Ilovepdf merged - all notes for sem

COMP2022 Models of Computation (University of Sydney)

What are programming languages for?

Programs solve computational problems/tasks.

What does the following code do?

```
def mentions_Australia(text : str) -> bool:
    if "Australia" in text:
        return True
    else:
        return False
```

- On input `"Commonwealth of Australia"` this function returns `True`.
- On input `"Republic of Austria"` this function returns `False`.

Input: string `text`
Output: `True` if `text` contains `"Australia"`, `False` otherwise.

Computational problem

- Specifies what the allowed inputs are, and for each allowed input what the corresponding output should be.[1]
- Also called a specification.

Implementation

- code, e.g., in `Python`, that takes an input and produces the corresponding output.
- In this case, we say that:
  the implementation solves the computational problem, or
  the implementation realises the specification.
- different from an algorithm, which is more high-level

[1] Formally, a computational problem is a function from strings to strings.

Big idea (1)

Most programming languages can solve the same computational problems as Python.

Two programming languages are expressively equivalent if they can solve the same computational problems.

Claim. Python and C are expressively equivalent.

How could we establish such a claim?

1. Every problem solvable in Python is solvable in C.
   - Why? The standard Python interpreter is written in C.
2. Every problem solvable in C is solvable in Python.
   - Why? There are C compilers written in Python (just for fun).

Big idea (2)

Some computational problems cannot be solved in Python.

We will prove this later in the semester.

To get a hint, let's think about a computational problem that is hard for humans to do...

Is there a program that takes the source code for two Python functions and tells if the two functions are equivalent?

The answer is No!

There is a general fact here: no general-purpose programming language (like Python) is powerful enough to solve problems about the behaviour of programs.

We will see this later in the course...

Big idea (3)

We don't need complex languages to solve computational problems.

Even tiny fragments of Python are as powerful as Python with all its features (and modules).

To get an idea of this, let's restrict ourselves to Python in which the only variables are non-negative integers.

If it bothers you that we are ignoring other data types like strings, lists, tuples etc, just remember that under the hood of the computer all data is represented as sequences of bits, which themselves can be thought of as numbers. E.g., the 8 bit sequence 00001001 is the binary representation of the number 9.

If it still bothers you, then just weaken the claim to "We don't need complex languages to solve computational problems **about non-negative integers.**"

Tiny Python is equivalent to Python in the following sense:

1. Every computational problem that can be solved in Tiny Python can also be solved in Python (Why?)

2. Every computational problem that can be solved in Python can also be solved in Tiny Python — this would require a lot of work to show. But here are some steps...

We can now do simple `for` loops in Tiny Python!

```
# Tiny Python code to do 'for x in range(0,y): ...'

x=0 # as above
s=y # as above

while s!=0:
  ...
  s=s-1
  x=x+1
```


What does this Tiny Python program do?

```
x=y # as above
t=z # as above

while t!=0:
  x=x+1
  t=t-1
```

It does the assignment $x = y + z$

Another idea we will see is that there are somewhat clear levels of computation.

In COMP2123, one can think of these levels as problems solvable in $O(n)$ time, $O(n^2)$ time, $O(n^3)$ time, etc.

In COMP2022 there are four main levels:

1. Turing-recognisable
2. Context-sensitive (we will not study this one)
3. Context-free
4. Regular

Chomsky Hierarchy: every problem solvable in one level is solvable in all higher levels too.

Regular level

The models in the regular level can do (advanced) pattern matching on strings.

Input: `pattern` and `text` (strings)
Output: `True` if `pattern` occurs in `text`, and `False` otherwise

`"456"` occurs in `"1234567890"` but `"1569"` does not.

The models are called regular expressions and finite-state automata

Context-free level

The models in the context-free level can do syntax checking of typical expressions.

Input: string `expr`
Output: `True` if `expr` is a legal Python arithmetic expression, and `False` otherwise.

`(1+2)*3` is legal, but `(1+)2*3` is not.

The models are called context-free grammars (and pushdown automata)

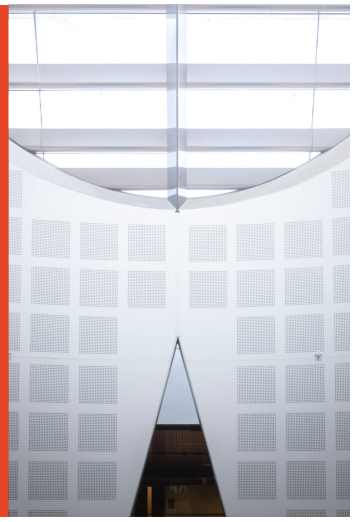
COMP2022
Models of Computation

Regular Expression

Sasha Rubin

August 12, 2025

THE UNIVERSITY OF SYDNEY



Regular expressions are patterns that describe (sets of) strings.

Uses:

- Text processing
- Features in (machine learning) classifiers
  `COMP5046:Natural Language Processing`
- Scanners (aka Lexical analysers, Tokenisers)
- Specification of data formats
- Foundations of query languages for graph databases

In Python, the years of the form 19xx can be described by the regular expression

`^19\d{2}$`

We want to match entire strings like 1926, not strings that instead contain a year, like 02 – 01 – 1926:

`^`, matches the start of the string
`$`, matches the end of the string

`\d` matches any digit

Here, \d stands for (0|1|2|3|4|5|6|7|8|9)
and `|` means "or"

`{2}` repeats the preceeding item (in this case \d) that many times.

Our regular expressions will match the entire string, so we don't need `^` and `$`:

19(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)

Characters and Alphabets

Characters in Python are Unicode code points.[5]

This is far too many for us. We will stick to just a few characters.

For us, a set of characters is called an alphabet.[6] Alphabets are named Σ (Sigma).

- $\Sigma = \{0, 1\}$ is the classic binary alphabet.
- $\Sigma = \{a, b, c, \cdots, z\}$ is the lower-case English alphabet.
- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, \div, \times, (,)\}$ is an arithmetic alphabet.

[5] Unicode provides a unique number for every character, independent of any device. For instance, U+0028 is left parenthesis (and U+0041 is the letter A.

[6] Similar to character sets in `Python`.

Strings in COMP2x22

Strings for us are similar — they are sequences of characters — except that:

- we often don't write the quotation marks
  $abc123$ is a string
- we specify the alphabet (= set of characters that can be used)
  011 and 10010 are strings over the alphabet $\{0, 1\}$ (they are also strings over the alphabet $\{0, 1, 2, 3\}$ but not over the alphabet $\{1, 2\}$)
- we don't use $+$ for concatenation; instead, we either use $\cdot$ (dot) or just put the strings next to each other
  the concatenation of abc and 123 is $abc123$
- the empty string is written ϵ (epsilon)

Regular expressions: Syntax

Definition
Let Σ be an alphabet. The regular expressions over Σ are defined by the following recursive process:

1. The symbols $\emptyset$ and ϵ are regular expressions[a]
2. Each symbol a from Σ is a regular expression
3. If R_1, R_2 are regular expressions then so is $(R_1 \,|\, R_2)$
4. If R_1, R_2 are regular expressions then so is $(R_1 R_2)$
5. If R is a regular expressions then so is R^*

[a] The regular expression $\emptyset$ (read "emptyset") will not match any string... we will see this shortly.

- $(R_1 \,|\, R_2)$ read "R_1 or R_2"
- $(R_1 R_2)$ read "R_1 cat R_2"
- R^* read "R star"

Shorthands!

- We may drop the outermost parentheses to improve readability.
  - write $a \,|\, \emptyset$ instead of $(a \,|\, \emptyset)$
  - write ab^* instead of (ab^*), which is different from $(ab)^*$
- We may drop parentheses for associative operations.
  - we may write $(R_1 \,|\, R_2 \,|\, R_3)$ instead of $((R_1 \,|\, R_2) \,|\, R_3)$
  - we may write $(R_1 R_2 R_3)$ instead of $((R_1 R_2) R_3))$
- If $A = \{a, b, c\}$ we may write A instead of $(a \,|\, b \,|\, c)$.
  - so, e.g., we may write A^* instead of $(a \,|\, b \,|\, c)^*$.
  - we may do this for any finite set A of strings.
- We may also use exponentiation
  - write A^2 for AA.

Let $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. The following is a regular expression that uses these shorthands:

$$19\Sigma^2$$

Without exponentiation shorthand:

$$19\Sigma\Sigma$$

Without the associative shorthand:

$$(((19)\Sigma)\Sigma)$$

Here Σ without shorthands would be:

$$((((((((((0|1)|2)|3)|4)|5)|6)|7)|8)|9)$$

First, let's recall some basic discrete mathematics, and introduce some terminology.

A set of strings whose characters are from the alphabet Σ is called a language over Σ.[7]

Examples of languages.

- All binary strings.
- All binary strings that end in 011.
- All legal arithmetic expressions in Python.
- All legal Python programs.
- All legal Python programs that halt on every input.

What is a suitable alphabet in each case?

[7] Sometimes these are called formal languages to distinguish them from natural languages, like English.

We now define some operations on languages, i.e., ways to form new languages from old ones.

Let A, B be languages over Σ.

The new languages will also be over Σ.

- Union of languages
- Intersection of languages
- Complement of languages
- Concatenation of languages
- Star of languages

Definition

For languages A, B:

- $A \cup B$ (read "A union B") is the set of strings that are in A, or in B, or in both.
- $A \cap B$ (read "A intersection B") is the set of strings that are in A and in B.
- $A \setminus B$ (read "A minus B") is the set of strings that are in A but not B.

Venn Diagrams can be useful.

Definition
Define AB (read "A concatenate B", or just "A cat B") as the set of strings formed by concatenating a string from A with a string from B.

In math:

$$AB = \{xy : x \in A, y \in B\}$$

We use exponentiation notation A^k (read 'A to the power of k') to mean concatenate A with itself k times.

- For $k \in \mathbb{Z}^{\geq 1}$, define A^k to be $\overbrace{AA\cdots A}^{k}$
- Convention: A^0 is defined to be $\{\epsilon\}$.
- Note that $A^{n+m} = A^n A^m$ (even if n or m is zero).

Definition
Define A^* (read "A star") as

$$A^* = A^0 \cup A^1 \cup A^2 \cup A^3 \cup \dots$$

Think of A^* as "repeated concatenation", but without fixing the number of iterations.

Sometimes you will see a plus instead of a star:
A^+ (read "A plus") is $A^1 \cup A^2 \cup A^3 \cup \dots$.

Language represented by a regular expression

Write $\mathrm{Lang}(R)$ for the language that the regular expression R represents.

Here are some examples with $\Sigma = \{a, b\}$:

- $\mathrm{Lang}((a \mid b)a) = \{aa, ba\}$.
- $\mathrm{Lang}(a^*) = \{\epsilon, a, aa, aaa, \dots\}$.
- $\mathrm{Lang}(a^*b^*) = \{\epsilon, a, b, aa, ab, bb, aaa, aab, abb, bbb, \dots\}$.

$\mathrm{Lang}(R)$ is also called the language specified/represented/defined by R, or the language of R.

Language represented by a regular expression

Definition
We define $\mathrm{Lang}(R)$ recursively as follows:

1. $\mathrm{Lang}(\epsilon) = \{\epsilon\}$
2. $\mathrm{Lang}(\emptyset) = \{\}$
3. $\mathrm{Lang}(a) = \{a\}$ (for $a \in \Sigma$)
4. $\mathrm{Lang}(R_1 \mid R_2) = \mathrm{Lang}(R_1) \cup \mathrm{Lang}(R_2)$
5. $\mathrm{Lang}(R_1 R_2) = \mathrm{Lang}(R_1)\mathrm{Lang}(R_2)$
6. $\mathrm{Lang}(R^*) = \mathrm{Lang}(R)^*$

If a string s is in the set $\mathrm{Lang}(R)$ we say s matches R

Which languages can be specified by regular expressions?

Regular expressions can be used to specify keywords and identifiers in programming languages[8]

- The set of identifiers in `Python`
- Signed and unsigned integers
- Hexadecimal numbers prefixed with '0x'

However, for specifying expressions and statements, we will need a more powerful model (context-free grammars, later).

For instance, the set of well-formed arithmetic expressions is not the language of any regular expression! We will prove this fact, later.

[8] Keyword = reserved word; identifier = user-defined name

Automata in a nutshell

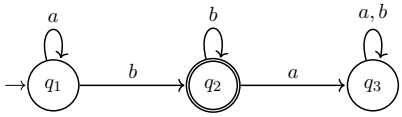
An automaton is a character-processing program that:

- Takes a string as input.
- Can only use variables with finite domains — we use one variable `state` taking finitely many values.
- Can read the input string character by character.
- Must decide to `"Accept"` or `"Reject"` the input string.

Graphical representation of an automaton

Such programs have a graphical representation as a directed edge-labeled graph:

- Vertices represent states.
- Labeled-edges $q \xrightarrow{a} q'$ represent transitions between states.
  For every state q and every character $a \in \Sigma$ there is a one transition leaving q labeled a.
- The start state is marked with an incoming arrow.
- The final states[1] are marked with an extra circle.

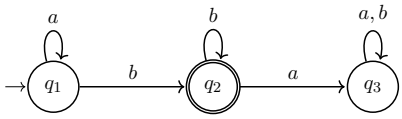


[1] There can be more than one!

Terminology

We call such automata deterministic finite automata (DFA).

- DFA takes a string `w` as input.
- The run on input `w` is the path from the start state that is labeled by the string `w`.
- If the run on `w` ends in a final state, the automaton accepts `w`, otherwise it rejects `w`.
- The language of a DFA D (aka, language recognised by D) is the set of strings that it accepts (no more, no less). It is written $\text{Lang}(D)$.



- The run of this DFA on input abb is

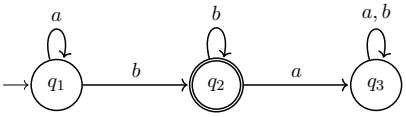
$$q_1 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{b} q_2$$

  Since q_2 is a final state, abb is accepted.
- The run of this DFA on input $abba$ is

$$q_1 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{b} q_2 \xrightarrow{a} q_3$$

  Since q_3 is not a final state, $abba$ is rejected.

Anatomy of a DFA




```
Alphabet = a b
States = q1 q2 q3
Start = q1
Final = q2
q1 a q1
q1 b q2
q2 a q3
q2 b q2
q3 a q3
q3 b q3
```


Definition of DFA

Definition

A deterministic finite automaton (DFA) D consists of 5 items

$$(Q, \Sigma, \delta, q_0, F)$$

where

1. Q is a finite set of states,
2. Σ is the alphabet (aka input alphabet),
3. $\delta : Q \times \Sigma \to Q$ (read "delta") is the transition function,
   - If $\delta(q, a) = q'$ we write $q \xrightarrow{a} q'$, called a transition.
4. $q_0 \in Q$ is the start state (aka initial state), and
5. $F \subseteq Q$ is the set of final states (aka accepting states).

Regular languages

The languages of DFAs are so important, we give them a name: they are the regular languages.

Designing automata tips (i)

Imagine you are the automaton reading the string symbol by symbol:

- what information about the string read so far do you need to make a decision on whether to accept or reject.
- can you update this information if another input symbol arrives?
- the states will store this information.

Designing automata tips (ii)

Build automata out of other automata using certain operations.

If you have DFA for L_1, L_2 then you can get DFA for

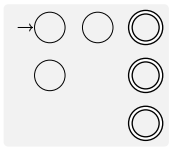
- $\Sigma^* \setminus L_1$
- $L_1 \cup L_2$ (and thus, by DeMorgan!, also $L_1 \cap L_2$)
- $L_1 L_2$
- $(L_1)^*$

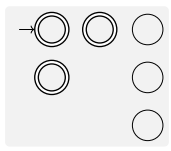
Let's see how to do complement.[2]

[2] Doing union is in the tutorial, concatenation and star are trickier and we will do them in a future lecture.

Complement for DFA

Idea: swap final and non-final states.





Important questions about DFA

- Can all languages be described by DFAs?
  No! (we will come back to this in a future lecture)
- There are natural computational problems associated with DFAs... Are there (polynomial time) algorithms that solve them?

Acceptance problem for DFA

Input: `D` (DFA), `w` (string).
Output: decide if M accepts `w`.

```python
def acceptance(D, w) -> bool:
  state = D["init"]
  for char in w:
    state = D["transition"][state,char]
  return state in D["final"]
```

Also called the membership problem for DFA since it is asking if w is a member of (element of) $\mathrm{Lang}(D)$.

Non-emptiness problem for DFA

Input: DFA M
Output: decide if $\mathrm{Lang}(M) \neq \emptyset$.

(See tutorial)

Equivalence problem for DFA

Input: DFAs M_1, M_2.
Output: decide if $\mathrm{Lang}(M_1) = \mathrm{Lang}(M_2)$.

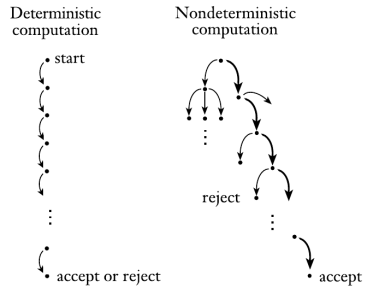
(See tutorial)

Terminology

A decision problem is a computational problem whose possible outputs are `True` and `False`.

Or `0` and `1`; or `"yes"` and `"no"`; or `"Accept"` and `"Reject"`.

Intuitively, "nondeterminism" refers to situations in which the next state of a computation is not uniquely determined by the current state and current input.



Let's introduce nondeterminism into automata.

- A nondeterministic finite automaton (NFA) is like a DFA except that states can have zero, one, or more outgoing transitions on the same input symbol.
  In an NFA, a string can label zero, one, or more paths.
- A string is accepted by an NFA if it labels some path from the start state to a final state.
  Such a path is called an accepting path

Pattern matching made easier!

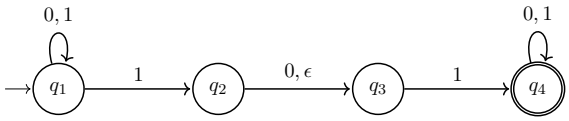
- NFAs are good for specifying languages of the form "the string has x as a substring"
- E.g., the set of strings that contain "aab" as a substring

Epsilon transitions

Our NFA also allow epsilon transitions.

This means that some edges can be labeled ϵ

Such a transition can be taken without 'consuming' an input symbol.



An accepting run of this NFA on input 011 is

$$q_1 \xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{\epsilon} q_3 \xrightarrow{1} q_4$$

Comparing NFAs and DFAs

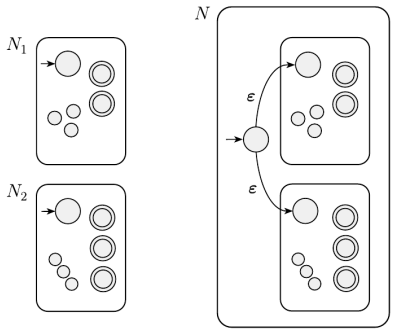
1. In a DFA, every input has exactly one run.[3] The input string is accepted if this run is accepting.
2. In an NFA, an input may have zero, one, or more runs. The input string is accepted if at least one of its runs is accepting.

[3] With our convention of not drawing error states, an input of a DFA may have no runs too.

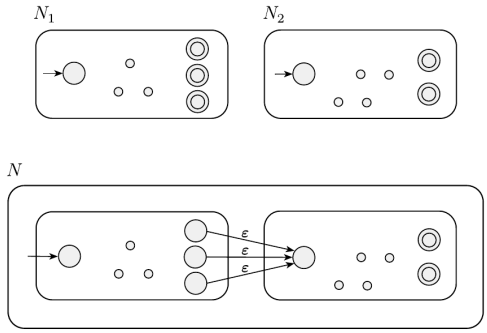
NFAs are at least as expressive as DFAs. Why?

You can think of a DFA as an NFA in which there is no nondeterminism.

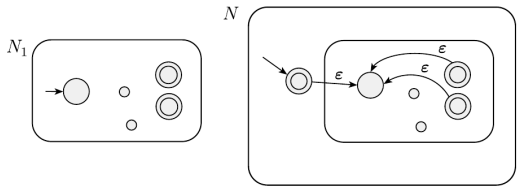
NFA for the union of NFAs



NFA for the cat of NFAs



NFA for the star of an NFA



We can now turn any RE into an equivalent NFA.

What does this show?

1. RE and NFA are expressively equivalent?
2. RE are at least as expressive as NFA?
3. NFA are at least as expressive as RE?

Where are we going?

Recall:

- We are going to show that DFA, NFA and regular expressions specify the same set of languages!
- We will do this with a series of transformations:

1. From Regular Expressions to NFAs (done today!)
2. From NFAs to NFAs without ϵ-transitions (next time)
3. From NFAs without ϵ-transitions to DFAs (next time)
4. From DFAs to Regular Expressions (next time)

The language recognised by a DFA

Definition

- A run (aka computation) of D on $w = w_1 w_2 \cdots w_n$ is a sequence of transitions $q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} q_2 \xrightarrow{w_3} \cdots \xrightarrow{w_n} q_n$ where q_0 is the start state.
- The run is accepting if $q_n \in F$.
- If w has an accepting run then we say that D accepts w.
- The set $\mathrm{Lang}(D) = \{w \in \Sigma^* : D \text{ accepts } w\}$ is the language recognised by D (aka language of D).

Regular languages

Definition
A language $L \subseteq \Sigma^*$ is called regular if $L = \mathrm{Lang}(D)$ for some DFA D.

This means that D must accept all strings in L and reject all strings (in Σ^*) that are not in L.

Definition of NFA

Definition
A nondeterministic finite automaton (NFA) $M = (Q, \Sigma, \delta, q_0, F)$ is the same as a DFA except that

$$\delta : Q \times \Sigma_\epsilon \to P(Q),$$

called the transition relation.

- So $\delta(q, a)$ is a set of states (empty set is allowed, multiple states are allowed)
- If $q' \in \delta(q, a)$ we write $q \xrightarrow{a} q'$, called a transition.
- Here $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$.

So, we also allow epsilon-transitions. This amounts to transitions that do not consume the next input symbol.

The language recognised by an NFA M

The following definition formalises the idea that an NFA M describes the language $\mathrm{Lang}(M)$ of all strings that label paths from the start state to a final state.

Definition

- A run (aka computation) of an NFA M on string w is a sequence of transitions $q_0 \xrightarrow{y_1} q_1 \xrightarrow{y_2} q_2 \ldots \xrightarrow{y_m} q_m$ such q_0 is the start state, each $y_i \in \Sigma_\epsilon$, and $w = y_1 y_2 \cdots y_m$.[4]
- The run is accepting if $q_m \in F$.
- If w has at least one accepting run, then we say that w is accepted by M.
- The language recognised by M is $\mathrm{Lang}(M) = \{w \in \Sigma^* : w \text{ is accepted by } M\}$.

[4] Recall that $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$, and $\epsilon x = x\epsilon = x$ for all strings x.

Where are we going?

1. From Regular Expressions to NFAs (last lecture)
2. From NFAs to NFAs without ϵ-transitions (today)
3. From NFAs without ϵ-transitions to DFAs (today)
4. From DFAs to Regular Expressions (today)

This shows that RE, NFAs, NFAs without epsilon transitions, and DFAs have the same expressive power

There are four separate ideas here:

1. "building NFA for the union/cat/star of NFAs"
2. "skipping over epsilon transitions"
3. "remembering sets of states"
4. "bypassing states"

Theorem

We can convert an NFA N into an equivalent NFA M without epsilon-transitions.

Idea: "M skips over ϵ-transitions of N and then does an ordinary transition"

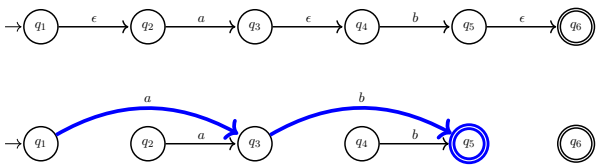
So M is like N but we remove all the ϵ-transitions and add new transitions and new final states...

Notation: Write $\mathrm{EC}(q, r)$ (stand for "Epsilon Closure") or $q \rightsquigarrow r$ to mean that N can go from q to r using zero or more ϵ-transitions.

Steps (for all states q, r, s of N, and all characters $a \in \Sigma$):

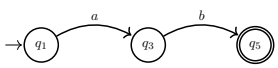
1. If $q \rightsquigarrow r$ and $r \xrightarrow{a} s$ then add $q \xrightarrow{a} s$ to M.
2. If $q \rightsquigarrow r$ and r is final, make q final in M.

How to compute if $\mathrm{EC}(q, r)$? (see Tutorial)



Since $q_1 \rightsquigarrow q_2 \xrightarrow{a} q_3$, add $q_1 \xrightarrow{a} q_3$
Since $q_3 \rightsquigarrow q_4 \xrightarrow{b} q_5$, add $q_3 \xrightarrow{b} q_5$
Since $q_5 \rightsquigarrow q_6$ and q_6 is final, make q_5 final.

This can be simplified...



Theorem

We can convert an NFA N without epsilon-transitions into an equivalent DFA.

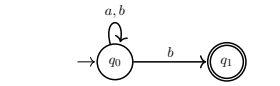
Idea: "subset construction"

Q: How would you simulate the NFA if you were pretending to be a DFA?

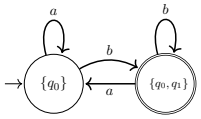
A: Keep track of the set of possible states that the NFA is in.

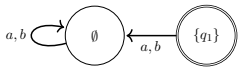
Q: And when would you accept?

A: If at least one of the states I'm keeping track of is an accepting state of the NFA.



DFA State	Input a	Input b
$\emptyset$	$\emptyset$	$\emptyset$
$\{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$
$\{q_1\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0, q_1\}$





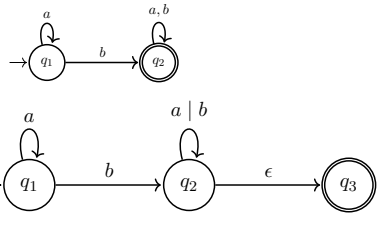
4. DFAs to Regular Expressions

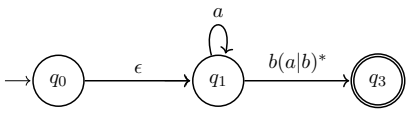
Theorem

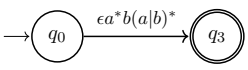
We can convert a DFA M into an equivalent regular expression R.

Idea:

1. We convert M into a generalised nondeterministic finite automaton (GNFA) which is like an NFA except that it can have regular expressions label the transitions.
2. We successively remove states from the automaton, until there is just one transition (from the start state to the final state).
3. We return the regular expression on this last transition.







Generalised nondeterministic finite automaton (GNFA)

- A run (aka computation) of a GNFA N on string w is a sequence of transitions

$$q_0 \xrightarrow{R_1} q_1 \xrightarrow{R_2} q_2 \xrightarrow{R_3} \dots \xrightarrow{R_m} q_m$$

  in N such that w matches the regular expression $R_1 R_2 \cdots R_m$.
- The run is accepting if $q_m \in F$.
- The language recognised by N is

$$\text{Lang}(N) = \{w \in \Sigma^* : w \text{ is accepted by } N\}$$

In the construction, we make sure that generalised NFAs are always of the following form:

1. the initial state has no incoming edges.
2. there is one final state, and it has no outgoing edges.
3. there are edges from every state (that is not final) to every state (that is not initial).

This simplifies the construction.

Here is how we convert the given DFA into a GNFA with these three restrictions.

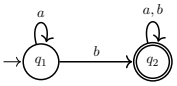
From DFAs to GNFAs

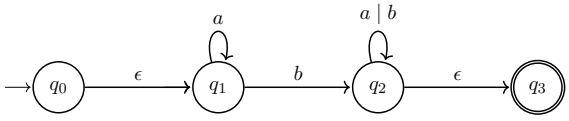
We can convert a DFA M into an equivalent NFA N.

Steps:

1. Add an initial state and ϵ-transition to the old intial state.
2. Add a final state and ϵ-transitions to it from all the old final states.
3. Add transitions for every pair of states, including from a state to itself, (except leaving the final, or entering the initial).

From DFAs to GNFAs





. . . and some $\emptyset$-transitions which we usually don't draw since the picture gets cluttered.

Removing states from GNFAs

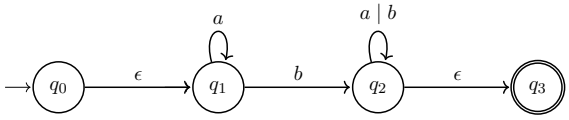
For every GNFA N with > 2 states there is a GNFA M with one less state such that $\mathrm{Lang}(N) = \mathrm{Lang}(M)$.

1. Pick a state q to eliminate.
2. For every pair of remaining states (s, t), replace the label from s to t by the regular expression

$$R_{s,t} \mid (R_{s,q}\ R_{q,q}^*\ R_{q,t})$$

where $R_{x,y}$ is the regular expression labeling the transition from state x to state y.

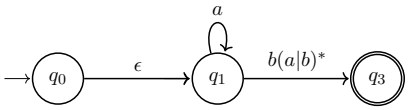
Removing states from GNFAs



Eliminate q_2:

$R_{s,t} \mid (R_{s,q}\ R_{q,q}^*\ R_{q,t})$

Removing states from GNFAs



Eliminate q_1:

$R_{s,t} \mid (R_{s,q}\ R_{q,q}^*\ R_{q,t})$

Summary

1. A language is regular if it is recognised by some DFA.
2. The following models of computation describe the same languages:
   - DFA
   - NFA
   - NFA without epsilon transitions
   - Regular Expressions.
3. The regular languages are closed under the Boolean operations union, intersection, complementation, as well as concatenation and Kleene star.

Other (more interesting?) non-regular languages...

Viewing expressions and programs as strings, the following are not regular:

–
– The set of arithmetic expressions.
– The set of Boolean expressions.
– The set of regular expressions.
– The set of Python programs.

Theorem
We can convert an NFA N that does not have epsilon transitions into an equivalent DFA M.

"Subset" Construction
Given NFA $N = (Q, \Sigma, \delta, q_0, F)$ without ϵ-transitions the subset construction builds a DFA M:

– every set $X \subseteq Q$ is a state of M,
– the alphabet is Σ,
– for a state $X \subseteq Q$ of M, define $\delta'(X, a) = \bigcup_{q \in X} \delta(q, a)$,
– the initial state of M is the set $\{q_0\}$,
– for a state $X \subseteq Q$ of M, put X into F' if X contains an accepting state of N.

How to show that a language is not regular

To show that a language is regular, it is sufficient to find a DFA, NFA, or Regular Expression for it.

But how do we show a language L is not regular?

One must show that there is no DFA that recognises L.

Equivalently... for every DFA D there is a string x that witnesses that $\text{Lang}(D) \neq L$:

– either $x \in \text{Lang}(D) \setminus L$
– or $x \in L \setminus \text{Lang}(D)$

The language $L = \{a^n b^n : n \geq 1\}$ is not regular.

Proof:

1. Let D be a DFA. We will argue there must be a witness that shows $\text{Lang}(D) \neq L$.
2. If D repeatedly reads as it must eventually return to a state it has been in before. Why?
3. So there are two different strings a^i and a^j that reach the same state of D.
4. Now look at the input string $a^i b^i$. Two possible cases...
   (a) D rejects $a^i b^i$. Then $a^i b^i$ is a witness.
   (b) D accepts $a^i b^i$. Then D also accepts $a^j b^i$. Why? And so $a^j b^i$ is a witness. Why?

Another technique

Once we know that a language is not regular, we can deduce that any language that is a "combination" of regular languages is regular. We can try show that a language L is not regular using a proof technique called proof of a negation (see Tutorial on Assumed Knowledge).

Proof of a negation:

> We want to prove that a statement P is false.
>
> 1. We assume that P is true.
> 2. We arrive at an impossible situation (incorrect conclusion).
> 3. Conclude the statement P must in fact be false.

We want to prove that a particular language L_1 is not regular.

> 1. Assume L_1 is regular.
> 2. Express some other language L_2 which we know is not regular (because we've proved this already) as a combination of regular languages (including L_1) using union, intersection, complementation, cat, and star.
> 3. This in an impossible situation (the 'contradiction').
> 4. Conclude L_1 cannot be regular.

Recall

We are interested in models that specify computational problems:

1. English or Mathematics (week 1).
2. Regular expressions (week 2)
3. Automata (weeks 3 - 4)
4. Context-free grammars (weeks 5 - 6)
5. Turing-machines (weeks 7-9)

Regular expressions vs context-free grammars

Regular expressions:

- Useful for basic pattern matching, e.g., recognising keywords.
- But they are limited.
- The basic difficulty is handling arbitrary nesting, (), (()), ((())), etc. And in fact, the language of balanced parentheses is not regular.

Context-free grammars:

- Naturally describe the structure of most programming languages.
- Form the basis for translating between different representations of programs.

Context-free grammars in a nutshell

A grammar is a set of rules which describes a language.

The rules are used to generate (aka derive) strings by rewriting.

The rules in a context-free grammar can be thought of as a recursive description of the set of strings.

$$S \to aSb$$
$$S \to T$$
$$T \to c$$

A context-free grammar is made of:

- Variables S, T
- Terminals a, b, c (like "input characters")
- Rules (three in this case)
- Start variable S

You can read the arrow $\to$ as produces or rewrites to

$$S \to aSb$$
$$S \to T$$
$$T \to c$$

The grammar derives strings (of terminals) by the following **steps**:

1. Write the start variable
2. Repeat the following until no variables are written down:
   2.1 Pick a variable X that is written down
   2.2 Pick a rule $X \to \ldots$
   2.3 Replace X with the right-hand side of that rule.
3. The string that remains is derived by the grammar

$$S \to aSb$$
$$S \to T$$
$$T \to c$$

- $S \Rightarrow T \Rightarrow c$
- $S \Rightarrow aSb \Rightarrow aTb \Rightarrow acb$
- $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaTbb \Rightarrow aacbb$
- $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaaTbbb \Rightarrow aaacbbb$
- $\ldots$

Each of these lines is called a derivation. The string of terminals is derived.

$$S \to aSb$$
$$S \to T$$
$$T \to c$$

The language generated by a grammar G is all the strings (of terminals) that can be derived by the grammar, and is written $\text{Lang}(G)$; we also call $\text{Lang}(G)$ the language of G.

In this example, $\text{Lang}(G) = \{a^n c b^n : n \geq 0\}$.

Notation.

- $\Rightarrow$ means "derives in 1 step" (sometimes "yields")
- $\overset{n}{\Rightarrow}$ means "derives in n steps"
- $\overset{*}{\Rightarrow}$ means "derives in zero or more steps"
- $\overset{+}{\Rightarrow}$ means "derives in one or more steps"

Definition
The set of strings over Σ that are derived from the start variable is called the language generated by G (aka language of G):

$$\text{Lang}(G) = \{u \in \Sigma^* : S \overset{+}{\Rightarrow} u\}$$

A language is called context-free if it is generated by a context-free grammar.

Shorthand notation

A variable can have many rules:

$$S \to aSb$$
$$S \to T$$

They can be written together:

$$S \to aSb \mid T$$

Tips for designing CFGs

1. Variables generate substrings with similar properties.
   - Think of the variables as storing information, or as having meaning.
2. Think recursively.
   - How can a string in the language be built from smaller strings in the language?
   - Make sure you cover all cases.

$$S \to S - S \mid x \mid y \mid z$$

Rightmost derivations of $x - y - z$

$$\begin{aligned} S &\Rightarrow S - S \\ &\Rightarrow S - z \\ &\Rightarrow S - S - z \\ &\Rightarrow S - y - z \\ &\Rightarrow x - y - z \end{aligned}$$

$$\begin{aligned} S &\Rightarrow S - S \\ &\Rightarrow S - S - S \\ &\Rightarrow S - S - z \\ &\Rightarrow S - y - z \\ &\Rightarrow x - y - z \end{aligned}$$

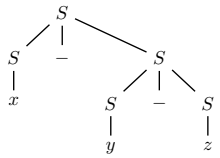
Parse Tree

$$S \to S - S \mid x \mid y \mid z$$

A parse tree of a string w is a tree labeled by variables and terminal symbols:

- root is labeled by the start variable
- interior node is labeled by a variable
- children of a node labeled X are labeled by the right hand side of a rule $X \to u$, in order.
- leaf nodes is labeled by a terminal or ϵ
- w read off leaves from left to right

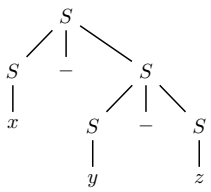
Parse tree of $w = x - y - z$



From parse trees to right-most derivation:

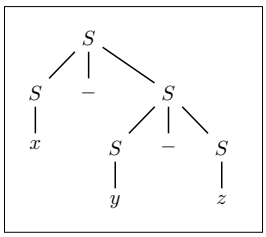
$$S \to S - S \mid x \mid y \mid z$$

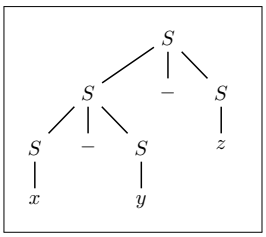
string $x - y - z$



$$\begin{aligned} S &\Rightarrow S - S \\ &\Rightarrow S - S - S \\ &\Rightarrow S - S - z \\ &\Rightarrow S - y - z \\ &\Rightarrow x - y - z \end{aligned}$$

Parse trees important because they give "meaning" to strings in the sense that each subtree can be seen as a unit or grouping.



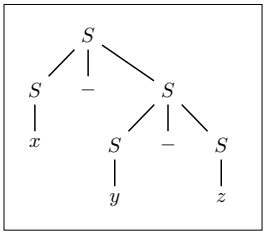


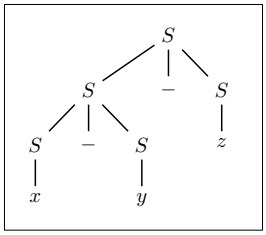
Parse trees give "meaning" to strings

$$S \to S - S \mid x \mid y \mid z$$

The string $x - y - z$ has two parse trees corresponding to different order of operations.

One means "$x - (y - z)$" and the other means "$(x - y) - z$"





Ambiguous grammars

Definition
For a grammar G:

- A string is ambiguous if it has ≥ 2 parse trees in G.
- The grammar G is ambiguous if it generates at least one ambiguous string.

E.g., the grammar $S \to S - S \mid x \mid y \mid z$ is ambiguous.

How to see if a string is ambiguous without drawing parse trees?

- A derivation is called rightmost if at each step it replaces the rightmost symbol.
- Fact: One can pair rightmost derivations and parse trees (one-to-one correspondence)
- So, a string is ambiguous if it has at least two rightmost derivations.

The same statements above hold with "leftmost" instead of "rightmost"

Context-Free Languages

Definition
A language is context-free if it is generated by a CFG.

Easy facts.

- The union of two CFL is also context-free.
  Why? Just add a new rule $S \to S_1 \mid S_2$ where S_i is the start symbol of grammar i.
- The concatenation of two CFL is also context-free
  Why? Just add a new rule $S \to S_1 S_2$
- The star closure of a CFL is also context-free
  Why? Just add a new rule $S \to SS_1 \mid \epsilon$

This implies that every regular language is context-free (Why?)

Definition of a context-free grammar

Definition. A context-free grammar G consists of four items:

1. Variables V, aka non-terminals
   $A, B, C, \ldots$
2. Terminals Σ, aka input characters
   $a, b, c, \ldots, 0, 1, 2, \cdots, +, -, (,), \cdots$
3. Rules R
   $A \to u$ where u is a string of variables and terminals.
4. Start variable
   usually S, or the first one listed.

$$G = (V, \Sigma, R, S)$$

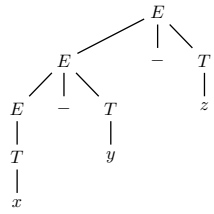
Removing ambiguity

- Suppose we want to change the grammar so that $x - y - z$ always means $(x - y) - z$, but we don't want to add parentheses to the strings
- Introduce a new nonterminal symbol T:

$$E \to E - T \mid T$$
$$T \to x \mid y \mid z$$

- Now the only rightmost derivation of $x - y - z$ is:

$$\begin{aligned} E &\Rightarrow E - T \\ &\Rightarrow E - z \\ &\Rightarrow E - T - z \\ &\Rightarrow E - y - z \\ &\Rightarrow T - y - z \\ &\Rightarrow x - y - z \end{aligned}$$



Good to know

The Chomsky Hierarchy consists of 4 classes of grammars, depending on the type of production rules that they allow:

Type 0 (Turing recognisable)	$z \to v$
Type 1 (context-sensitive)	$uAv \to uzv$
Type 2 (context-free)	$A \to u$
Type 3 (regular)	$A \to aB$ and $A \to a$

- u, v, z string of variables and terminals, z not empty.

Good to know

- $\{ww : w \in \{a, b\}^*\}$ is not context-free (Sipser Chapter 2.3)
- Let's look at a Type 0 grammar for it.

$$S \to aAS \mid bBS \mid T$$
$$Aa \to aA$$
$$Ab \to bA$$
$$Ba \to aB$$
$$Bb \to bB$$
$$AT \to Ta$$
$$BT \to Tb$$
$$T \to \epsilon$$

Derive $aabaab$:

$S \Rightarrow aAS \Rightarrow aAaAS \Rightarrow aAaAbBS \Rightarrow aAaAbBT$
$\Rightarrow aAabABT \Rightarrow aaAbABT \Rightarrow aabAABT$
$\Rightarrow aabAATb \Rightarrow aabATab \Rightarrow aabTaab \Rightarrow aabaab$

- A context-free grammar (CFG) generates strings by rewriting.
- Today we will see an algorithm that tells if a given context-free grammar (CFG) generates a given string.
- This basic problem is solved by compilers and parsers every day.

Membership problem for CFG: Given a CFG G and string w decide if G generates w.[1]

Parsing problem for CFG: Given a CFG G and string w produce a parse-tree for w or say there is none.

[1] Might also call this the "Derivation problem for CFGs".

Normal forms

An object can have more than one representation. A normal form is a "standard" or "canonical" representation of objects.

Example: lines in high-school geometry have normal forms:
- General form $Ax + By + C = 0$
- Gradient-intercept form $y = mx + b$

Converting an object into a normal form is sometimes used as a "preprocessing" step that makes it easier to write and understand algorithms.

You will see important normal forms in Databases and Logic.

Chomsky Normal Form

Definition. A grammar is in Chomsky Normal Form (CNF) if every rule is in of one of these forms:

1. $A \to BC$
   Here A, B, C are any non-terminals, except that neither B nor C is the start non-terminal
2. $A \to a$
   A is any non-terminal and a is a terminal
3. In addition, we permit $S \to \varepsilon$ where S is the start non-terminal

Note.
- We disallow rules like $A \to 0A1$ and $A \to B$ and $A \to ab$
- Parse trees of a grammar in CNF are binary trees (i.e., every internal node has 1 or 2 children).

$$T \to aTb \mid \epsilon$$

$$\begin{aligned} S &\to AX \mid \epsilon \\ T &\to AX \\ X &\to TB \mid b \\ A &\to a \\ B &\to b \end{aligned}$$

Equivalent grammar in CNF.

If G is in CNF then every non-empty string $w \in \mathrm{Lang}(G)$ can be derived in $2|w| - 1$ steps.

Why? Every derivation of a non-empty string must make $|w|$ steps of the form $A \to a$. But the only way to get these $|w|$ many variables is by rules $A \to BC$ which add one variable at a time; so we need at least $|w| - 1$ such steps since each adds just one variable (and we start with one variable). Note that we also can't have more than this many steps, because there is no rule to delete a variable.

So... we can just search through all derivations of length $2|w| - 1$ and see if any produces w.

Q: How many (rightmost) derivations with N steps are there?

A: In general, at least 2^N, because at each step we may have two or more possible rules to fire for the (rightmost) variable... This means listing all these derivations is slow.

Later today we will see a polynomial time algorithm.

How can we turn a CFG into CNF?

Let's give names to some of the rules we want to eliminate.

1. rule of the form $A \to \epsilon$ is called an epsilon rule.
2. rule of the form $A \to B$ is called a unit rule.

Given a grammar G:

1. We transform G into an equivalent grammar that has no epsilon rules (except perhaps $S \to \epsilon$).
2. We transform G into an equivalent grammar that has no unit rules.

We do each of these by removing the offending rules and then (repeatedly) adding some rules to the rule set R in order to compensate...

Remove epsilon rules: idea

The idea is to **inline:**

$$\text{if } B \to uAv \text{ and } A \overset{+}{\Rightarrow} \epsilon \text{ then also } B \overset{+}{\Rightarrow} uv$$

Here u, v are strings of terminals and non-terminals.

1. Find the 'nullable' variables, i.e., $A \overset{+}{\Rightarrow} \epsilon$ (tutorial)
2. Remove from R all rules of the form $A \to \epsilon$
3. If $B \to uAv$ is in R and A is nullable, then add the rule $B \to uv$ to R (unless $uv = \epsilon$).
4. Repeat Step 3 until no new rules are added.
5. Add $S \to \epsilon$ to R if $S \overset{+}{\Rightarrow} \epsilon$

Note: This procedure may add unit rules.

Remove epsilon rules: example 1

$$A \to \epsilon$$
$$B \to AA$$
$$C \to aB$$

$$A \to \epsilon$$
$$B \to AA \mid A$$
$$C \to aB$$
$$C \to a$$

- Nullable: A, B
- Remove $A \to \epsilon$
- Add rule: $C \to a$
- Add rule: $B \to A$

Remove epsilon rules: example 2

$$S \to ABA \mid \epsilon$$
$$A \to aA \mid \epsilon$$
$$B \to bB \mid \epsilon$$

$$S \to ABA \mid AB \mid BA \mid AA \mid A \mid B \mid \epsilon$$
$$A \to aA \mid a$$
$$B \to bB \mid b$$

- Nullable: S, A, B
- Remove $S \to \epsilon, A \to \epsilon, B \to \epsilon$
- Add rules: S produces AB, BA, AA, A, B
- Add rule: $A \to a$
- Add rule: $B \to b$

Removing unit rules: idea

The idea is to **inline:**

If $A \overset{+}{\Rightarrow} B$ and $B \to u$ then also $A \overset{+}{\Rightarrow} u$

Here u is a strings of terminals and non-terminals.

1. Find all pairs A, B such that $A \overset{+}{\Rightarrow} B$ (tutorial)
2. Remove all unit rules from R.
3. If $A \overset{+}{\Rightarrow} B$ and $B \to u$ is in R then add $A \to u$ to R
4. Repeat Step 3 until no new rules are added.

Removing unit rules: example

$$S \to AB \mid A \mid B$$
$$A \to aA \mid a$$
$$B \to b$$

$$S \to AB \mid aA \mid a \mid b$$
$$A \to aA \mid a$$
$$B \to b$$

- Find that $S \overset{+}{\Rightarrow} A$, $S \overset{+}{\Rightarrow} B$
- Remove $S \to A, S \to B$
- Add $S \to aA \mid a$ and add $S \to bB \mid b$

Chomsky Normal Form

Theorem
Every context-free language is generated by a grammar in CNF.

In the next slides, we will give a 5-step algorithm to do this:

1. START: Eliminate the start variable from the RHS of all rules
2. TERM: Eliminate rules with terminals (except of form $A \to a$)
3. BIN: Eliminate rules with more than two variables
4. EPSILON: Eliminate epsilon rules (except $S \to \epsilon$)
5. UNIT: Eliminate unit rules

Chomsky Normal Form: algorithm

START: Eliminate the start variable from the RHS of all rules

- Add a new start variable S and the rule $S \to T$ where T was the old start variable.

Chomsky Normal Form: algorithm

TERM: Eliminate rules with terminals (except of form $A \to a$)

- Replace every terminal a on the RHS of a rule (that is not of the form $A \to a$) by the new variable N_a.
- For each such terminal a create the new rule $N_a \to a$.

Chomsky Normal Form: algorithm

BIN: Eliminate rules with more than two variables

For every rule of the form $A \to DEFGH$, say, delete it and create new variables A_1, A_2, A_3 and add rules:

$$A \to DA_1$$
$$A_1 \to EA_2$$
$$A_2 \to FA_3$$
$$A_3 \to GH$$

Chomsky Normal Form: algorithm

EPSILON: Eliminate epsilon rules (except $S \to \epsilon$)

- Use the procedure from before.
- This will not mess up the results of START, TERM, BIN since it only removes symbols from the right-hand-side of some rules. So:
  - it will not add the start variable to the RHS of a rule
  - it will not add rules with terminals (since the only rules with terminals are of the form $A \to a$)
  - it will not add rules with more than two variables
- It may introduce unit rules.

Chomsky Normal Form: algorithm

UNIT: Eliminate unit rules

Use the procedure from before.

- This will not mess up START,TERM, BIN since it only adds rules that change the left-hand-sides of some rules.
- It will also not mess up EPSILON, i.e., it will not add epsilon rules.
  - Why? because to add $A \to \epsilon$ we need that $S \to \epsilon$ (since this is the only possible epsilon rule now) and $A \overset{+}{\Rightarrow} S$, but because of START we know that S does not occur on the right-hand-side of any rule.

Chomsky Normal Form: example

$$T \to aTb \mid \epsilon$$

1. START: Eliminate start variable from the RHS of all rules:

$$S \to T$$
$$T \to aTb \mid \epsilon$$

Chomsky Normal Form: example

$$S \to T$$
$$T \to aTb \mid \epsilon$$

2. TERM: Eliminate rules with terminals (except form $A \to a$):

$$S \to T$$
$$T \to ATB \mid \epsilon$$
$$A \to a$$
$$B \to b$$

Chomsky Normal Form: example

$$S \to T$$
$$T \to ATB \mid \epsilon$$
$$A \to a$$
$$B \to b$$

3. BIN: Eliminate rules with more than two variables:

$$S \to T$$
$$T \to AX \mid \epsilon$$
$$X \to TB$$
$$A \to a \qquad B \to b$$

Chomsky Normal Form: example

$$S \to T$$
$$T \to AX \mid \epsilon$$
$$X \to TB$$
$$A \to a \qquad B \to b$$

4. EPSILON: Eliminate epsilon rule $T \to \varepsilon$

$$S \to T \mid \epsilon$$
$$T \to AX$$
$$X \to TB \mid B$$
$$A \to a \qquad B \to b$$

Chomsky Normal Form: example

$$S \to T \mid \epsilon$$
$$T \to AX$$
$$X \to TB \mid B$$
$$A \to a \qquad B \to b$$

5. UNIT: Eliminate unit rules (e.g., first $S \to T$, then $X \to B$)

$$S \to AX \mid \epsilon$$
$$T \to AX$$
$$X \to TB \mid b$$
$$A \to a \qquad B \to b$$

Chomsky Normal Form: example

All done!

$$S \to AX \mid \epsilon$$
$$T \to AX$$
$$X \to TB \mid b$$
$$A \to a \qquad B \to b$$

Important idea: to decide if G can derive w we will build a table that stores, for every non-terminal V and every contiguous substring x of w, whether or not V derives x.

The reason for this is that if we know that A derives u and that B derives v, and if $V \to AB$ is a production rule, then we can conclude that V derives uv.

Define a procedure *table*(var, i, j) that decides if G can derive $w_i \cdots w_j$ starting with the nonterminal var.

Then we just check *table*$(S, 0, len(w) - 1)$.[2]

We can also write $V \in$ *table*(i, j) instead of *table*(V, i, j).

We can draw *table* as a triangle...

[2] If $w = \epsilon$ we just check if $S \to \epsilon$ is a rule.

$$
\begin{aligned}
S &\to AB \mid AX \mid \epsilon \\
T &\to AB \mid AX \\
X &\to TB \\
A &\to a \\
B &\to b
\end{aligned}
$$

S,T			
∅	X		
∅	S,T	∅	
A	A	B	B

$w = aabb$

(0,3)			
(0,2)	(1,3)		
(0,1)	(1,2)	(2,3)	
(0,0)	(1,1)	(2,2)	(3,3)

E.g., the only variable that derives $w[1,3] = abb$ is X
E.g., no variable derives $w[0,2] = aab$

$$
\begin{aligned}
S &\to AB \mid AX \mid \epsilon \\
T &\to AB \mid AX \\
X &\to TB \\
A &\to a \\
B &\to b
\end{aligned}
$$

?	?	?	?

$w = aabb$

$S \to AB \mid AX \mid \epsilon$
$T \to AB \mid AX$
$X \to TB$
$A \to a$
$B \to b$

A	A	B	B

$w = aabb$

Fill in each successive cell with an A if there is a rule of the form $A \to BC$ and certain earlier entries are filled with B and C...

A?			
	C		
B			

(0,3)			
(0,2)	(1,3)		
(0,1)	(1,2)	(2,3)	
(0,0)	(1,1)	(2,2)	(3,3)

A?			
B		C	

(0,3)			
(0,2)	(1,3)		
(0,1)	(1,2)	(2,3)	
(0,0)	(1,1)	(2,2)	(3,3)

A?			
B			
			C

(0,3)			
(0,2)	(1,3)		
(0,1)	(1,2)	(2,3)	
(0,0)	(1,1)	(2,2)	(3,3)

$$S \to AB \mid AX \mid \epsilon$$
$$T \to AB \mid AX$$
$$X \to TB$$
$$A \to a$$
$$B \to b$$

?	?	?	
A	A	B	B

$w = aabb$

$$S \to AB \mid AX \mid \epsilon$$
$$T \to AB \mid AX$$
$$X \to TB$$
$$A \to a$$
$$B \to b$$

∅	S,T	∅	
A	A	B	B

$w = aabb$

$$S \to AB \mid AX \mid \epsilon$$
$$T \to AB \mid AX$$
$$X \to TB$$
$$A \to a$$
$$B \to b$$

?	?		
∅	S,T	∅	
A	A	B	B

$w = aabb$

$S \to AB \mid AX \mid \epsilon$
$T \to AB \mid AX$
$X \to TB$
$A \to a$
$B \to b$

∅	X		
∅	S,T	∅	
A	A	B	B

$w = aabb$

$S \to AB \mid AX \mid \epsilon$
$T \to AB \mid AX$
$X \to TB$
$A \to a$
$B \to b$

?			
∅	X		
∅	S,T	∅	
A	A	B	B

$w = aabb$

$S \to AB \mid AX \mid \epsilon$
$T \to AB \mid AX$
$X \to TB$
$A \to a$
$B \to b$

S,T			
∅	X		
∅	S,T	∅	
A	A	B	B

$w = aabb$

Good to know

What if I want to compute a derivation/parse-tree?

When computing the entries of the table, store the needed information.

Idea: if A is in *table*(i, j) store a rule $A \to BC$ and a number k (the "split") that witnesses why A was put into *table*(i, j).

The "split" k means we will find B in *table*(i, k) and C in *table*$(k + 1, j)$.

Motivation

- Turing Machines are a simple but powerful model of computation.
- A "general purpose" programming language means expressively equivalent to a Turing Machine.
- A "computable" problem means implementable by a TM.
- The simplicity of Turing Machines helps computer scientists reason about what problems can or cannot be solved by algorithms.

Turing Machines in a Nutshell

A Turing Machine (TM) M is a program that:

- Has a state variable taking finitely many values.
  - This includes halting values that cause the program to return.
- Has a bi-infinite tape, made of cells.
  - A cell can be blank or contain a character (aka symbol).
  - there is a single pointer (aka head) on the tape.
  - the pointer can move left or right one step or stay still.
  - the pointer can read and write symbols from a finite tape alphabet.

(Continued)

- Has transition rules from the current state + current cell contents to:
  - a new cell value to replace current cell with (which may be unchanged)
  - a direction to move the head (left/right/stay)
  - a new state for the head (which may be unchanged)

Inputs, Special States & Outputs

Inputs for Turing Machines are defined as follows:

- The Input Alphabet Σ is a subset of the Tape Alphabet, excluding blanks.
- An Input String of the Input Alphabet is written on a tape with infinitely many blanks on each side. We call this tape an Input Tape

There are also special states:

- An initial state, accept state and optionally a reject state.
- The accept and reject states are called halting states.
- There are no transition rules from halting states.

If a Turing Machine halts, its output is the final contents of the tape.

Specifying Turing Machines: Morphett Notation

For these lectures we use a standard notation that can be pasted into Morphett's Turing Machine Simulator

⟨current state⟩	⟨current cell⟩	⟨new cell⟩	⟨direction⟩	⟨new state⟩

E.g., **q 0 1 R p** means *"if we are in state q and a 0 is under the head: replace it with a 1, go right and change the head's state to p"*

A Turing Machine can be specified by a list of these transition rules.

Only ⟨current state⟩ and ⟨current cell⟩ is need to determine which Turing Machine transition to use.

Conventions for lectures and assessments:

- The first line of the TM transition rules defines the initial state — i.e., the line's ⟨current state⟩ argument[1].
- A state is an alphanumeric word, e.g., **q**, **doX**, **2**, etc — it is case-sensitive.
- The blank symbol is written as an **underscore**.
- The directions are **L** for left, **R** for right and ***** for stay.
- The halting states are states that start with **halt**, e.g., **halt-accept** and **halt-reject**.

[1] In the simulator the initial state is called 0 by default but can be changed in the "advanced settings"

Morphett $*$ Notation

⟨current state⟩	⟨current cell⟩	⟨new cell⟩	⟨direction⟩	⟨new state⟩

The $*$ symbol has a special meaning in the ⟨current cell⟩ and ⟨new cell⟩ arguments:

- Writing ***** for ⟨current cell⟩ means this transition applies to any cell we don't already have a rule for (with respect to ⟨current state⟩) [2]

  E.g., **q * 1 r q** means *"if we are in state q and there are no existing rules for the current cell under the head, turn into a 1, move right and stay in state q"*

- ***** for ⟨new cell⟩ means don't change ⟨current cell⟩.

[2] It doesn't matter if this is done above or below any other rules.

Example II

This Turing Machine scans binary strings and **does nothing**.

```
1 q 0 * R q
2 q * * R q
3 q _ _ * halt-accept
```

Why?

- **Line 1** applies if the state is q and the current cell is 0, and leaves the cell unchanged.
- **Line 3** applies if the state is q and the current cell is blank; it leaves the cell blank, stays, and accepts.
- **Line 2** only applies if the state is q and the current cell is 1 (because $q + 0$ and $q +$ _ already have rules) and also leaves the cell unchanged.

Recognising Languages & Deciders

The language recognised by Turing Machine M (or language of M) is the set of input strings M accepts.

A language is Turing-recognisable (aka recognisable[3]) if a TM recognises it.

A decider is a Turing Machine that halts on all inputs.

A language is Turing-decidable (aka decidable[4]) if a decider recognises it.

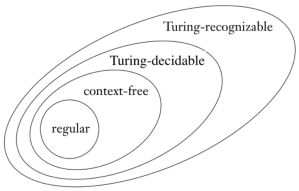
If a decider recognises a language, we additionally say it decides that language.

[3] also computably enumerable, recursively enumerable
[4] also computable, recursive

Recognising Languages & Deciders

In Week 9 we will show:

- There is a language that is not recognisable.
- There is a recognisable language that is not decidable.



What is the right level of detail to describe Turing Machines?

1. High-level description.
   - English description describing an algorithm.
   - No mention of the Turing Machine tape or head.
   - This is how you do it in COMP2123 (Data Structures and Algorithms)
2. Implementation description.
   - English description of the way the Turing Machine moves its head and marks data on the tape **and why**.
   - There are many good examples of this in Sipser [2] (the recommended reading).
3. Formal description.
   - Lowest level of detail.
   - Specify the states and transitions in a table or diagram (e.g., in Morphett notation).

Redoing Example I

Task: Switch 0s and 1s in a binary string, e.g., $01101 \to 10010$.

1. High-level description.

   Loop over the string replacing 0s for 1s and 1s for 0s.

2. Implementation description.

   Move rightwards along the binary string on the tape, at each step replacing each bit by its opposite. Once the head sees a blank, there's nothing left to do, so we halt accept.

3. Formal description.

```
q 0 1 R q
q 1 0 R q
q _ _ * halt-accept
```

Visualising Transition Rules with State Diagrams

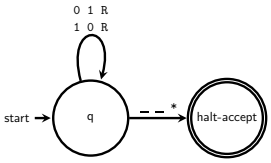


Figure 2: A State Diagram for Example I.

Example III: Incrementor

Task: Add 1 to a binary string, e.g., $10101 \to 10110$ or $111 \to 1000$.

1. High-level description.

   Add 1 to the last bit of the input string s and "carrying the 1" leftwards if necessary to complete the addition.

2. Implementation description.

   Move right to the end of the input string s, then at each step, moving leftwards, we add a 1 by either:
   - *turning a 0 to a 1 and accepting.*
   - *turning 1 to 0 (i.e., "carrying").*
   - *turning the blank before the string to 1 and accepting.*

Example III: Incrementor

3. Formal description.

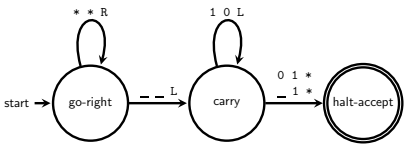


Figure 3: State diagram for an Incrementor Turing Machine.

(Or the Morphett-notation program on the next slide)


```
1 go-right 0 0 R go-right
2 go-right 1 1 R go-right
3 go-right _ _ L carry
4
5 carry 0 1 L halt-accept
6 carry 1 0 L carry
7 carry _ 1 L halt-accept
```

Tips for designing Turing Machines

1. States contain information about what has been done or what needs to be done. So use informative names for states.
2. Break your problem down: design from high level descriptions, to implementation descriptions then to formal descriptions.
3. View the tape as having multiple tracks, and a tape alphabet whose symbols contain multiple pieces of information each stored on a different track.

Note: We already used tips 1 & 2 in our incrementor solution.

Tip (1/3): Using informative names for states

Writing state names informatively, e.g., like you do for function and variable names when you program.

This helps with design and debugging.

For example, in Example III (Incrementor) our states were called "go-right", "carry" and "halt-accept".

Tip (2/3): Break your problem down

Case Study: Here is a computational problem.

Input: string s over the alphabet $\{0, 1\}$

Output: decide if the string is of the form ww for some string $w \in \{0, 1\}^*$

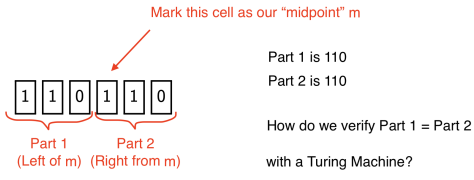
High-Level Description of a solution:

1. Check the string is even length and halt reject if it isn't.[5]
2. Find and mark the midpoint m (Rounding up, e.g., the midpoint of 123456 is 4)
3. Check the input is of the form ww by matching the symbols to the left of m to symbols to the right from m.

[5] If the input is of odd length, it's impossible for it to be of the form ww

Focusing on the High Level step 3:

Example: for the input string 110110:



Tip (3/3): Using multitracked tapes

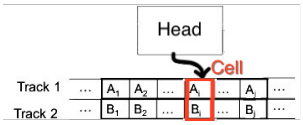


Figure 4: A Turing Machine with A-type info on track 1 and B-type info on track 2.

For example – In our case study:

- We use the input alphabets $\Sigma = \{0, 1\} \times \{m, b\}$, i.e., pairs (x, y) where x is 0, 1, and y is m or b.

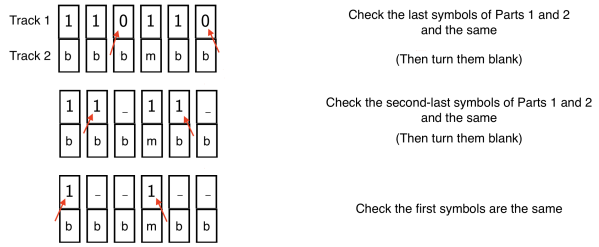
  b *is a placeholder symbol that means "not the midpoint" – Recall that we cannot have blanks in the input alphabet.*

- We use input tapes with:

  Track 1: the binary string we're testing
  Track 2: all b

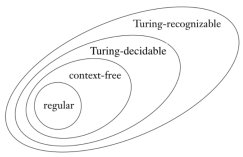
- Then, once we find the midpoint cell - we modify that cell's second track to m.

Implementation Level Description: Solving the following sub-problem of checking: Part 1 = Part 2. E.g.



The Power of Turing Machines

Recall: context-free grammars cannot recognise the language $\{ww \mid w \in \{0, 1\}^*\}$. Thus, Turing Machines are more powerful than context-free grammars.



Next week: we will show that all problems solvable by an algorithm are precisely the problems solvable by Turing Machines.

Week 9: we will show some problems cannot be solved by Turing Machines — or indeed, any algorithm.

Summary: Definition of a Turing Machine (1/3)

- Each cell of the tape consists of a symbol from a finite tape alphabet, which includes a blank symbol.
- The Turing Machine has a state out a finite number of possibilities.
- The Turing Machine has transition rules:

current state + current tape cell →

new tape cell + left/right/stay head movement + new state

Summary: Definition of a Turing Machine (2/3)

Inputs for Turing Machines are defined as follows:

- The Input Alphabet Σ is a subset of the Tape Alphabet, excluding blanks.
- An Input String of the Input Alphabet is written on a tape with infinitely many blanks on each side. We call this tape an Input Tape

There are also special states:

- An initial state, accept state and optionally a reject state.
- The accept and reject states are called halting states.
- There are no transition rules from halting states.

Summary: Definition of a Turing Machine (3/3)

- A Turing Machine M runs on input string u by applying M's transition rules starting with:
  - u on the input tape (blank everywhere else).
  - M's head on the left-most symbol of u
  - M in its initial state
- If M running on u reaches the accept state denotes an accepting computation, and reaching the reject state denotes a rejecting computation.
- A Turing Machine does not necessarily halt on all input tapes – it may run forever.
- **Note:** a Turing Machine cannot have missing transitions.[6]

[6] In the Morphett Simulator, if you want to exclude unreachable transition rules, you can put `* * * * halt-reject` at the bottom.

Tip (2/3): Using multitracked tapes

Tape alphabets symbols can encode multiple pieces of information.

Recall that for any two sets A and B, the product $A \times B$ consists of all tuples (a, b) where $a \in A$ and $b \in B$.

Hence, we can use a product $A \times B$ as part of our tape alphabet.

This means each cell can have A-type information and / or B-type information. We refer to A and B as tracks of the Turing Machine.

In this case, you can imagine the Turing Machine tape as a table; each row is a track, and the head processes 1 column at a time.

Three Big Ideas in Computer Science

These 2 ideas from week 1 are the subject of this lecture:

- *"Very few of the programming constructs and features we learn are actually needed for solving computational problems. Instead, they are there to make it easier to read/write programs."*
- *"Most programming languages can solve the same computational problems as Python. They are* **expressively equivalent** *to Python."*

The remaining big idea is the subject of next week's lecture:

- *"Some computational problems cannot be solved in Python or in any expressively equivalent language either."*

Note:

- The output of a Python program can be taken to be what it does to its input variables, i.e., the final contents of its input variables at termination.
- The output of a TM is what is written on the tape once it halts.
- The specific halting state of a TM is additional information, e.g., `halt-accept`/`halt-reject` is used for language recognition.

Computational Tasks, Algorithms and Turing Machines

The input/output behaviour of an algorithm, Python Program or Turing Machine X refers to what happens to every valid input.

I.e., whether or not X halts on that input, and if it does, what the corresponding output is.

The Statement

To demonstrate that Turing Machines are expressively equivalent to Python programs, we need to prove:

1. *"For any Turing Machine, there is a Python program with the same input-output behaviour"*.

   In this case, the Python Program simulates that Turing Machine.

2. *"For any program written in Python, there is a Turing Machine with the same input-output behaviour"*.

   In this case, the Turing Machine simulates that Python Program.

We will prove Turing Machines can simulate Python Programs by proving:

$$\text{Turing Machines} \xrightarrow{\text{Simulates}} \text{Tiny Python} \xrightarrow{\text{Simulates}} \text{Python}$$

We prove "Turing Machines $\xrightarrow{\text{Simulates}}$ Tiny Python" in two parts:

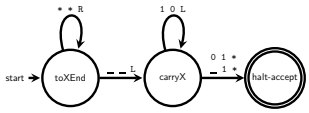
1. We implement Turing Machines to simulate each type of individual instruction of Tiny Python:
   - Incrementing, e.g., `x = x + 1`
   - Decrementing, e.g., `x = x - 1` (with `0 - 1 == 0`).
   - Loops of the form: `while x != 0:`

2. We demonstrate these Turing Machine implementations can be combined the same way that Tiny Python instructions can be combined, i.e.:

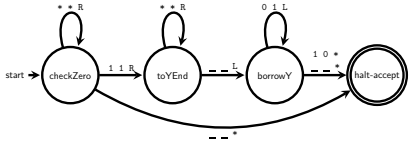
   Sequencing Executing one instruction after another

   Nesting Putting instructions inside a `while` loop.

E.g., if the Python instruction `x = x + 1` is simulated by the TM:



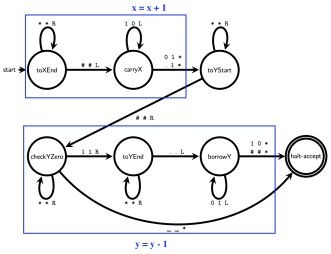
And the Python instruction `y = y - 1` is simulated by the TM:



We will show the program combining the instructions:

```
x = x + 1
y = y - 1
```

can be implemented by combining the TMs:



The instruction `x = x + 1` is a program that has input x and output $x + 1$.

The instruction `y = y - 1` is a program that has input y and output $y - 1$.

The program:

```
x = x + 1
y = y - 1
```

Has two inputs and two outputs.

How can our Turing Machine have two inputs and two outputs?

The combined TM will have input alphabet $\{0, 1, \#\}$ where $\#$ separates variables, i.e.,

- Input strings have the form $x\#y$ for binary numbers x and y
- The output tape has $(x + 1)\#(y - 1)$
- e.g., $110\#101 \rightarrow 111\#100$

Recall:

- We represent non-negative integers as binary strings, e.g.,

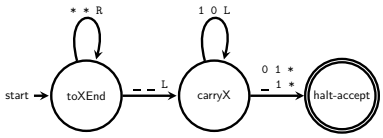
$$13 \quad = \quad 8 + 4 + 1 \quad = \quad (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \quad \rightarrow \quad 1101$$

- To increment, because $1 + 1 = 10$ in binary we carry the 1 leftwards.
  I.e., $1011 + 1 = 1100$.
- To decrement, because $10 - 1 = 1$ in binary, we borrow the 1 leftwards.
  I.e., $1100 - 1 = 1011$.

Note: leading 0s are not a problem, e.g., decrementing $10 \rightarrow 01$ is ok.

Incrementor

Recall our solution for `x = x + 1` from week 7:



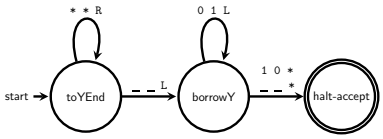
The actual incrementing happens entirely in the **carryX** state. However, it requires the head to be at the end (i.e., right-most symbol) of our input string `x`.

All **toXEnd** does is move the head to the end of `x`.

Decrementor

Here is an attempt at a decrementor, but it is not completely correct.

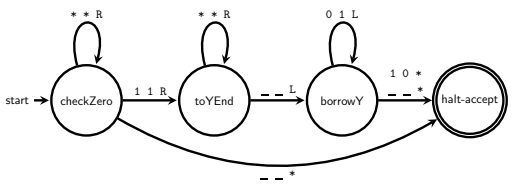
Question: what is the problem with this decrementor and how can we fix it?



Answer: it does not compute `0 - 1 = 0`, i.e., it turns $0 \to 1,\ 00 \to 11$, etc.

Solution: Before borrowing, check if the input string consists entirely of 0s.

We can achieve this by starting our implementation with transition rules that scan the input for any 1s, accepting if none are found.



Sequencing

We know how to implement Turing Machines to simulate `x = x + 1` and `y = y - 1`.

How about the combination?

```
x = x + 1
y = y - 1
```

Task: Create a Turing Machine with inputs of the form $x\#y$ where x, y are numbers written in binary. Output $(x+1)\#(y-1)$.

E.g., $11\#01 \to 100\#00$.

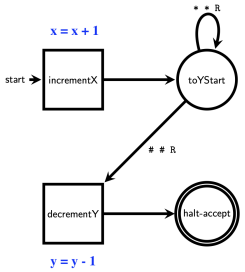
Implementation Description: the following allows us to reuse our TMs for incrementing (Slide 21) and decrementing (Slide 23).

For input $x\#y$

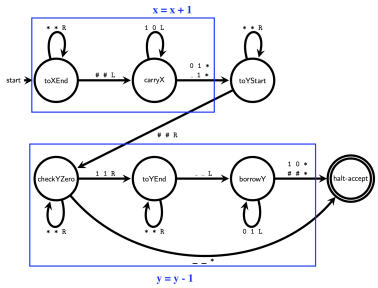
1. Scan to the start of x
2. Apply the incrementor to x
3. Scan to the start of y.
4. Apply the decrementor to y.

Note: in this case, step 1 can be ignored because the TM already starts at the beginning of x.

Formal Description: black-boxing the nodes specific to incrementing x and decrementing y:



Filling in the transitions for **incrementX** and **decrementY**:



Freeing up Tape

Note: The output tape for the TM implementing `x = x + 1` may have 1 more non-blank cell than the input!

E.g., for $x = 3$, $x + 1$ in binary has input/output behaviour $11 \to 100$.

Problem: for a TM implementing the combined program:

```
x = x - 1
y = y + 1
```

The tape usage of `y = y + 1` may increase and clash with `x = x - 1`

Solution: free up tape space between instructions as needed.

Tutorial Exercise: Create a Turing Machine that takes an input string $x\#y$ for strings $x \in \{0,1\}^*$ and $y \in \{0,1,\#\}^*$ and adds a single blank between $\#$ and y, i.e., returns $x\#\_y$.

E.g., adding space freeing to a TM implementing:

```
x = x - 1
y = y + 1
```

For input $x\#y$:

- After incrementing x the tape has $(x-1)\#y$.
- After tape freeing, the tape has $(x-1)\#\_y$.
- After incrementing y, the tape either has: $(x-1)\#(y+1)$ or $(x-1)\#\_(y+1)$.

Note: one can also more cleverly only free up tape as needed.

While Loops

To implement while loops, let's start with a simple example:

Task: Create a Turing Machine with inputs of the form $x\#y$ where x, y are numbers written in binary, and implement:

```
while x != 0:
    y = y + 1
    x = x - 1
```

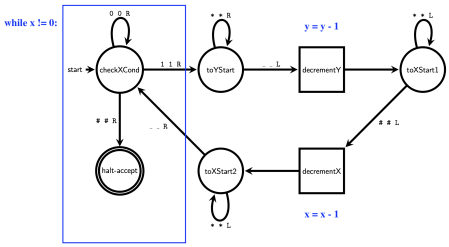

While Loops

```
while x != 0:
    y = y + 1
    x = x - 1
```

Implementation Description: for input $x\#y$

1. Check `x != 0` by scanning the binary string of x for a 1. If we find one before the $\#$, continue to 2, otherwise, accept.
2. Move to the start of y
3. Decrement y
4. Move to the start of x
5. Decrement x
6. Move to the start of x and repeat 1.

While Loops



Exercise: fill in the transitions for **decrementY** and **decrementX**.

Nesting

`y = y - 1` (i.e., the **decrementY** transitions) can be replaced with any loop body.

However, one has free up tape space between x and the loop body as necessary.

Summary

We now have Turing Machines to implement: incrementors, decrementors, while loops, combination (sequencing and nesting), thus:

$$\text{Turing Machines} \xrightarrow{\text{Simulates}} \text{Python}$$

In Slide 11 we showed

$$\text{Python} \xrightarrow{\text{Simulates}} \text{Turing Machines} \quad (1)$$

Combing the two results we conclude:

Python programs are expressively equivalent to Turing Machines.

The Church-Turing Thesis

In Part 1, we showed that Turing Machines and Python Programs (i.e., algorithms) are expressively equivalent.

> The Church-Turing Thesis states that for every algorithm A, its input/output behaviour can be simulated by a Turing Machine M_A.

The Church-Turing Thesis states that for every algorithm A, its input/output behaviour can be simulated by a Turing Machine M_A.

- Turing machines are a precise, formal equivalent of the intuitive notion of "algorithm".
- Any computational problem that can be solved in a general purpose programming language is also solvable by a Turing machine.

What is an Algorithm?

It's the Church Turing Thesis and not a theorem because it depends on how one defines "algorithm".

Typically, an algorithm is defined as a procedure P (aka effective method) satisfying properties such as:

1. P is set out in terms of a finite number of instructions.
2. P will produce the desired result in a finite number of steps.
3. P can (in practice or in principle) be carried out by a human being unaided by any machinery except paper and pencil.
4. P demands no insight, intuition, or ingenuity, on the part of the human being carrying out the method. [2]

Slide 41: All-known models of general computation are expressively equivalent to TMs, supports that:

Algorithms are at least as expressive as Turing Machines

Slide 42: Problems beyond Turing Machines don't appear to be algorithmic in the sense of Slide 40, supports that:

Algorithms are no more expressive as Turing Machines

Together, we get the Church-Turing Thesis that:

Algorithms and Turing Machines are expressively equivalent

Applications of the Church-Turing Thesis

For every level of Models of Computation studied we have studied in this course:

- Regular expressions and finite-state automata
- Context-free grammars (pushdown automata)
- Turing Machines

For each model of computation, we can pose the acceptance problem (aka the membership problem).

For example, in week 3, we constructed a Python Program with

Input: D (DFA) and w (string).

Output: decide whether D accepts w.

Applying the Church-Turing Thesis to that Python progam, we can conclude the DFA acceptance problem is Turing-decidable.

Note: it is also feasible to write an explicit (i.e., formal description of a) Turing Machine that solves the DFA acceptance problem by simulating DFAs.

Likewise, in week 6 we solved the membership problem for CFGs using the CYK algorithm.

Hence, the acceptance problem for context free grammars is also Turing-decidable.

Next week we will prove the acceptance problem for Turing Machines is not decidable (aka undecidable).

Universality

The Church-Turing Thesis proves that there are Universal Turing Machines (UTMs)

A Universal Turing Machine U takes as input the source code of a Turing Machine M and an input string s for M, and simulates M running on s.

I.e., U halts exactly when M does, and if M has an output on s, U has the same output.

We will use Universal Turing Machines to prove undecidability results next week.

This is feasible because almost any object can be encoded as a string, e.g.:

- Numbers $\rightarrow$ binary strings
- A pair of strings a and $b \rightarrow$ the string $a; b$
- A Turing Machine $M \rightarrow \text{Source}_M$, the source code of M.

For a UTM U, we are interested in inputs of the form $\text{Source}_M; s$.

Note: U must not accept inputs not of the form: $\text{Source}_M; s$.

Recall: The language recognised by Turing Machine M (or the language of M) is the set of input strings M accepts.

A decider is a Turing Machine that halts on all inputs.

Question 1: is a Universal Turing Machine U a decider?

Answer: (b) No. U does not halt on pairs $\text{Source}_M; s$ where M does not halt on s.

Question 2: what is the language of U?

Answer: $\{\ \text{Source}_M; s \mid M$ is a TM and s is an input that M accepts $\}$

Universality via the Church-Turing Thesis

1. The Morphett Simulator is a program X that takes as input a string encoding a Turing Machine M, e.g., the string:

```
flip 0 1 R flip
flip 1 0 R flip
flip _ _ R halt-accept
```

   And an input string s, and returns as output: the output tape of M running on s — with identical halting behaviour.

2. By the Church-Turing Thesis, there is a Turing Machine U that simulates X.

   U by definition is a Universal Turing Machine.

A challenge for constructing UTMs

Question The union of all Tape Alphabets across all Turing Machines is infinite — so how can one Turing Machine have a tape alphabet to interpret all others?

Answer Turing Machines are expressively equivalent to Binary Turing Machines, i.e., Turing Machines with Tape alphabet {`"0"`, `"1"`, `"_"`}. Hence, it suffices for our UTM to interpret Binary Turing Machines.

Binary Turing Machines are an example of a Turing Machine variant.

Universality by Hand

Theorem: Ordinary Turing Machines are expressively equivalent to Binary Turing Machines.

Proof Idea:

- For any Turing Machine M, there is a $k \in \mathbb{N}$ such that M's tape alphabet has less than 2^k symbols.
- We can rewrite M's tape into one organised into "chunks" of k cells — where each cell in the chunk contains a 0 or 1, and each chunk encodes a single character from M's alphabet.
- We then design a Binary Turing Machine M_{Bin} whose transition rules emulate M's transition rules at the chunk level.

Universality — Applications

In week 1 we discussed how Python and C are expressively equivalent because the Python interpreter is written in C and C compilers have been written in Python.

By the Church-Turing Thesis, a general purpose programming language P can intepret the source code of P programs.

This is an important concept called bootstrapping.

E.g., The compilers for Go and Haskell were originally written in C and now are written in Go and Haskell themselves. This makes language development easier.

Alan Turing discovered UTMs as early as 1937 to prove undecidability results — This will be the subject of next week's lecture.

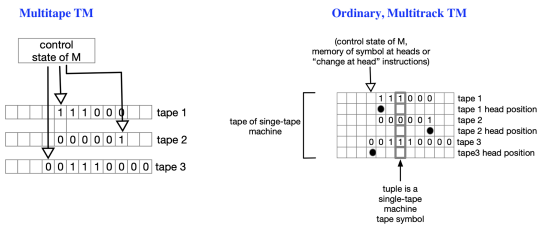
UTM variants: Mutlitape TMs

- A multitape TM has multiple tapes, each with its own head for reading and writing.
- Initially the input appears on tape 1, and the others start out blank.
- The heads move simultaneously (not necessarily same direction).
- The type of the transition function of a k-tape TM becomes a k-tuple of ordinary TM transition rules.
- Obviously every ordinary TM is a multitape TM (with $k = 1$).
- Is every multitape TM equivalent to a ordinary TM?

Theorem: Every multitape TM M has an equivalent ordinary TM B.

Proof Idea:

- Split the tape into $2k$-many tracks of the single tape of B.
- For each tape of M, use one track to store tape contents, and one track to mark head position on that tape.
- Each transition of M is simulated by a series of transitions of B.



Implementation level description of a multitape UTM U:

- Tape 1 holds the transition function δ_M of the input TM M.
- Tape 2 holds the simulated contents of M's tape.
- Tape 3 holds the current state of M, and the current position of M's tape head.
- U simulates M on input x one step at a time:
  - In each step, it updates M's state and simulated tape contents and head position as dictated by δ_M.
  - If ever M halts and accepts or halts and rejects, then U does the same

An Ordinary UTM

Here, we outline an ordinary Universal Turing Machine [1].

- Full explanation
- Loaded in Morphett's simulator

This Universal Turing Machine simulates any Binary Turing Machine — recall Binary Turing Machines are expressively equivalent to ordinary Turing Machines.

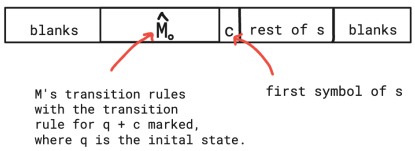
Implementation Level Description:

For an input Binary Turing Machine M and binary string s and steps $1, 2, 3, \ldots$ of M running on s, we write s_i for the tape at step i.

Then for any step i: we write a string $\widehat{M_i}$ and put it in between s_i to encode the following about M running on s:

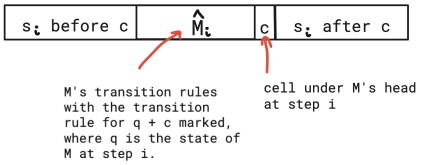
- a symbolic representation of M's transition rules
- a marking for the cell c in s_i that would be under M's head at step i.
- a marking for the state q that M is in — and hence the transition rule M would apply to q and c.

U's tape for Binary TM M running on binary string s at step 1:



(s_1 is input tape of s — i.e. s with blanks infinitely on both sides)

U's tape simulating Binary TM M running on binary string s at step i:



(Recall s_i is M's tape running s at step i)

The Universal Turing Machine U has transition rules to simulate the steps of M running on s.

I.e., for each simulated step i of M running on s, U progressively:

- Updates $\widehat{M}_i \to \widehat{M}_{i+1}$
- Updates $s_i \to s_{i+1}$
- Puts $\widehat{M}_{i+1}$ inside s_{i+1} to indicate the cell that would be under M's head at step $i+1$.

If at any step N, $\widehat{M}_N$ is in halting state, halt U accordingly.

Formal Description:

The Universal Turing Machine (UTM) U has tape alphabet A with 16 symbols:

`0 1 [ ] , : ! L R . - + # < >`

We will progressively explain how each of these symbols is used by U.

Let M be a Binary Turing Machine with **states** $q_0, q_1, \ldots, q_n$.

Each state q_{cur} has up to 3 transition rules for each cell that can be under M's head, i.e., `"_", "0", "1"`. We encode them comma separated as:

$$x_{cur}, y_{cur}, z_{cur}$$

x_{cur} is the encoded rule for q_{cur} + `"_"`, y_{cur} is the encoded rule for $q_{cur} + 0$, and z_{cur} is the encoded rule for $q_{cur} + 1$.

Note: some rules are non-existent and left blank e.g., when q_{cur} is a halting state.

For a symbol $c \in$ {`"0", "1", "_"`}: the transition rules x_{cur}, y_{cur} and z_{cur} have the following format:

⟨new cell⟩⟨direction⟩⟨change of index for new state⟩

For example:

Morphett Transition Rule	Encoding
q_3 c 1 L q_5	`1L++`
q_4 c 0 R q_1	`0R---`
q_1 c 1 L q_1	`1L.`
q_1 c _ L q_1	`_L.`

Continued:

Morphett Transition Rule	Encoding
$q_3\ c\ 1\ L\ q_5$	`1L++`
$q_4\ c\ 0\ R\ q_1$	`0R---`
$q_1\ c\ 1\ L\ q_1$	`1L.`
$q_1\ c\ \_\ L\ q_1$	`_L.`

In row 1 "++" indicates the transition rule updates state q_3 to q_5, i.e., 2 plus symbols to indicate the index going up by 2. Minus symbols are used for the state's index going down, and the full-stop for no change in the state's index.

Continued:

Morphett Transition Rule	Encoding
$q_3\ c\ 1\ L\ q_5$	`1L++`
$q_4\ c\ 0\ R\ q_1$	`0R---`
$q_1\ c\ 1\ L\ q_1$	`1L.`
$q_1\ c\ \_\ L\ q_1$	`_L.`

Note: for the encoding "_L.", a different symbol to _ should be used as that's the blank of the tape alphabet, which is not allowed in the input alphabet. However, this rule is ignored by the Morphett Simulator and UTM author for convenience.

To write $\widehat{M_i}$: If at step i, M is in state q_{cur} with symbol c under its head:

- We use the `:` symbol to join each state's encoded transition rules.
- We use the `!` symbol to mark M's transition rule for $q_{cur} + c$.

Putting it all together, $\widehat{M_i}$ has the form:

$$x_1, y_1, z_1 \ : \ x_2, y_2, z_2 \ : \ \ldots \ : \ x_{cur}, y_{cur}, z_{cur} \ ! \ \ldots \ : \ x_n, y_n, z_n :$$

Note: the remaining symbols of U's alphabet $\#$, $<$, $>$ are used during the running of U updating $\widehat{M_i} \to \widehat{M_{i+1}}$ and $s_i \to s_{i+1}$, etc.

Example:

U running on the input string:

`[ L+,0R.,1R.!1L+,1L+,0L.:,0L.,1L.:]1011`

Is equivalent to the following Turing Machine running on the input string 1011 with initial state $q1$.

```
; Go right (Initial state)
q1 _ _ L q2 ; L+
q1 0 0 R q1 ; 0R.
q1 1 1 R q1 ; 1R.!  - First transition rule to apply for 1011

; Carry
q2 _ 1 L q3 ; 1L+
q2 0 1 L q3 ; 1L+
q2 1 0 L q2 ; 0L.

; Go left then halt
q3 _ _ L halt-accept ; empty rule
q3 0 0 L q3          ; 0L.
q3 1 1 L q3          ; 1L.
```

Assessable: the definition of UTMs - e.g., when they accept, reject, halt, and encoding notation such as:

$$\text{Source}_M; w \text{ for an input string } w \text{ of a TM } M$$

- Meaning of the expressive equivalence between TMs and Python programs, ability to understand TM implementations in these slides.
- The expressive equivalence of multitape TMs to ordinary TMs (just what expressive equivalence between TM variants means, **but not explicitly how to convert between them**).
- Multitrack TMs, as these are just ordinary TMs where each cell value contains multiple pieces of information.
- Proving statements about the recognisability/decidability of languages, using properties of UTMs, and the Church-Turing Thesis.

Not Assessable:

- The exact proof of the fact that Python is expressively equivalent to TMs. **However, you should be very familiar with the meaning of the statement and the implications of it.**
- Knowledge about particular implementations of UTMs, e.g., those found in the reference slides (i.e., the specific multi-tape, and 1 track UTM constructions).

Undecidability in the Wild

Imagine you're an engineer at a software company.

Your users:

- want to run highly customisable jobs on your servers.
- want the general-purpose programming language Python to specify those jobs.

A product manager likes the idea but is concerned users will accidentally write programs that will run forever, wasting company resources.

The product manager approaches you and asks you to create an algorithm to analyse every user-program before it runs, and decide whether that program always halts.

If this algorithm exists, it would decide the language

$$L_{Python-Halt} = \{\text{Source}_P; w \mid P \text{ is a python program that halts on } w\}$$

After this lecture you will be able to argue that this language is undecidable, and therefore that this algorithm is impossible to create.

Converting a TM to a Python Function

For every TM M, there is a Python function `pyM` of type[1]:

```
pyM(w: str)-> bool
```

Such that for an input string w of M:

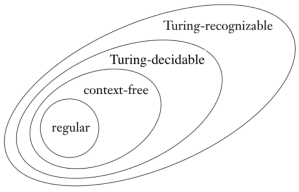
`pyM(w) == True`	$\iff$	M accepts w
`pyM(w) == False`	$\iff$	M rejects w
`pyM(w)` does not return	$\iff$	M does not halt

Note: M is a decider if and only if `pyM` returns on all inputs.

[1] Sometimes, for convenience if `pyM` runs on inputs of the form $\text{Source}_M; w$ we give `pyM` two arguments and write `pyM(Source`$_M$`, w)`; we can also consider all invalid inputs rejected.

Today we will show:

- There are languages that are not Turing-recognisable.
- There are languages that are Turing-recognisable but not Turing-decidable.



Where should we look for languages that are not decidable or not recognisable?

All the membership problems we've studied about automata, regular expressions, context-free grammars are decidable.

E.g.,

- We can decide if a regular expression R matches a string w (use automata!)
- We can decide if a CFG G generates a string w (use CYK!)

What about the acceptance problem for TMs?

Warmup

Are either of these Turing-recognisable or Turing-decidable?

- $L_{TM-accept} = \{\text{Source}_M; w : M \text{ is a TM that accepts } w\}$
- $L_{TM-not-accept} = \{\text{Source}_M; w : M \text{ is a TM that does not accept } w\}$

We know that $L_{TM-accept}$ is recognisable, but suspect it is not decidable.

We suspect that $L_{TM-not-accept}$ is not even recognisable.

We first prove that a slight variation of $L_{TM-not-accept}$ is not recognisable (and come back to it later).

A Non-Recognisable Language

Theorem

The language $L_{diag} = \{Source_M : M$ is a TM that does not accept $Source_M\}$ is not recognisable.

Proof: We will show for every TM B that

$$Lang(B) \neq L_{diag}$$

Which by definition means L_{diag} is not recognisable.

Note that: $Lang(B) \neq L_{diag}$ if and only if there is a witness to them being different, i.e., a string that is in one language but not the other.

We now show that the string $Source_B$ is such a witness.

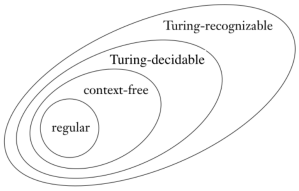
There are two cases: either $Source_B \in L_{diag}$ or not.

1. If $Source_B \in L_{diag}$, then B does not accept $Source_B$, so $Source_B \notin Lang(B)$.
2. If $Source_B \notin L_{diag}$ then B does accept $Source_B$, so $Source_B \in Lang(B)$.

In either case, the input $Source_B$ is in one language and not the other.

□

In the following space of Languages:



We have found a language (L_{diag}) beyond the outermost circle.

We now prove that $L_{TM-accept}$ is not decidable (which shows that Turing-decidable $\neq$ Turing-recognisable).

We do this using proof by contradiction

Proof by Contradiction (assumed knowledge)

To prove a statement is true, we can try follow these steps:

1. Assume the statement is not true.
2. Show this assumption implies an untrue statement. We refer to this statement as a contradiction.
3. This then allows us to reject the assumption that the statement is not true.
4. We then conclude the opposite, i.e., that the statement is true.

So: we can try prove a language is undecidable by assuming the language is decidable and show that leads to a contradiction.

This is what we will do to show that $L_{TM-accept}$ is undecidable.

What is the untrue statement going to be? That L_{diag} is recognisable (which we know is untrue!)

Theorem

$L_{TM-accept} = \{Source_M; w \mid M \text{ is a TM that accepts } w\}$ *is undecidable.*

Proof: Assume $L_{TM-accept}$ is decided by a TM A.

There is a Python function `pyA(Source`$_M$`, w)` that returns `True` if and only if M accepts w and returns `False`, otherwise.

We will use A to build a TM B that recognises L_{diag} (which gives the contradiction).

B is simply a TM that on inputs $Source_M$ has the opposite halting behaviour to A running on $Source_M; Source_M$. I.e.,

```
def pyB(x):
    return not pyA(x, x)
```



```
def pyB(x):
    return not pyA(x, x)
```

For every TM M, the following are equivalent statements:

- B accepts $Source_M$
- A does not accept $Source_M; Source_M$
- M does not accept $Source_M$
- $Source_M \in L_{diag}$

This implies B recognises (in fact, decides) L_{diag}. This is a contradiction. Hence, $L_{TM-accept}$ is undecidable. □

The Halting Problem

Another important undecidable language is:

$$L_{TM-halt} = \{\text{Source}_M; w \mid M \text{ is a TM that halts on } w\}$$

Note: proving $L_{TM-halt}$ is undecidable is a question in this week's tutorial.

Exercise: show the undecidability of the Halting Problem implies the Undecidability of $L_{Python-Halt}$ (Hint: Use the Church Turing Thesis).

Summary so far

- L_{diag} is not recognisable.
- $L_{TM-accept}$ is recognisable but not decidable.
- $L_{TM-halt}$ is recognisable but not decidable.

We now develop some techniques that can show, in particular, that $L_{TM-not-accept}$ is not recognisable.

Closure Properties

Closure Properties are theorems about the effect of set-theoretic operations (intersections, unions, complements, etc.) have on languages in terms of recognisability and decidability.

Closure Properties are useful for proving recognisability / decidability by applying set-theoretic operations on languages whose recognisability / decidability is already known.

Theorem

The union of decidable languages is decidable.

Proof: Let L_1 and L_2 be languages that are decided by TMs M_1 and M_2, respectively. We find that the following Python program[2]:

```
def pyM1UnionM2(x):
    return pyM1(x) or pyM2(x)
```

Corresponds to a Turing Machine M that accepts an input w if and only if either M_1 or M_2 does. Hence, M recognises $L_1 \cup L_2$.

Moreover, M always halts because M_1 and M_2 always halt (because they're deciders). Thus, M decides $L_1 \cup L_2$. □

[2] If input string x is valid for M_1 but not M_2 we can easily set `pyM1(x) == False`.

Theorem

If a language is decidable then so is its complement.

Proof: If a language L is decidable then we find the complement of L (denoted L^c) is decided by the Turing Machine corresponding to the following Python program:

```
def pyNotD(x):
    return not pyD(x)
```

Where D is the decider for L.

The corresponding TM **NotD** recognises L^c because $w \in L^c$ if and only if D does not accept it.

Moreover, **NotD** decides L^c because `pyNotD` always returns given D is a decider. □

An Application of Closure Properties

Theorem

$L_{TM-not-accept} = \{Source_M; w \mid M$ is a TM that does not accept $w\}$ is undecidable.

Proof: We assume $L_{TM-not-accept}$ is decidable and will show this leads to a contradiction.

The set of all strings Σ^* is the union of $L_{TM-not-accept}$, $L_{TM-accept}$ and L_{invald}, where:

$$L_{invalid} = \{s \in \Sigma^* \mid s \neq \text{Source}_M; w \text{ for any TM } M \text{ and input } w\}$$

I.e.,

$$L_{TM-accept} = \Sigma^* \setminus (L_{invalid} \cup L_{TM-not-accept}) = (L_{invalid} \cup L_{TM-not-accept})^c$$

We have already proven $L_{invalid}$ is decidable, so if $L_{TM-not-accept}$ is decidable then the union $L_{invalid} \cup L_{TM-not-accept}$ is also decidable.

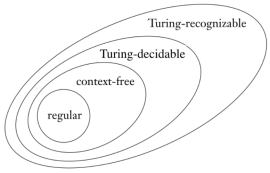
The equation $L_{TM-accept} = (L_{invalid} \cup L_{TM-not-accept})^c$ implies that $L_{TM-accept}$ is the complement of a decidable language.

However, we have already shown $L_{TM-accept}$ is undecidable, a contradiction. □

Note: In fact, $L_{TM-not-accept}$ is not recognisable either. We prove this in the reference slides using additional closure properties.

Summary

- Not Turing-recognisable: L_{diag} and $L_{TM-not-accept}$.
- Turing-recognisable but not Turing-decidable: $L_{TM-accept}$ and $L_{TM-halt}$.
- The undecidability of $L_{TM-not-accept}$ followed from useful closure properties with respect to decidability.



Additional Closure Properties

The following theorems are straightforward exercises.

Theorem

The union of recognisable languages is recognisable.

Theorem

The intersection of recognisable languages is recognisable.

Theorem

The intersection of decidable languages is decidable.

Important: the complement of a recognisable language is not necessarily recognisable!

Hint: $L_{TM-halt}$ is recognisable but what about its complement?

Theorem

A language is decidable exactly when it and its complement are recognisable.

Proof: ($\Longrightarrow$): A decidable language is recognisable by definition. We also proved the complement a decidable language is decidable, and so recognisable too.

($\Longleftarrow$): If M_1 recognises L and M_2 recognises L^c, the following program decides L.

- For any input w, run M_1 and M_2 in parallel on w.
- use one tape for each machine.
- simulate one step of M_1 on tape 1 and one step of M_2 on tape 2.
- If M_1 ever accepts, then accept, and if M_2 ever accepts then reject.

The point is that exactly one of M_1, M_2 must eventually accept w, and so the program decides L. □

Application:

Theorem

$L_{TM-not-accept}$ is not recognisable.

Proof: We first note that ${L_{TM-accept}}^c = L_{TM-not-accept} \cup L_{invalid}$.

Furthermore, $L_{invalid}$ is decidable and hence recognisable (see "Good to know slides").

So, if we assume that $L_{TM-not-accept}$ is recognisable, because the union of recognisable languages is recognisable, that means $L_{TM-accept}$ and its complement are both recognisable. However, that implies that $L_{TM-accept}$ is decidable, which we know is impossible.

□

$L_{invalid}$ is Decidable

$$L_{invalid} = \{s \in \Sigma^* \mid s \neq \text{Source}_M; w \text{ for any TM } M \text{ and input } w\}$$

is Turing-decidable.

Reason: By the CT-thesis we just need to show that there is an algorithm that decides $L_{invalid}$: it accepts input s precisely when s is not of the form $\text{Source}_M; w$.

A parser for a programming language checks for syntax errors. It is not hard to write a program P (that always halts) and that checks if a string s can be split into $u; w$ and that u is the Morphett code of a TM M, and that w only uses characters from M's input alphabet.

So, the algorithm for $L_{invalid}$ takes s as input, runs the parser P on s, and does the opposite (accepts if P rejects, and rejects if P accepts).

History

Alan Turing developed the theory of Turing Machines in 1936 after a question posed by M. H. A. Newman in a lecture:

> *[Is] there a "mechanical process" which could be applied to a mathematical statement, and which would come up with the answer as to whether it was provable? [2, p. 93]*

Turing work famously proves the answer to this question is "no" — This is closely related to the undecidability results of today.

Figure 1: Alan Turing in 1951 [1]

Diagonalisation

Many results about the limitations of Computer Science, Math and Logic involve constructions that look like:

$$L_{diag} = \{\text{Source}_M : M \text{ is a TM that does not accept Source}_M\}$$

This is typically called diagonalisation — why?

Let's look at proving the undecidability of $L_{TM-accept}$ as an example.

Diagonalisation is not assessable in COMP2022/2922.

Firstly, the set of all TMs can be enumerated: $M_1, M_2, \ldots$ and an empty grid can be drawn as follows:

	Source$_{M_0}$	Source$_{M_1}$	Source$_{M_2}$	$\cdots$	Source$_{M_k}$	$\cdots$
M_0				$\cdots$		$\cdots$
M_1				$\cdots$		$\cdots$
M_2				$\cdots$		$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
M_k				$\cdots$		$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

If $L_{TM-accept}$ were decidable, we could fill the entries in:

	Source$_{M_0}$	Source$_{M_1}$	Source$_{M_2}$	$\cdots$	Source$_{M_k}$	$\cdots$
M_0	P_{00}	P_{01}	P_{02}	$\cdots$	P_{0k}	$\cdots$
M_1	P_{10}	P_{11}	P_{12}	$\cdots$	P_{1k}	$\cdots$
M_2	P_{20}	P_{21}	P_{22}	$\cdots$	P_{2k}	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
M_k	P_{k0}	P_{k1}	P_{k2}	$\cdots$	P_{kk}	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Such that P_{ij} is the halting state: *accept* if M_i accepts Source$_{M_j}$ and *reject* otherwise.

The top-left to bottom-right diagonal entries are those of the form P_{jj}.

L_{diag} consists of every Source$_{M_i}$ where P_{ii} is reject.

If $L_{TM-accept}$ were decidable, we could list all the TMs: $M_1, M_2, \ldots$ and the grid:

	Source$_{M_0}$	Source$_{M_1}$	Source$_{M_2}$	$\cdots$	Source$_{M_k}$	$\cdots$
M_0	P_{00}	P_{01}	P_{02}	$\cdots$	P_{0k}	$\cdots$
M_1	P_{10}	P_{11}	P_{12}	$\cdots$	P_{1k}	$\cdots$
M_2	P_{20}	P_{21}	P_{22}	$\cdots$	P_{2k}	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
M_k	P_{k0}	P_{k1}	P_{k2}	$\cdots$	P_{kk}	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

With P_{ij} be the halting state: *accept* if M_i accepts Source$_{M_j}$ and *reject* otherwise.

There is a TM $Liar$ whose halting state on E_j is the opposite of P_{jj}.

$Liar = M_k$ for some k. But if you try to evaluate P_{kk} you will reach a contradiction!

In short, everything in the Main slides & Reference Slides, but not the "Good to Know" slides.

Assessable Proving a language is recognisable/not recognisable, decidable/undecidable by:

- Creating New TMs, e.g., "if L is decidable, we can build a Turing Machine that halts when ..."[3]
- Closure Properties e.g., "if L is decidable then so is its complement, which implies ..."
- Proof by Contradiction, e.g., "If L is decidable then ..., which implies $L_{TM-accept}$ is decidable".

You should know the statements of all the theorems proven this week and their implications, e.g., to prove new facts.

[3] Creating TMs in this way is often called a "reduction"