this:

|   |   |   |   | 1 | 3 | 7 | 4 | 2 |

ends up having the tape look like this:

|   |   |   |   | 1 | 7 | 9 | 0 | 0 |

In other words, a Turing machine that can add two numbers.
There are many ways we could design this Turing machine.

Here’s one approach:

Build a Turing machine to increment a number.

Build a Turing machine to decrement a number.

Combine them, repeatedly decrementing the second number and incrementing the first.

def add(num1, num2):
  while num2 > 0:
    increment(num1)
    decrement(num2)

Let’s write this subroutine first.

Incrementing numbers

Let’s begin by building a TM that increments a number. We’ll assume that

- the tape head points at the start of a number
- there are at least two blanks to the left of the number
- there’s at least one blank at the end of the number

The tape head will end at the start of the number after incrementing it.
**Incrementing numbers**

Let's begin by building a TM that increments a number. We'll assume that

- the tape head points at the start of a number
- there are at least two blanks to the left of the number
- there's at least one blank at the end of the number

The tape head will end at the start of the number after incrementing it.

<table>
<thead>
<tr>
<th>9</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
</table>

The tape head will end at the start of the number after incrementing it.

<table>
<thead>
<tr>
<th>9</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
</table>
Incrementing numbers
Let’s begin by building a TM that increments a number. We’ll assume that

- the tape head points at the start of a number
- there are at least two blanks to the left of the number
- there’s at least one blank at the end of the number

The tape head will end at the start of the number after incrementing it.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>9</th>
<th>9</th>
<th>9</th>
</tr>
</thead>
</table>

Incrementing numbers
Let’s begin by building a TM that increments a number. We’ll assume that

- the tape head points at the start of a number
- there are at least two blanks to the left of the number
- there’s at least one blank at the end of the number

The tape head will end at the start of the number after incrementing it.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>9</th>
<th>9</th>
<th>9</th>
</tr>
</thead>
</table>
Let's begin by building a TM that increments a number. We'll assume that:
- The tape head points at the start of a number.
- There are at least two blanks to the left of the number.
- There's at least one blank at the end of the number.

The tape head will end at the start of the number after incrementing it.
Let’s begin by building a TM that increments a number. We’ll assume that

- the tape head points at the start of a number
- there are at least two blanks to the left of the number
- there’s at least one blank at the end of the number

The tape head will end at the start of the number after incrementing it.
Incrementing numbers

Let’s begin by building a TM that increments a number. We’ll assume that
the tape head points at the start of a number
there are at least two blanks to the left of the number
there’s at least one blank at the end of the number

The tape head will end at the start of the number after incrementing it.

```python
def increment(num):
    go to the end of the number
    while current_digit == 9:
        current_digit = 0
        go left one digit
        current_digit += 1
    go to the start of the number
```
To end -> L

0 → 0, R
1 → 1, R
...
9 → 9, R

wrap 9s

0 → 1, L
1 → 2, L
...
8 → 9, L

back home

done!

1 0 0 2
def add(num1, num2):
    while num2 > 0:
        increment(num1)
        decrement(num2)  # Next!

Decrementing numbers

Now let’s build a TM that decrements a number. We’ll assume that

- the tape head points at the start of a number
- there’s at least one blank on each side of the number

The tape head will end at the start of the number after decrementing it.

If the number is 0, the subroutine should signal this rather than go negative.
Decrementing numbers

Now let’s build a TM that decrements a number. We’ll assume that

- the tape head points at the start of a number
- there’s at least one blank on each side of the number

The tape head will end at the start of the number after decrementing it.

If the number is 0, the subroutine should signal this rather than go negative.

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
</table>

Decrementing numbers

Now let’s build a TM that decrements a number. We’ll assume that

- the tape head points at the start of a number
- there’s at least one blank on each side of the number

The tape head will end at the start of the number after decrementing it.

If the number is 0, the subroutine should signal this rather than go negative.

| 1 | 0 | 1 |
Decrementing numbers

Now let’s build a TM that decrements a number. We’ll assume that

- the tape head points at the start of a number
- there’s at least one blank on each side of the number

The tape head will end at the start of the number after decrementing it.

If the number is 0, the subroutine should signal this rather than go negative.
Decrementing numbers

Now let's build a TM that decrements a number. We'll assume that

the tape head points at the start of a number
there's at least one blank on each side of the number

The tape head will end at the start of the number after decrementing it.

If the number is 0, the subroutine should signal this rather than go negative.

1 0 1
Decrementing numbers

Now let's build a TM that decrements a number. We'll assume that

- the tape head points at the start of a number
- there's at least one blank on each side of the number

The tape head will end at the start of the number after decrementing it.

If the number is 0, the subroutine should signal this rather than go negative.
Decrementing numbers

Now let’s build a TM that decrements a number. We’ll assume that
the tape head points at the start of a number
there’s at least one blank on each side of the number
The tape head will end at the start of the number after decrementing it.
If the number is 0, the subroutine should signal this rather than go negative.
Decrementing numbers

Now let's build a TM that decrements a number. We'll assume that

- the tape head points at the start of a number
- there's at least one blank on each side of the number

The tape head will end at the start of the number after decrementing it.

If the number is 0, the subroutine should signal this rather than go negative.
Decrementing numbers

def decrement(num):
    go to the end of the number
    if (every digit was 0):
        signal that we’re done
    while current_digit == 0:
        current_digit = 9
        go left one digit
        current_digit -= 1
    go to the start of the number
non-zero?

0 → 0, R

0 → 0, R

0 → 0, R

0 → 0, R

1 → 1, R

1 → 1, R

2 → 2, R

2 → 2, R

... ...

9 → 9, R

9 → 9, R

0 → 0, R

0 → 0, R

non-zero?
non-zero?

start

0 -> 0, R

1 -> 1, R

2 -> 2, R

... 

9 -> 9, R

to end

0 -> 0, R

1 -> 1, R

2 -> 2, R

... 

9 -> 9, R

to end

0 -> 0, R

1 -> 1, R

2 -> 2, R

... 

9 -> 9, R

0 2 0 0

0 2 0 0
0 → 0, R
1 → 1, R
... 9 → 9, R

0 → 0, R
1 → 1, R
2 → 2, R
... 9 → 9, R

0 → 0, R
1 → 1, R
2 → 2, R
... 9 → 9, R

0 → 0, R
start
non-zero?

□ → □, L

to end

0 → 0, R
1 → 1, R
... 9 → 9, R

0 → 0, R
1 → 1, R
2 → 2, R
... 9 → 9, R

0 → 0, R
start
non-zero?

□ → □, L

to end

0 → 0, R
1 → 1, R
2 → 2, R
... 9 → 9, R

0 → 0, R
start
non-zero?

□ → □, L

to end

0 → 0, R
1 → 1, R
2 → 2, R
... 9 → 9, R

0 → 0, R
start
non-zero?

□ → □, L

to end

0 → 0, R
1 → 1, R
2 → 2, R
... 9 → 9, R

0 → 0, R
start
non-zero?

□ → □, L

to end

0 → 0, R
1 → 1, R
2 → 2, R
... 9 → 9, R

0 → 0, R
start
non-zero?

□ → □, L

to end

0 → 0, R
1 → 1, R
2 → 2, R
... 9 → 9, R

0 → 0, R
start
non-zero?

□ → □, L

to end

0 → 0, R
1 → 1, R
2 → 2, R
... 9 → 9, R

0 → 0, R
start
non-zero?

□ → □, L

to end

0 → 0, R
1 → 1, R
2 → 2, R
... 9 → 9, R

0 → 0, R
start
non-zero?

□ → □, L

to end

0 → 0, R
1 → 1, R
2 → 2, R
... 9 → 9, R

0 → 0, R
start
non-zero?
non-zero?

start

0 → 0, R
1 → 1, R
2 → 2, R
... 9 → 9, R

0 → 0, R

to end

wrap zeroes

0 → 0, L
1 → 1, L
2 → 2, L
... 9 → 9, L

0 → 0, R

non-zero?

back home

0 → 0, L
1 → 1, L
2 → 2, L
... 9 → 9, L

0 → 0, R

start

0 → 0, R
1 → 1, R
2 → 2, R
... 9 → 9, R

0 → 0, R

to end

wrap zeroes

0 → 0, L
1 → 1, L
2 → 2, L
... 9 → 9, L

0 → 0, R

non-zero?

back home

0 → 0, L
1 → 1, L
2 → 2, L
... 9 → 9, L

0 → 0, R

start

0 → 0, L
1 → 1, L
2 → 2, L
... 9 → 9, L

0 → 0, R

to end

wrap zeroes

0 → 0, L
1 → 1, L
2 → 2, L
... 9 → 9, L

0 → 0, R

non-zero?

back home

0 → 0, L
1 → 1, L
2 → 2, L
... 9 → 9, L

0 → 0, R

start

0 → 0, L
1 → 1, L
2 → 2, L
... 9 → 9, L

0 → 0, R

to end

wrap zeroes

0 → 0, L
1 → 1, L
2 → 2, L
... 9 → 9, L

0 → 0, R

non-zero?

back home

0 → 0, L
1 → 1, L
2 → 2, L
... 9 → 9, L

0 → 0, R

start

0 → 0, L
1 → 1, L
2 → 2, L
... 9 → 9, L

0 → 0, R

to end

wrap zeroes

0 → 0, L
1 → 1, L
2 → 2, L
... 9 → 9, L

0 → 0, R

non-zero?

back home

0 → 0, L
1 → 1, L
2 → 2, L
... 9 → 9, L
Turing machine subroutines

Sometimes a subroutine needs to report back some information about what happened.

Just as a function can return multiple different values, we can have subroutines to have different “done” states.

Each state can then be wired to a different state, so a Turing machine using the subroutine can control what happens next.

def add(num1, num2):
    while num2 > 0:
        increment(num1)
        decrement(num2)

Putting it all together

Our goal is to build a Turing machine that, given two numbers, adds those numbers together.

Using our subroutines

We’ll build our new machine using our existing increment and decrement subroutines:
Using our subroutines

We'll build our new machine using our existing increment and decrement subroutines:
Using our subroutines

We'll build our new machine using our existing increment and decrement subroutines:

1 3 7 4 2

Using our subroutines

We'll build our new machine using our existing increment and decrement subroutines:

1 3 7 4 2

Using our subroutines

We'll build our new machine using our existing increment and decrement subroutines:

1 3 7 4 2

Using our subroutines

We'll build our new machine using our existing increment and decrement subroutines:

1 3 7 4 1
Using our subroutines
We'll build our new machine using our existing increment and decrement subroutines:
Using our subroutines

We'll build our new machine using our existing increment and decrement subroutines:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 1 | 3 | 7 | 4 | 1 |

Using our subroutines

We'll build our new machine using our existing increment and decrement subroutines:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 1 | 3 | 7 | 4 | 1 |
Using our subroutines

We'll build our new machine using our existing increment and decrement subroutines:

1 3 8 4 1
start

1 3 7 4 2

start

1 3 7 4 2

start

0 → 0, R
1 → 1, R
... 
9 → 9, R

start

1 3 7 4 2
start
0 → 0, R
1 → 1, R
...
9 → 9, R

□ → □, R

start
0 → 0, R
1 → 1, R
...
9 → 9, R

□ → □, R

start
0 → 0, R
1 → 1, R
...
9 → 9, R

□ → □, R

1 3 7 4 2

□ → □, R

1 3 7 4 2
1 to 2nd num.

\[\begin{align*}
0 & \to 0, R \\
1 & \to 1, R \\
\vdots & \\
9 & \to 9, R
\end{align*}\]
start → 2nd num. □ → □, R
decc.

0 → 0, R
1 → 1, R
...
9 → 9, R

1 3 7 4 2

start → 2nd num. □ → □, R
decc.

0 → 0, R
1 → 1, R
...
9 → 9, R

1 3 7 4 2

start → 2nd num. □ → □, R
decc.

0 → 0, R
1 → 1, R
...
9 → 9, R

1 3 7 4 1

start → 2nd num. □ → □, R
decc.

0 → 0, R
1 → 1, R
...
9 → 9, R

1 3 7 4 1
to 2nd num. "to 1st num." start □ → □, R decr. □ → □, R
0 → 0, R
1 → 1, R
... 
9 → 9, R
to 2nd num. to 1st num. start □ → □, R decr. □ → □, R
0 → 0, R
1 → 1, R
... 
9 → 9, R
to 1st num. to 2nd num. start □ → □, R decr. □ → □, R
0 → 0, R
1 → 1, R
... 
9 → 9, R
to 1st num.
to 2nd num.

\[ \text{start} \quad \begin{array}{c}
0 \rightarrow 0, R \\
1 \rightarrow 1, R \\
\cdots \\
9 \rightarrow 9, R \\
\end{array} \]

do to 1st num.

done

\[ \text{to 2nd num.} \quad \begin{array}{c}
0 \rightarrow 0, L \\
1 \rightarrow 1, L \\
\cdots \\
9 \rightarrow 9, L \\
\end{array} \]

do to 1st num.

done

\[ \text{to 1st num.} \quad \begin{array}{c}
0 \rightarrow 0, L \\
1 \rightarrow 1, L \\
\cdots \\
9 \rightarrow 9, L \\
\end{array} \]
| 0 → 0, R |
| 1 → 1, R |
| ... |
| 9 → 9, R |
| 0 → 0, L |
| 1 → 1, L |
| ... |
| 9 → 9, L |

| 0 → 0, L |
| 1 → 1, L |
| ... |
| 9 → 9, L |

| go home |
| incr. |
| □ → □, R |

| incr. |
| □ → □, R |

| to 2nd num. |
| □ → □, R |

| to 1st num. |
| □ → □, R |

| start |
| done |

| 1 | 3 | 8 | 4 | 0 |

| 1 | 3 | 8 | 4 | 0 |

| 1 | 3 | 8 | 4 | 0 |

<p>| 1 | 3 | 8 | 4 | 0 |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

**Diagram:**

- **Start:**
  - **Increment:** $0 \rightarrow 0$, $R$
  - $1 \rightarrow 1$, $R$
  - $9 \rightarrow 9$, $R$

- **Decrement:**
  - $0 \rightarrow 0$, $L$
  - $1 \rightarrow 1$, $L$
  - $9 \rightarrow 9$, $L$

- **Go Home:**
  - $0 \rightarrow 0$, $L$
  - $1 \rightarrow 1$, $L$
  - $9 \rightarrow 9$, $L$

- **Done:**
  - $0 \rightarrow 0$, $L$
  - $1 \rightarrow 1$, $L$
  - $9 \rightarrow 9$, $L$

- **To 2nd Num.:**
  - $0 \rightarrow 0$, $R$
  - $1 \rightarrow 1$, $R$
  - $9 \rightarrow 9$, $R$

- **To 1st Num.:**
  - $0 \rightarrow 0$, $L$
  - $1 \rightarrow 1$, $L$
  - $9 \rightarrow 9$, $L$

**Table:**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>3</th>
<th>8</th>
<th>4</th>
<th>0</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
to 2nd num.  

start  

to 1st num.  

done  

to home

1 3 9 3 9

to 2nd num.  

start  

to 1st num.  

done  

to home

1 3 9 3 9

to 2nd num.  

start  

to 1st num.  

done  

to home

1 4 0 3 9

to 2nd num.  

start  

to 1st num.  

done  

to home

1 4 0 3 9
Many transitions later…
Using subroutines

Once you’ve built a subroutine, you can wire it into another Turing machine with something that, schematically, looks like this:

Intuitively, this corresponds to transitioning to the start state of the subroutine, then replacing the “done” state of the subroutine with the state at the end of the transition.

How does this relate to languages?

Every computation can be encoded as a language. We could, for instance, have the language of all strings \( \{x\#y\#z \mid x + y = z\} \).

How do we check whether an input is in the language?

Add \( x \) and \( y \) and then compare it with \( z \).
Many of the languages we’ve recognized with Turing machines have been context-free; we could have used a pushdown automaton to recognize them.

But Turing machines can also recognize languages that PDAs cannot.

Be sure you understand the example Turing machines in the textbook for these languages:

- \{w#w \mid w \in \{0, 1\}^*\}, pages 166–7
- \{0^n \mid n \geq 0\}, pages 171–2
- \{abck \mid i \times j = k \text{ and } i, j, k \geq 1\}, page 174
- \{#x_1#x_2#\ldots #x_k \mid \text{each } x_i \in \{0, 1\}^* \text{ and } x_i \neq x_j \text{ for each } i \neq j\}, page 175

All of these are non-context-free languages.

Acknowledgments

This lecture incorporates material from:

- W. Daniel Hillis
- Nancy Ide
- Keith Schwarz
- Michael Sipser