COMPUTATION && LOGIC

SCHOOL OF INFORMATICS
THE UNIVERSITY OF EDINBURGH

Authors:

Haoran Peng: hpeng2@ed.ac.uk

Professor Michael Fourman: michael.fourman@ed.ac.uk

Contents

Many people have, directly or indirectly, assisted me in writing this book. Erik Rydow provided valuable feedback on content delivery and reader experience. Dr Iain Murray suggested multiple potential improvements on the book structure. Professor Don Sannella introduced me to Haskell and functional thinking and provided extremely detailed feedback on the book. Professor Stuart Anderson had been confident in me and this book since the beginning. I especially would like to thank Professor Michael Fourman for putting his trust in my ability to carry out such an important task and for his continuous support throughout the writing of this book. Most importantly, thank you for reading!

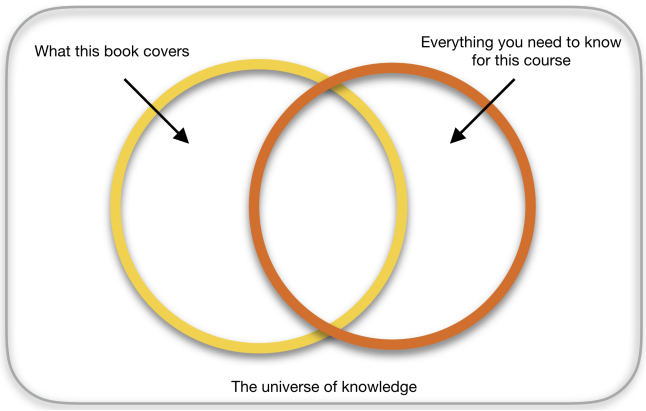
Preface

I took this course back when I was a fresher and as a student who had barely done any computing related stuff before, this introductory course was certainly not very "introductory". I still remember to this day the first few weeks when I was constantly confused and frustrated. But thank [insert whatever you believe here], it all came together at last. I sincerely hope that this book would save you some pain and help you build a solid foundation for your future years in the field of computer science.

This course is not easy and if anyone tells you that this course is easy, they are wrong. Though I can assure you that if you spend the time required, you will pass. There are also a lot of resources you can and should utilize, these include but not limited to:

- Prof Fourman and I, though we might be inaccessible at times
- Piazza Forum where you can ask questions anonymously, questions get answered quickly either by teaching staffs or fellow students
- Your tutors in tutorials, most of them took this course before and achieved excellent results, they should be able to answer most of your questions

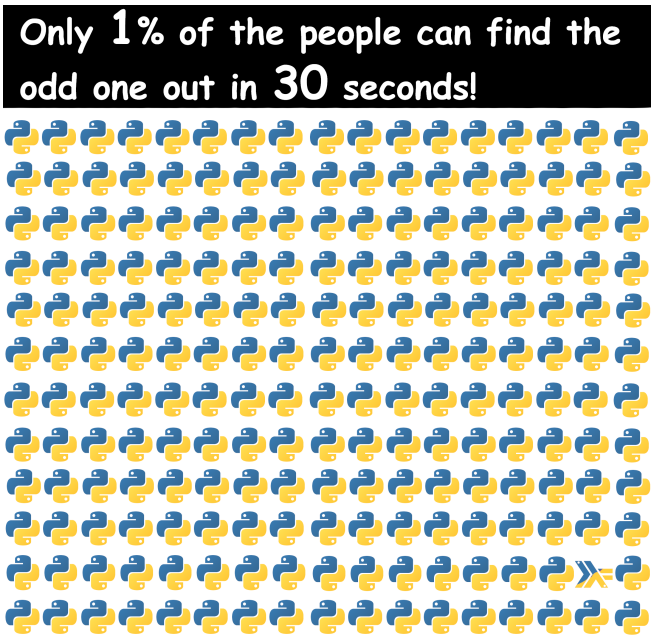
It is also important for you to realize that this book does not cover everything you need to know while also covering more than what you need to know. The Venn diagram below visualize this point:



Lastly, good luck and have fun!

Types

Odd One Out



You can skip this chapter if you get the joke about types in this 'quiz'.

I am sure you are the 1% and found the different logo very easily. Can you find the odd one out from the things below?

Hopefully, my face does not look too much like a fruit and you were able to tell that I am the odd one. Which is the odd one out below?

$$22 \qquad 3.14 \qquad \frac{3}{7} \qquad \text{True}$$

Easy to see that True is the odd one.

The point of these 'quizzes' is that humans are able to distinguish things. We know that apples, pears, and bananas are types of

fruits while my face isn't; 22, 3.14 and $\frac{3}{7}$ are types of numbers while True isn't. But computers only know two things, 0s and 1s. How do we tell them what things are?

Types in Haskell

We tell Haskell what things are by declaring their **types**. For example, if we want to tell Haskell `x` is an integer and it has a value of $25 - 3$, we write:

```
x :: Int
x = 25 - 3
```

In GHCi, you will need to enter:

```
GHCi> let x :: Int; x = 25 - 3
```

where the **type declaration** on the first line declares the type of the value on the second line.

Similarly, if we want `y` to be a floating-point number and take a value of 3.14, we write:

```
y :: Float
y = 3.14
```

Basic types in Haskell include but are not limited to:

- `Int`, integer with defined precision *i.e.* 32 or 64 bits depending on the computer.

  You can check it using GHCi:

  ```
  GHCi> maxBound :: Int
  ```

- `Integer`, on the other hand, has arbitrary precision.
- `Char`, characters like `'a'`, `'b'` etc.
- `[Char]` / `String`, a list of characters which is also called a string in Haskell. Thus the type declarations below are the same:

  ```
  s :: [Char]
  s :: String
  ```

  Use single quotes (`' '`) for `Char` and double quote (`" "`) for `String`

- `Bool`, if something is of type `Bool` then it can only take on boolean values `True` or `False`.

We will dive deeper into types in the future.

To check the type of things in GHCi, we use the `:t` command:

```
GHCi> :t "Is Haskell fun?"
"Is Haskell fun?" :: [Char]
GHCi> :t True
True :: Bool
```

Function Definitions

A **function** receives inputs and produces outputs. In mathematics, a function takes the form:

$$f(x) = x^2$$

where f is the name of the function, x is the input, and x^2 is the output.

x is just a variable, the output depends on the actual value you supply.

Similarly in Haskell:

```
f :: Int -> Int
f x = x ^ 2
```

Notice the type declaration says that the function `f` should be applied to an argument of type `Int` and it should return a result of type `Int`. Haskell will stop you from writing code that does not fit the type declaration. *e.g.*

```
GHCi> f True    -- This doesn't work: f can only be applied to an Int

<interactive>:32:3: error:
    • Couldn't match expected type ‘Int’ with actual type ‘Bool’
...
```

Suppose that we now want to square two numbers and add them together:

```
f' :: Int -> Int -> Int
f' x y = f x + f y
```

Notice that the type declaration now gives the type of each argument and the type of the result.

Type Checking

We have seen how typing can help us catch errors. Explicitly, Haskell's type checking does two things:

- It checks if a function's definition matches its type declaration. For example, the code below wouldn't work:

  ```
  GHCi> let f :: Int -> Int -> Bool; f a b = a + b

  <interactive>:18:38: error:
      • Couldn't match expected type ‘Bool’ with actual type ‘Int’
  ...
  ```

  Haskell throws a type error at us because the result's type is not `Bool`.

- It checks if a function is applied to arguments with correct types. If we have a valid function:

  ```
  GHCi> let f :: Int -> Int -> Bool; f a b = a == b
  ```

  but applied it to arguments of incorrect types:

  ```
  GHCi> f 3 True

  <interactive>:25:5: error:
      • Couldn't match expected type ‘Int’ with actual type ‘Bool’
  ...
  ```

  We get a type error as expected.

Set

Much of this chapter is borrowed from the first chapter of Mathematical Methods in Linguistics[1].

[1] Barbara Hall-Partee, Alice G. B. ter Meulen, and Robert Eugene Wall. *Mathematical methods in linguistics*. Kluwer Academic, 1990

What is a set?

A **set** is just a collection of things. To denote a set, we put its elements (the things) inside curly brackets. For a set of cities:

$$\text{Cities} = \{\text{Guangzhou}, \text{Stockholm}, \text{Edinburgh}, \text{Boston}\}$$

A set with only 1 element is called a **singleton** and a set with no element is called an empty/null set (denoted with a $\emptyset$). A set can also have infinitely many elements, *e.g.* the set of non-negative even integers {0, 2..}.

Set Membership

For a set A, an element a is either in the set (denoted $a \in A$), or not in the set (denoted $a \notin A$). For example:

$$\begin{aligned} \text{Edinburgh} &\in \text{Cities} \\ 3 &\in \{1, 2, 3\} \\ 4 &\notin \{1, 2, 3\} \end{aligned}$$

$\in$ and $\notin$ correspond to `elem` and `notElem` respectively:

```
GHCi> 3 `elem` [1,2,3]
True
GHCi> 4 `notElem` [1,2,3]
True
```

Three Ways to Specify Sets

1. **List** all of its elements directly as we have done above.

   ```
   data City = Guanghzou | Stockholm | Edinburgh | Boston

   cities :: [City]
   cities = [Guangzhou, Stockholm, Edinburgh, Boston]
   ```

   Don't worry about the first line just yet, it just lets Haskell know which cities we want to talk about.

2. **Select** from a set using a property. *e.g.* the set of positive integers with the property that each integer is divisible by 2 and 3:

$$\{x \mid x \text{ is divisible by } 2, x \text{ is divisible by } 3\}$$

The notation is called **set comprehension**.

The | reads 'such that', and the whole thing reads 'the set of x such that x is divisible by 2 and x is divisible by 3'.

```
div2and3 :: [Int]
div2and3 = [x | x <- [1..], x `mod` 2 == 0, x `mod` 3 == 0]
```

Putting backticks ` around a prefix function like `mod` allows us to use it like an infix function.
Infix operator, Jan 2008. URL https://wiki.haskell.org/Infix_operator

This is called **list comprehension** in Haskell. We are testing all positive integers (`x <- [1..]`), if an integer is divisible by 2 and 3 (`x `mod` 2 == 0, x `mod` 3 == 0`), then put it in the set. Observe that `div2and3` generates an infinite list, press Control-C to terminate when testing.

3. **Generate** the elements using rules. A simple example is to produce the set of non-negative even numbers:

Let E be the set of non-negative even numbers.
$0 \in E$ and if $x \in E$ then $x + 2 \in E$

We start with the singleton set with only 0 as its element: $E = \{0\}$ (first underlined rule). Then we can apply the second underlined rule repetitively. In the first repetition, we know that 0 is in E, so $0 + 2 = 2$ is also in E, so now $E = \{0, 2\}$. In the second repetition, we know that 0 and 2 are in E, so $0 + 2 = 2$ and $2 + 2 = 4$ are in E, so now $E = \{0, 2, 4\}$ etc. This can be done compactly in Haskell:

```
evens :: [Int]
evens = 0:[x + 2 | x <- evens]
```

You will be able to understand the code soon, just test it out in ghci for now:

```
GHCi> evens
[0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42...
```

Here is an example of specifying the set of all even numbers S in three ways:

1. $S = \{.. -4, -2, 0, 2, 4..\}$.
2. $S = \{x \mid x \text{ is divisible by } 2\}$ where $x \in \mathbb{Z}$
3. $0 \in S$ and if $x \in S$ then $x + 2 \in S$ and $x - 2 \in S$.

Although all three specifications describe the same set, one might find that some specifications are easier to implement in Haskell than others.

Infinite Lists and Haskell's Laziness

You have just seen that Haskell can deal with infinite lists such as the positive integers `[1..]`. This might be surprising - Haskell is ***lazy***. Haskell will do no computation until it is needed.[2] You will gradually understand laziness in subsequent chapters. To make things easier, you can use the function `take`, which takes the first elements from a list:

[2] Kind of like a university student who does not start an assignment until right before the deadline.

```
GHCi> take 10 evens
[0,2,4,6,8,10,12,14,16,18]
```

Sets and Lists

Two sets are equal if and only if they have precisely the same elements[3]. , thus the equality below is true:

$$\{1,1,1,2,3,3,4,5\} = \{3,2,2,2,1,5,4,5\}$$

[3] Robert Roth Stoll. *Sets, logic, and axiomatic theories*. Freeman and Company, 1974

While for a list in Haskell, this is not true:

```
GHCi> [1,1,1,2,3,3,4,5] == [3,2,2,2,1,5,4,5]
False
```

and we have to be very careful when we use lists to represent sets!

Lists **allow duplications** and are **ordered**. To test equality between two 'sets' that are represented using lists, we can first remove duplicated elements from the lists and then sort the lists.

First import the `Data.List module`:

```
-- nub and sort are in the Data.List module
GHCi> import Data.List
```

We use `nub` to remove duplications:

```
GHCi> nub [3,2,2,2,1,5,4,5]
[3,2,1,5,4]
```

and `sort` to sort:

```
GHCi> sort (nub [3,2,2,2,1,5,4,5])
[1,2,3,4,5]
```

Do the same for both lists and check equality:

```
GHCi> sort (nub [1,1,1,2,3,3,4,5]) == sort (nub [3,2,2,2,1,5,4,5])
True
```

Voilà!

You could use the Data.Set module which provides the function `fromList` that creates a set from a list.

```
GHCi> import Data.Set
GHCi> fromList [1,1,1,2,3,3,4,5] == fromList [3,2,2,2,1,5,4,5]
True
```

But we will continue to represent sets using lists in the following sections. **Do keep in mind the differences between sets and lists!**

Subsets and Power Sets

If every element in set A is in set B, then set A is a **subset** of set B[4] – denoted $A \subseteq B$. *e.g.*

$$\{a,b,c\} \subseteq \{s,b,a,e,g,i,c\} \qquad \{a,b,j\} \nsubseteq \{s,b,a,e,g,i,c\}$$
$$\{a,b,c\} \subset \{s,b,a,e,g,i,c\} \qquad \{s,b,a,e,g,i,c\} \not\subset \{s,b,a,e,g,i,c\}$$
$$\{a,\{a\}\} \subseteq \{a,b,\{a\}\} \qquad \emptyset \subset \{a\}$$
$$\{a\} \not\subset \{\{a\}\} \qquad \{a\} \in \{\{a\}\}$$

When using $\subset$ instead of $\subseteq$, we insist that the sets in comparison should not be equal. Also remember that the null set $\emptyset$ is a subset of every set.

Given a set A, the **power set** of A, denoted as $\wp(A)$, is a set of all[5] the subsets of A. *e.g.*

$$A = \{a,b,c\}$$
$$\wp(A) = \{\emptyset,\{a\},\{b\},\{c\},\{a,b\},\{a,c\},\{b,c\},\{a,b,c\}\}$$

Let us denote the **cardinality** (number of elements) of the set A as $|A|$. Then the number of elements in the power set of A is $|\wp(A)| = 2^{|A|}$. Please think about why this is true[6].

[4] In other words, B contains every element of A.

Important: Lists/Sets in Haskell are **homogeneous**, meaning that elements of a particular list must have the same type. So you cannot have sets such as $\{a,\{a\}\}$ in Haskell.

[5] Including the $\emptyset$ and A itself!

[6] Hint: For every subset of A, an element of A is either in it or not in it.

```
powerset :: [a] -> [[a]]
powerset [] = [[]]
powerset (x:xs) = [x:s | s <- powerset xs] ++ powerset xs
```

```
GHCi> powerset [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

Union, Intersection, Difference, Complement

Here are two sets of positive integers, one consists of integers divisible by 2 and the other by 3.

```
div2 :: [Int]
div2 = take 100 [x | x <- [1..], x `mod` 2 == 0]

div3 :: [Int]
div3 = take 100 [x | x <- [1..], x `mod` 3 == 0]
```

```
GHCi> div2
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42...
GHCi> div3
[3,6,9,12,15,18,21,24,27,30,33,36,39,42,45,48,51,54,57,60,63...
```

The **union** of two sets A and B is denoted as $A \cup B$ and defined as $\{x \mid x \in A \text{ or } x \in B\}$.

```
union :: [Int] -> [Int] -> [Int]
union setA setB = [x | x <- [1..], x `elem` setA || x `elem` setB]
```

```
GHCi> take 20 (union div2 div3)
[2,3,4,6,8,9,10,12,14,15,16,18,20,21,22,24,26,27,28,30]
```

The **intersection** of two sets A and B, $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$.

```
intersection :: [Int] -> [Int] -> [Int]
intersection setA setB = [x | x <- [1..], x `elem` setA && x `elem` setB]
```

```
GHCi> take 20 (intersection div2 div3)
[6,12,18,24,30,36,42,48,54,60,66,72,78,84,90,96,102,108,114,120]
```

Logical 'or' and 'and' operations in Haskell are performed using double vertical bars || and double ampersand &&, similar to many other programming languages.

The **difference** of two sets A and B, $A - B = \{x \mid x \in A \text{ and } x \notin B\}$.

```
difference :: [Int] -> [Int] -> [Int]
difference setA setB = [x | x <- [1..], x `elem` setA && x `notElem` setB]
```

```
GHCi> take 20 (difference div2 div3)
[2,4,8,10,14,16,20,22,26,28,32,34,38,40,44,46,50,52,56,58]
```

The **complement** of a set A, $A' = \{x \mid x \notin A\}$.[7]

[7] Note that we are talking about this with respect to a universe of positive integers $x \in \mathbb{Z}^+$. Otherwise complement doesn't make much sense.

```
complement :: [Int] -> [Int]
complement setA = [x | x <- [1..], x `notElem` setA]
```

```
GHCi> take 20 (complement div2)
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39]
```

As expected, the complement of positive even integers are positive odd integers.

Venn Diagram and Propositional Logic

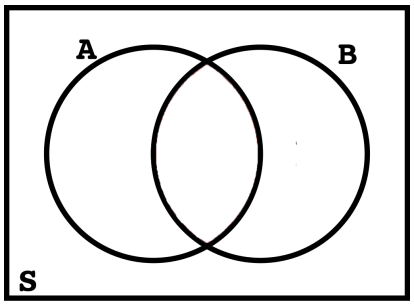
We just introduced sets and set operations and now we can use Venn diagrams to visualize them. We first need to specify a **universal set** S which contains all the numbers we are interested in. For now, I will choose a relatively small set of numbers:

$$S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$$

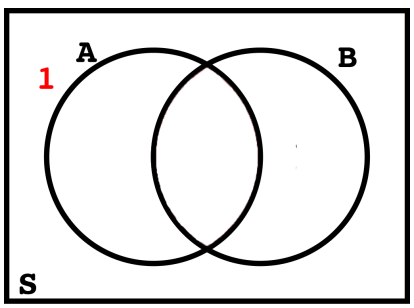
Let's visualize the relationship between set A and B where:

$$A = \text{Numbers in } S \text{ that are divisible by } 2$$
$$B = \text{Numbers in } S \text{ that are divisible by } 3$$

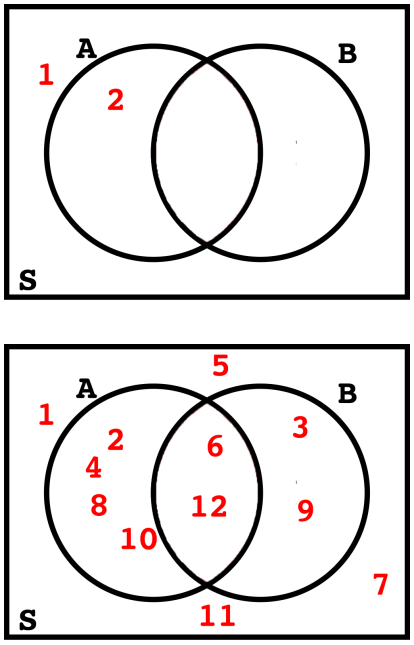
A Venn diagram template looks like so:



We can fill the diagram in by testing all the numbers in S against the properties of A and B *i.e.* divisible by 2 or divisible by 3. For example, 1 is not divisible by 2 and not divisible by 3, so it doesn't belong to set A or set B. Thus we put 1 outside the circles:



2 is divisible by 2 but not by 3, so it belongs to set A but not to set B. Thus we put 2 in the region of circle A where it doesn't overlap circle B:



...6 is divisible by both 2 and 3, thus we put 6 in the overlapping region of circle A and circle B...:

Finally we have the diagram filled. Notice that when I was describing the process, words like "**not**", "**or**", "**and**" were used frequently. Wouldn't it be nice if there is a language that makes this process less cluttered and more formal?

Yep, that language is called **propositional logic**. A proposition is a statement (declarative sentence) that is either true or false *e.g.* 4 is divisible by 2, 11 is divisible by 3 etc. A counter example would be "Is logic fun?" because it is a question and it doesn't make sense to say it is true or false[8].

To make more interesting propositions, we connect propositions with **logical connectives** like"**not**", "**or**", "**and**" *e.g.* 6 is divisible by 2 and 6 is divisible by 3. We will talk more about logical connectives in the next chapter.

[8] On the other hand, "Logic is fun" is a proposition.

Logical Connectives and Propositional Equivalences

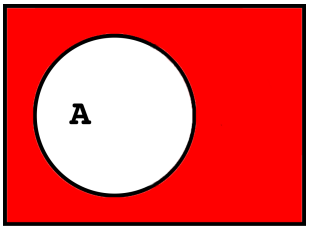
Let's have a look at some of the most common logical connectives presented in this course. For each logical connective, a corresponding **truth table** and Venn diagram will be shown. For now, we will use 1 and 0 to represent true and false respectively[9]. A and B are propositions that are either true or false.

[9] Other notations include $\top$ and $\bot$, T and F.

Common Connectives

$\neg$, not, negation

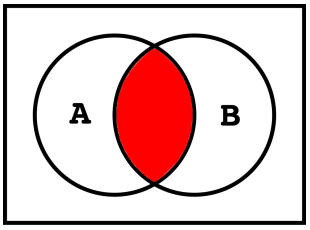
A	$\neg A$
0	1
1	0



If A is true, then "**not** A" is false and vice versa.

$\wedge$, and, conjunction

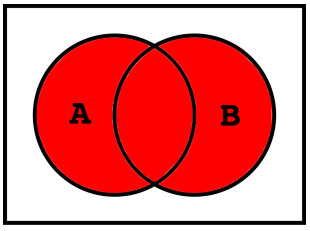
A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1



For "A **and** B" to be true, both A and B must be true.

$\vee$, or, disjunction

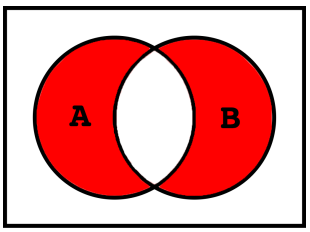
A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1



For "A **or** B" to be true, at least one of A and B must be true. Note that 'or' in English is almost always 'exclusive or' in logic. For example, if you ask me to bring you a beer or a cider at the bar, you'd be surprised if I bring you both.

$\oplus$, xor, exclusive or

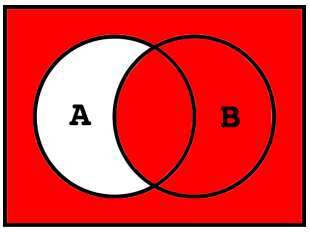
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0



For "A **xor** B" to be true, A and B must have opposite truth values.

$\rightarrow$, implies, implication

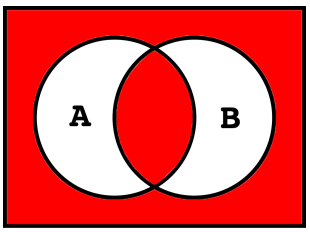
A	B	$A \rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1



Implication can be quite confusing at first. $A \rightarrow B$ is false **only** when A is true and B is false. So when A is false, $A \rightarrow B$ is always true.

$\leftrightarrow$, if and only if, iff, double implication

A	B	$A \leftrightarrow B$
0	0	1
0	1	0
1	0	0
1	1	1



Unlike xor, A and B must have the same truth value for "$A \leftrightarrow B$" to be true .

Order of Precedence

Suppose that you are asked to evaluate the logical expression below given that $A = B = C = 0$:

$$\neg A \leftrightarrow B \wedge \neg C$$

In what order should we do the evaluation? It depends on the precedence of the connectives. The connective (and the propositions it connects) with the highest precedence is evaluated first[10].

Similarly, in what order do we evaluate $1 + 2 \times 3 - 4$? We do multiplication first, then addition, then subtraction.

[10] Unless the expression is bracketed, in which case you would evaluate what's in the brackets first.

Connective	Precedence
$\neg$	1
$\wedge, \vee$	2
$\rightarrow, \leftrightarrow$	3

For $\neg A \leftrightarrow B \wedge \neg C$, according to the table, we evaluate $\neg$ first, then $\wedge$, then $\leftrightarrow$:

Observe that we have put $\wedge, \vee$ / $\rightarrow, \leftrightarrow$ on the same precedence because the precedence is not agreed upon. You will be presented with well-bracketed expressions when there are ambiguities.

$$
\begin{aligned}
&\neg A \leftrightarrow B \wedge \neg C \\
=&(\neg A) \leftrightarrow (B \wedge (\neg C)) \\
=&(\neg 0) \leftrightarrow (0 \wedge (\neg 0)) \\
=&1 \leftrightarrow (0 \wedge 1) \\
=&1 \leftrightarrow 0 \\
=&0
\end{aligned}
$$

When programming, it is recommended to check your language's documentation for operators' precedence. The (relative) precedence of common connectives in Haskell is:

Connective	Haskell	Precedence
$\neg$	`not`	1
$\leftrightarrow$	`==`	2
$\wedge$	`&&`	3
$\vee$	`||`	4

So we can put $\neg A \leftrightarrow B \wedge \neg C$ in Haskell as:

```
exp :: Bool -> Bool -> Bool -> Bool
exp a b c = not a == (b && not c)
```

And to evaluate its truth value given that $A = B = C = 0$:

```
GHCi> exp False False False
False
```

Propositional Equivalences

Some definitions you need to know:

- A proposition that is always true is a **tautology** *e.g.* $A \vee \neg A$.
- A proposition that is always false is a **contradiction** *e.g.* $A \wedge \neg A$.
- A proposition that is neither a tautology nor a contradiction is a **contingency** *e.g.* $A \vee B$.
- A proposition is **satisfiable** if it is either a tautology or a contingency.

Logical Equivalence

For two propositions A and B, they are logically equivalent if $A \leftrightarrow B$ is a tautology.

For example, $A \rightarrow B$ and $\neg A \vee B$ are logically equivalent and below is the truth table of $(A \rightarrow B) \leftrightarrow (\neg A \vee B)$:

A	B	$A \rightarrow B$	$\neg A \vee B$	$(A \rightarrow B) \leftrightarrow (\neg A \vee B)$
0	0	1	1	1
0	1	1	1	1
1	0	0	0	1
1	1	1	1	1

Boolean Algebra

In this section, you will learn how to manipulate logical expressions just like you learned how to manipulate algebraic expressions like $1 \div (2 + 3 \times 4)$. It does take some time to wrap your head around and the best way to master it is to do a bunch of exercises.

You need to import QuickCheck for this section in order to check the correctness of the properties:

```
GHCi> import Test.QuickCheck
```

Connetive Properties

Double Negation

$$\neg(\neg x) = x$$

This should come as no surprise, the negation of the negation of something is that something.

Associativity of $\wedge$ and $\vee$

Similar to addition where $1 + (2 + 3) = (1 + 2) + 3 = 1 + 2 + 3$

$$x \wedge (y \wedge z) = (x \wedge y) \wedge z$$

```
prop_associativity :: Bool -> Bool -> Bool -> Bool
prop_associativity x y z = (x && (y && z)) == ((x && y) && z)

GHCi> quickCheck prop_associativity
+++ OK, passed 100 tests.
```

$$x \vee (y \vee z) = (x \vee y) \vee z$$

```
prop_associativity2 :: Bool -> Bool -> Bool -> Bool
prop_associativity2 x y z = (x || (y || z)) == ((x || y) || z)

GHCi> quickCheck prop_associativity2
+++ OK, passed 100 tests.
```

Commutativity of $\wedge$ and $\vee$

Again, similar to $+$, $\times$

$$x \wedge y = y \wedge x$$

```
prop_commutativity :: Bool -> Bool -> Bool
prop_commutativity x y = (x && y) == (y && x)

GHCi> quickCheck prop_commutativity
+++ OK, passed 100 tests.
```


$$x \vee y = y \vee x$$

```
prop_commutativity2 :: Bool -> Bool -> Bool
prop_commutativity2 x y = (x || y) == (y || x)

GHCi> quickCheck prop_commutativity2
+++ OK, passed 100 tests.
```

Distributivity of $\wedge$ over $\vee$ and $\vee$ over $\wedge$

Similar to distributing $x(y+z)$ into $xy+xz$. Note that: you can distribute both "$\wedge$ over $\vee$" and "$\vee$ over $\wedge$" in boolean algebra.

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$

```
prop_distributivity :: Bool -> Bool -> Bool -> Bool
prop_distributivity x y z = (x && (y || z)) == ((x && y) || (x && z))

GHCi> quickCheck prop_distributivity
+++ OK, passed 100 tests.
```

$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$$

```
prop_distributivity2 :: Bool -> Bool -> Bool -> Bool
prop_distributivity2 x y z = (x || (y && z)) == ((x || y) && (x || z))

GHCi> quickCheck prop_distributivity2
+++ OK, passed 100 tests.
```

Useful identities for $\leftrightarrow$ and $\rightarrow$

$$x \leftrightarrow y = (x \rightarrow y) \wedge (y \rightarrow x)$$
$$x \rightarrow y = \neg x \vee y$$

Often in computer science, true is larger than false because $1 > 0$. This is also the case in Haskell:

```
GHCi> True > False
True
```

This essentially means that $x \leftrightarrow y$ can be written as `x == y` in Haskell and that $x \rightarrow y$ can be written as `x <= y`:

```
prop_iff :: Bool -> Bool -> Bool
prop_iff x y = (x == y) == (x <= y && x >= y)

GHCi> quickCheck prop_iff
+++ OK, passed 100 tests.
```

De Morgan's laws

$$\neg(x \wedge y) = \neg x \vee \neg y$$

You can draw truth tables and Venn diagrams to verify the correctness. I suggest to just remember them.

```
prop_demorgan :: Bool -> Bool -> Bool
prop_demorgan x y = (not (x && y)) == (not x || not y)

GHCi> quickCheck prop_demorgan
+++ OK, passed 100 tests.
```

$$\neg(x \vee y) = \neg x \wedge \neg y$$

```
prop_demorgan2 :: Bool -> Bool -> Bool
prop_demorgan2 x y = (not (x || y)) == (not x && not y)

GHCi> quickCheck prop_demorgan2
+++ OK, passed 100 tests.
```

Some extensions

Notice that the number of terms increases exponentially after distribution.

$$a \wedge (b_1 \vee b_2 \vee ... \vee b_n) = (a \wedge b_1) \vee (a \wedge b_2) \vee ... \vee (a \wedge b_n)$$
$$a \vee (b_1 \wedge b_2 \wedge ... \wedge b_n) = (a \vee b_1) \wedge (a \vee b_2) \wedge ... \wedge (a \vee b_n)$$
$$(a \wedge b) \vee (c \wedge d) = (a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d)$$
$$\neg(a_1 \wedge a_2 \wedge ... \wedge a_n) = \neg a_1 \vee \neg a_2 \vee ... \vee \neg a_n$$
$$\neg(a_1 \vee a_2 \vee ... \vee a_n) = \neg a_1 \wedge \neg a_2 \wedge ... \wedge \neg a_n$$

Check if some properties holds

The most systematic way is to draw a truth table[11], hereby I will demonstrate the associativity of $\oplus$ and the **non**-associativity of $\uparrow$[12].

[11] Not recommended when there are many variables, why?

[12] $\uparrow$ is called the Sheffer stroke or NAND. $x \uparrow y = \neg(x \wedge y)$.

x	y	z	$x \oplus y$	$y \oplus z$	$(x \oplus y) \oplus z$	$x \oplus (y \oplus z)$
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	1	1	1	1
0	1	1	1	0	0	0
1	0	0	1	0	1	1
1	0	1	1	1	0	0
1	1	0	0	1	0	0
1	1	1	0	0	1	1

The truth values for $(x \oplus y) \oplus z$ and $x \oplus (y \oplus z)$ are the same and therefore $\oplus$ is associative.

Now let's look at $\uparrow$:

x	y	z	$x \uparrow y$	$y \uparrow z$	$(x \uparrow y) \uparrow z$	$x \uparrow (y \uparrow z)$
0	0	0	1	1	1	1
0	0	1	1	1	0	1
0	1	0	1	1	.	.
0	1	1	1	0	.	.
1	0	0	1	1	.	.
1	0	1	1	1	.	.
1	1	0	0	1	.	.
1	1	1	0	0	.	.

The truth values already differ on the second row so $\uparrow$ is not associative. We can stop as soon as we find a counter-example.

Karnaugh Map Preview

A Karnaugh map is essentially a two dimensional truth table. We will demonstrate this by filling in some Karnaugh maps in this section.

Karnaugh Map Template

AB \ CD	00	01	11	10
00				
01				
11				
10				

To Fill a Karnaugh Map In

Given a logical expression, you can choose to first draw the truth table for that expression and then fill the Karnaugh map correspondingly or just fill out the Karnaugh map directly. Below is the truth table of $A \vee B \vee C$:

A	B	C	D	$A \vee B \vee C$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Every square of the Karnaugh map corresponds to a row in the truth table *e.g.* when $A = 1, B = 0, C = 1, D = 1$, and $A \vee B \vee C = 1$, the square on the row labelled 10 and column labelled 11 is filled with a 1. Fill in all the squares this way and we have:

AB \ CD	00	01	11	10
00	0	0	1	1
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

Karnaugh Map of $A \vee B \vee C$

Below is a second example in which we fill in a Karnaugh map for $A \vee C$:

A	B	C	D	$A \vee C$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

AB \ CD	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	1	1	1	1
10	1	1	1	1

Karnaugh Map of $A \vee C$

Below are some more filled Karnaugh maps:

AB \ CD	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	1	1	1	1

Karnaugh Map of A

AB \ CD	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

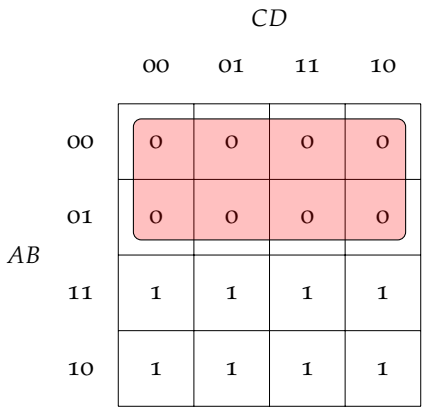
Karnaugh Map of $A \vee B \vee C \vee D$

Interesting Observations

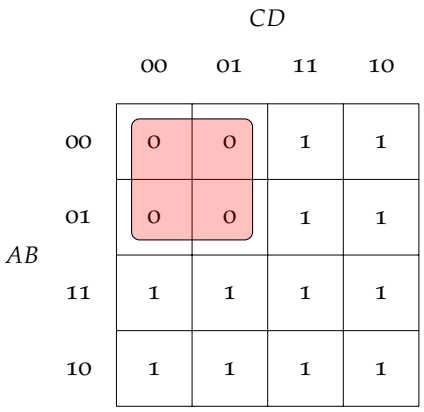
You might have noticed that all the examples Karnaugh maps are of disjunctions, *i.e.* A, $A \vee C$, $A \vee B \vee C$, $A \vee B \vee C \vee D$, this is intentional. Although we can construct Karnaugh maps for any propositions, we can observe some interesting things in Karnaugh maps of disjunctions.

The 0s form **blocks**.

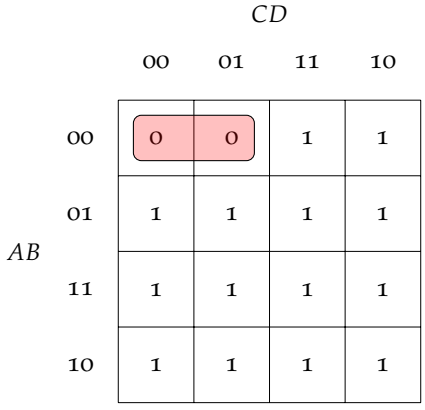
On the next page, I coloured these **blocks** so you can see them better.



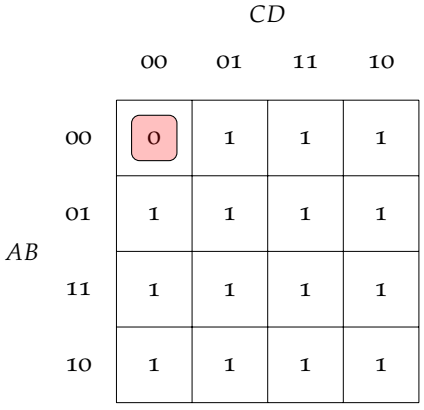
Karnaugh Map of A



Karnaugh Map of $A \vee C$



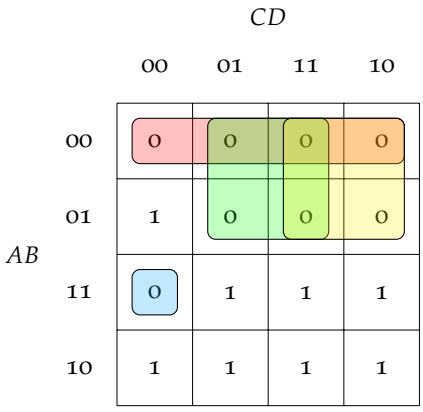
Karnaugh Map of $A \vee B \vee C$



Karnaugh Map of $A \vee B \vee C \vee D$

A block's size (the number of 0s it consists of) must be a power of 2 *i.e.* $\{1, 2, 4, 8, 16, ...\}$.

It turned out that **we can read off a disjunction from any block of 0s in a Karnaugh map**. How is this useful? Well, we will see why it is important in later chapters where we have Karnaugh maps with multiple blocks:



For now, you just need to have an idea about what Karnaugh maps are and how we can fill them in.

Entailment

Coming Soon

Conjunctive Normal Form

Definitions

Literal: a literal is a propositional variable or its negation, *e.g.* A, $\neg A$, B
Clause: a disjunction[13] of literals, *e.g.* $A \vee B \vee C$
Conjunctive normal form CNF: a conjunction of clauses, *e.g.* $(A \vee B) \wedge (\neg A \vee C \vee B) \wedge (B \vee \neg C)$
Clausal normal form: a CNF represented as a set of sets, *e.g.* $\{\{A, B\}, \{\neg A, C, B\}, \{B, \neg C\}\}$

[13] Sometimes conjunction, we will mainly be dealing with disjunctions.

Benefits of CNFs

Every logical expression can be converted to its CNF. Many conversion techniques will be covered in this chapter. It can be difficult for you to see the point of converting expressions to CNFs at this stage[14] but here are some remarks about its importance:

[14] I certainly didn't see the benefits of CNF when I learnt about it ¯_(ツ)_/¯

- Logical expressions can nest very deeply, it is easier both for humans and computers to understand CNFs, *e.g.*

$$\begin{aligned} & A \leftrightarrow (B \rightarrow (C \vee D)) \\ =& (A \vee B) \wedge (A \vee \neg C) \wedge (A \vee \neg D) \wedge (\neg A \vee \neg B \vee C \vee D) \end{aligned}$$

- To test whether an expression is satisfiable, we often convert it to its CNF and solve it using, for example, the DPLL algorithm.
- CNF can simplify circuit design. CNF only contains $\wedge$, $\vee$, $\neg$ and the corresponding circuit will only need these 3 types of gate to implement.

Throughout this chapter, we will convert two logical expressions to their corresponding CNFs using different approaches. The two expressions are:

$$\begin{aligned} & A \leftrightarrow (B \rightarrow (C \vee D)) \\ & ((A \rightarrow \neg C) \rightarrow \neg B) \rightarrow D \end{aligned}$$

Now let's start converting!

CNF with Boolean Algebra

We can use laws of Boolean algebra from last chapter to convert an expression into its CNF. You should get good at doing this method very quickly by doing some exercises.

$$
\begin{aligned}
& A \leftrightarrow (B \rightarrow (C \vee D)) \\
=& (A \rightarrow (B \rightarrow (C \vee D))) \wedge ((B \rightarrow (C \vee D)) \rightarrow A) && \leftrightarrow \text{ rule} \\
=& (A \rightarrow (\neg B \vee (C \vee D))) \wedge ((\neg B \vee (C \vee D)) \rightarrow A) && \rightarrow \text{ rule} \\
=& (A \rightarrow (\neg B \vee C \vee D)) \wedge ((\neg B \vee C \vee D) \rightarrow A) && \vee \text{ associativity} \\
=& (\neg A \vee (\neg B \vee C \vee D) \wedge (\neg(\neg B \vee C \vee D) \vee A) && \rightarrow \text{ rule} \\
=& (\neg A \vee \neg B \vee C \vee D) \wedge (\neg(\neg B \vee C \vee D) \vee A) && \vee \text{ associativity} \\
=& (\neg A \vee \neg B \vee C \vee D) \wedge ((\neg\neg B \wedge \neg C \wedge \neg D) \vee A) && \text{De Morgan's law} \\
=& (\neg A \vee \neg B \vee C \vee D) \wedge ((B \wedge \neg C \wedge \neg D) \vee A) && \text{Double Negation} \\
=& (\neg A \vee \neg B \vee C \vee D) \wedge (A \vee (B \wedge \neg C \wedge \neg D)) && \vee \text{ commutativity} \\
=& (\neg A \vee \neg B \vee C \vee D) \wedge ((A \vee B) \wedge (A \vee \neg C) \wedge (A \vee \neg D)) && \vee \text{ over } \wedge \text{ distributivity} \\
=& (\neg A \vee \neg B \vee C \vee D) \wedge (A \vee B) \wedge (A \vee \neg C) \wedge (A \vee \neg D) && \wedge \text{ associativity}
\end{aligned}
$$

$$
\begin{aligned}
& ((A \rightarrow \neg C) \rightarrow \neg B) \rightarrow D \\
=& ((\neg A \vee \neg C) \rightarrow \neg B) \rightarrow D && \rightarrow \text{ rule} \\
=& (\neg(\neg A \vee \neg C) \vee \neg B) \rightarrow D && \rightarrow \text{ rule} \\
=& \neg(\neg(\neg A \vee \neg C) \vee \neg B) \vee D && \rightarrow \text{ rule} \\
=& (\neg\neg(\neg A \vee \neg C) \wedge \neg B) \vee D && \text{De Morgan's law} \\
=& ((\neg A \vee \neg C) \wedge B) \vee D && \text{Double Negation} \\
=& D \vee ((\neg A \vee \neg C) \wedge B) && \vee \text{ commutativity} \\
=& (D \vee (\neg A \vee \neg C)) \wedge (D \vee B) && \vee \text{ over } \wedge \text{ distributivity} \\
=& (\neg A \vee \neg C \vee D) \wedge (D \vee B) && \vee \text{ associativity}
\end{aligned}
$$

Generally speaking, you should:

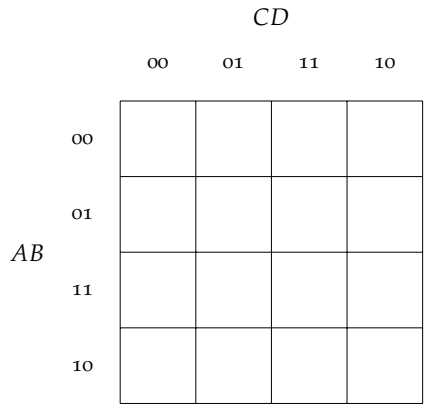
1. Remove implications.
2. Push negations in using De Morgan's law and remove double negations *i.e.* there should only be negations of atomic propositions after this step[15].
3. Distribute $\vee$ over $\wedge$.

These steps can, though, be performed in different orders depending on your preference. Furthermore, many steps can be omitted once you get familiar with Boolean algebra. For simple logical expressions, this systematic algebraic approach is fine. For more complicated expressions however, doing all the algebra is not only tedious but also often results in exponential blow-up of terms. You will learn other techniques to circumvent this later in this section.

[15] The syntactic form of such an expression is called negated normal form or NNF for short.

CNF with Karnaugh Map

A Karnaugh map (shown in the margin), is really just a two dimensional truth table.



Karnaugh Map Template with 4 Variables

To Fill a Karnaugh Map In - Revisited

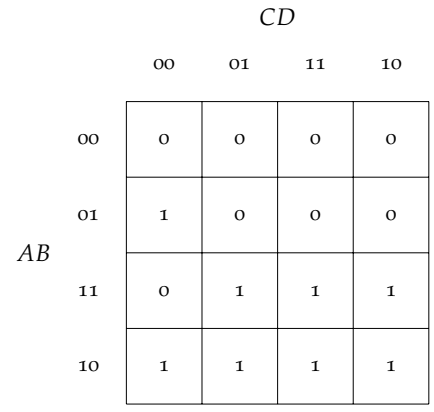
Given a logical expression, you can choose to first draw the truth table for that expression and then fill the Karnaugh map correspondingly or just fill out the Karnaugh map directly. Below is the truth table of $A \leftrightarrow (B \rightarrow (C \vee D))$:

A	B	C	D	$C \vee D$	$B \rightarrow (C \vee D)$	$A \leftrightarrow (B \rightarrow (C \vee D))$
0	0	0	0	0	1	0
0	0	0	1	1	1	0
0	0	1	0	1	1	0
0	0	1	1	1	1	0
0	1	0	0	0	0	1
0	1	0	1	1	1	0
0	1	1	0	1	1	0
0	1	1	1	1	1	0
1	0	0	0	0	1	1
1	0	0	1	1	1	1
1	0	1	0	1	1	1
1	0	1	1	1	1	1
1	1	0	0	0	0	0
1	1	0	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	1	1	1

Every square of the Karnaugh map corresponds to a row in the truth table *e.g.* when $A = 1, B = 0, C = 1, D = 1$, and $A \leftrightarrow (B \rightarrow (C \vee D)) = 1$, the square on the row labelled 10 and column labelled 11 is filled with a 1.

Before we fill in the Karnaugh map for another expression, we shall have a look at some important properties of Karnaugh maps:

- The truth values, *i.e.* the labels, changes from 00 to 01 to 11 to 10. It does **NOT** go from 00 to 01 to 10 to 11 as many might think. This is done to ensure that only one digit changes each time.
- The encoding (we used AB for rows and CD for columns) can be different (*e.g.* CB for rows and AD for columns).
- We visualize a Karnaugh map as a surface in three dimensional space. I will elaborate more about this.



Filled-in Karnaugh Map of $A \leftrightarrow (B \rightarrow (C \vee D))$

Below is the second example $(((A \to \neg C) \to \neg B) \to D)$:

A	B	C	D	$A \to \neg C$	$(A \to \neg C) \to \neg B$	$((A \to \neg C) \to \neg B) \to D$
0	0	0	0	1	1	0
0	0	0	1	1	1	1
0	0	1	0	1	1	0
0	0	1	1	1	1	1
0	1	0	0	1	0	1
0	1	0	1	1	0	1
0	1	1	0	1	0	1
0	1	1	1	1	0	1
1	0	0	0	1	1	0
1	0	0	1	1	1	1
1	0	1	0	0	1	0
1	0	1	1	0	1	1
1	1	0	0	1	0	1
1	1	0	1	1	0	1
1	1	1	0	0	1	0
1	1	1	1	0	1	1

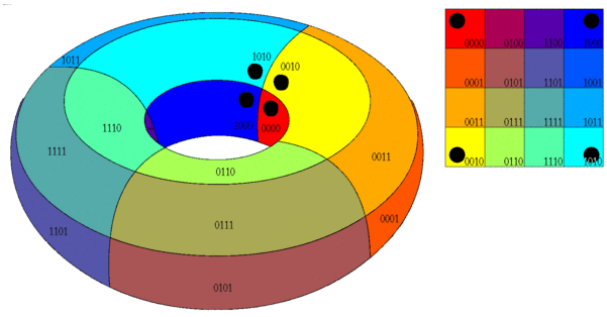
AB \ CD	00	01	11	10
00	0	1	1	0
01	1	1	1	1
11	1	1	1	0
10	0	1	1	0

Filled-in Karnaugh Map of $((A \to \neg C) \to \neg B) \to D$

To Read a Karnaugh Map Off

Figure below [16] is commonly used to visualize a Karnaugh map in three dimensional space. Keep this donut in mind when you are reading off a Karnaugh map.

[16] Karnaugh map, May 2019c. URL https://en.wikipedia.org/wiki/Karnaugh_map



The procedure of reading off CNFs from Karnaugh maps is not that trivial and there are some small caveats, so take your time when reading the next few pages.

The procedure is as follows:

1. Group all 0s into rectangular blocks. The number of 0 in each block must be a power of 2, that is 1, 2, 4, 8, or 16 in a 4-by-4 Karnaugh map. Blocks of 0 can overlap and you should always try to make the blocks as large as possible to achieve a minimal CNF *i.e.* a canonical CNF. **Although it is equally valid to have non-optimal blocks.** I will show you the difference soon.

2. For each block, look at its rows' and columns' truth values, *i.e.* its labels, and fill them in a table:

AB \ CD	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	0	1	1	1
10	1	1	1	1

Optimal K-Map Grouping of $A \leftrightarrow (B \to (C \vee D))$

	AB		CD	
Red Block	00		00,01,11,10	
Green Block	00,01		01,11	
Yellow Block	00,01		11,10	
Blue Block	11		00	

3. Identify the literals whose truth value do not change. For the red block, A and B can only be 0, write down $\neg A$ and $\neg B$; C and D can be both 0 or 1, write down nothing. For the green block, A is always 0 while B can be 0 or 1, write down only $\neg A$; C can be 0 or 1 while D can only be 1, write down only D...

	AB		CD	
Red	00	$\neg A, \neg B$	00,01,11,10	
Green	00,01	$\neg A$	01,11	D
Yellow	00,01	$\neg A$	11,10	C
Blue	11	A, B	00	$\neg C, \neg D$

4. Negate ($\neg$) all the literals.

	AB		CD	
Red	00	A, B	00,01,11,10	
Green	00,01	A	01,11	$\neg D$
Yellow	00,01	A	11,10	$\neg C$
Blue	11	$\neg A, \neg B$	00	C, D

5. Disjoin ($\vee$) the literals in each row then conjoin ($\wedge$) the rows for the CNF.

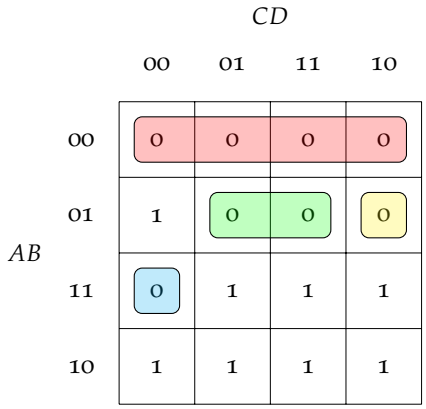
Red	$A \vee B$
Green	$A \vee \neg D$
Yellow	$A \vee \neg C$
Blue	$\neg A \vee \neg B \vee C \vee D$

$$
\begin{aligned}
&((A \to \neg C) \to \neg B) \to D \\
=&(A \vee B) \wedge (A \vee \neg D) \wedge (A \vee \neg C) \wedge (\neg A \vee \neg B \vee C \vee D)
\end{aligned}
$$

Once you get familiar with reading off a Karnaugh map, it is entirely **unnecessary** for you to make the tables. They are only here to help with your initial understanding. I'm sure you will get very good at this procedure very soon.

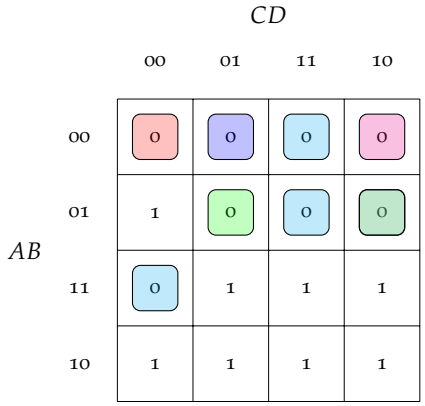
What happens when your blocks are not optimal?

Take the same example, you can choose to group the 0s like this:



Non-optimal K-Map Grouping of $A \leftrightarrow (B \rightarrow (C \vee D))$

Or even like this:



Pessimal K-Map Grouping of $A \leftrightarrow (B \rightarrow (C \vee D))$

For the Karnaugh map above, you will get this CNF following the steps described:

$$(A \vee B) \wedge (A \vee \neg B \vee \neg D) \wedge (A \vee \neg B \vee \neg C \vee D) \wedge (\neg A \vee \neg B \vee C \vee D)$$

For the one in the margin:

$$(A \vee B \vee C \vee D) \wedge (A \vee B \vee C \vee \neg D) \wedge (A \vee B \vee \neg C \vee \neg D) \wedge (A \vee B \vee \neg C \vee D) \wedge$$
$$(A \vee \neg B \vee C \vee \neg D) \wedge (A \vee \neg B \vee \neg C \vee \neg D) \wedge (A \vee \neg B \vee \neg C \vee D) \wedge (\neg A \vee \neg B \vee C \vee D)$$

Compare these CNFs with the one on the previous page and you should see that these are much longer.[17] Hopefully, it is now clear why you should always try to make the blocks as large as possible[18].

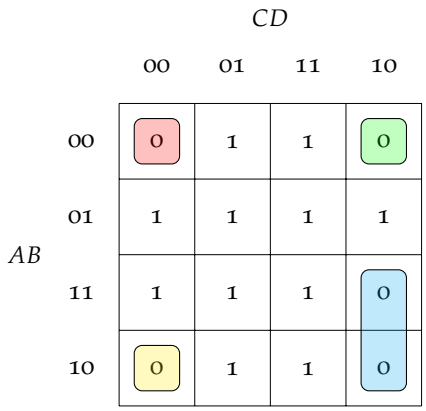
This behaviour also tells you an important fact about logical expressions and their CNFs - **a logical expression can have multiple equivalent CNFs.** It is not enough to just look at two 'different' CNFs and decide they are logically inequivalent.

[17] Although they are all equivalent.

[18] Nobody likes unnecessary work aye?

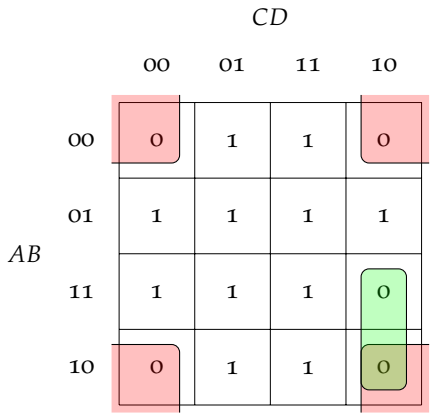
Two Dimensional Surface in Three Dimensional Space

I have emphasized a lot that Karnaugh maps are “surfaces of donuts". Below is an example to showcase this property. Consider the blocks of 0s for the following Karnaugh map:



Non-optimal K-Map Grouping of $((A \to \neg C) \to \neg B) \to D$

Remember that you can wrap a Karnaugh map around and connecting the four corners. A better grouping is thus:



Optimal K-Map Grouping of $((A \to \neg C) \to \neg B) \to D$

The CNF for $((A \to \neg C) \to \neg B) \to D$ is then $(B \vee D) \wedge (\neg A \vee \neg C \vee D)$.

Intuition on Correctness

First consider what happens if we group every single 0 into a block.

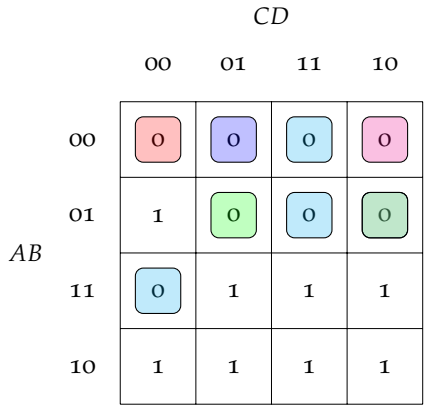


Figure 1: Pessimal K-Map Grouping of $A \leftrightarrow (B \rightarrow (C \vee D))$

For the first red block, we negate the literals and disjoin them ($\vee$):

$$(A \vee B \vee C \vee D) \wedge ...$$

... means there is something there but we don't care about it at the moment.

If we plug $A = 0, B = 0, C = 0, D = 0$ into it, we will get:

$$(0 \vee 0 \vee 0 \vee 0) \wedge ... = 0$$

which is correct because $0 \vee 0 \vee 0 \vee 0 = 0$ and $0 \wedge$ anything $= 0$. At this point if we plug in $A = 0, B = 0, C = 0, D = 1$, we will get a result of 1 which is incorrect. So we will need to conjoin $(A \vee B \vee C \vee \neg D)$ to eliminate such behaviour:

$$(A \vee B \vee C \vee D) \wedge (A \vee B \vee C \vee \neg D) \wedge ()$$

$$\begin{aligned} &(0 \vee 0 \vee 0 \vee 1) \wedge (0 \vee 0 \vee 0 \vee \neg 1) \\ =&1 \wedge 0 = 0 \end{aligned}$$

Do this for all the blocks, we will then 'exclude' all the blocks of 0s. The reason why we can make the blocks larger is because certain literals' values do not matter once we know some conditions. E.g. for the first row of the Karnaugh map above, if we know $A = 0, B = 0$, then the final truth value will be 0 no matter what CD are.

The blocks can overlap because it does not matter how many times some 0s are excluded as long as they are excluded at least once.

Tseytin Transformation

A paper (rather hard to read) written by Tseytin that should be of interest to you: https://link.springer.com/content/pdf/10.1007/3-540-11157-3_37.pdf

Remember that for a more complex logical expression, CNF conversion with boolean algebra usually results in an exponential blow-up in the number of clauses. Tseytin transformation is a technique to avoid that problem and keep the complexity down. Here is a quick demonstration of the procedure using the expression:

$$A \leftrightarrow (B \rightarrow (C \vee D))$$

Going by the order of operation, we first look at the inner most brackets $(C \vee D)$. We introduce a new variable x_1 then claim that $(C \vee D)$ and x_1 are equivalent:

$$x_1 \leftrightarrow (C \vee D) \quad (1)$$

Now substitute x_1 back into the original expression:

$$A \leftrightarrow (B \rightarrow x_1)$$

Repeat the same thing with $(B \rightarrow x_1)$ by introducing a new variable x_2:

$$x_2 \leftrightarrow (B \rightarrow x_1) \quad (2)$$

Substitute back again:

$$A \leftrightarrow x_2$$

Same thing again for $A \leftrightarrow x_2$:

$$x_3 \leftrightarrow (A \leftrightarrow x_2) \quad (3)$$

Substitute back:

$$x_3 \quad (4)$$

The core procedures of Tseytin transformation is effectively over by now. We now just need to convert each of (1)(2)(3)(4) into CNF (using either Boolean algebra or Karnaugh maps) then conjoin the CNFs together - $(1) \wedge (2) \wedge (3) \wedge (4)$.

Finishing off the transformation:

$x_1 \leftrightarrow (C \vee D) \quad (1) = (x_1 \vee \neg C) \wedge (x_1 \vee \neg D) \wedge (\neg x_1 \vee C \vee D)$

$x_2 \leftrightarrow (B \rightarrow x_1) \quad (2) = (x_2 \vee B) \wedge (x_2 \vee \neg x_1) \wedge (\neg x_2 \vee \neg B \vee x_1)$

$x_3 \leftrightarrow (A \leftrightarrow x_2) \quad (3) = (x_3 \vee \neg A \vee \neg x_2) \wedge (x_3 \vee A \vee x_2) \wedge (\neg x_3 \vee \neg A \vee x_2) \wedge (\neg x_3 \vee A \vee \neg x_2)$

$x_3 \quad (4)$

Conjoin all CNFs, the final **equisatisfiable**[19] CNF is:

$$(x_1 \vee \neg C) \wedge (x_1 \vee \neg D) \wedge (\neg x_1 \vee C \vee D) \wedge (x_2 \vee B) \wedge (x_2 \vee \neg x_1)$$
$$\wedge (\neg x_2 \vee \neg B \vee x_1) \wedge (x_3 \vee \neg A \vee \neg x_2) \wedge (x_3 \vee A \vee x_2)$$
$$\wedge (\neg x_3 \vee \neg A \vee x_2) \wedge (\neg x_3 \vee A \vee \neg x_2) \wedge x_3$$

[19] This expression is satisfiable if, and only if, the original expression is satisfiable. It is **not** equivalent to the original expression because it contains additional variables.

This CNF is only satisfiable when x_3 is true. Thus we can reduce the CNF to:

$$(x_1 \vee \neg C) \wedge (x_1 \vee \neg D) \wedge (\neg x_1 \vee C \vee D) \wedge (x_2 \vee B) \wedge (x_2 \vee \neg x_1)$$
$$\wedge(\neg x_2 \vee \neg B \vee x_1)$$
$$\wedge(\neg A \vee x_2) \wedge (A \vee \neg x_2)$$

Syntax of Propositional Logic in Haskell

So far, we have used the functions &&, ||, `not`, <=, and == to 'do logic'. It looks like we can do everything we might want to do using these until we start to manipulate the syntax of propositions[20].

In this chapter, we are first going to let Haskell know the syntactic structure of propositional logic. Then we will let it do Boolean algebra. Our final goal is for Haskell to be able to convert logical expressions into CNFs.

[20] Such as turning an arbitrary expression into its CNF using Boolean algebra.

Define Well-formed Expressions in Haskell

We will start by making a new data type called **Exp** (expression) using the **data** keyword:

```
data Exp = ...  -- Hey Haskell, I want you to know about logical expressions, nicknamed Exp...
```

A well-formed logical expression is an expression that conform to some grammar[21]. For example, $A \leftrightarrow (B \rightarrow (C \vee D))$ is a well-formed expression and conversely, $AA \rightarrow B \wedge C \vee$ isn't. We can tell if an (simple) expression is well-formed easily but in order for a computer to do that, we need to tell it explicitly what is allowed. The three most basic well-formed logical expressions are:

[21] Rules regarding how things can be structured, think about English grammar.

1. Atomic propositional **Var**iables like p, q, r
2. **T** – true
3. **F** – false

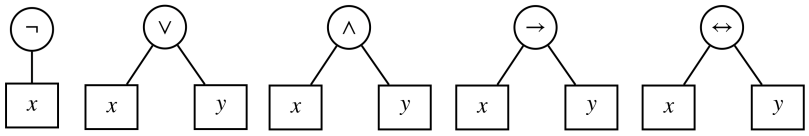
We can let Haskell know by enumerating them in Exp:

```
data Exp = Var String
         | T
         | F
         | ...
-- Exp can be a variable (which is constructed using a String)
-- Exp can be T
-- Exp can be F
-- Exp can be ...
```

Now we can introduce the logical connectives. We know they operate on logical expressions and except for $\neg$, all connectives we have studied are binary - meaning that they join two propositions together (in Haskell's term, they represent operators that take 2 arguments):

4. If x is an expression, then so is $\neg x$
5. If x and y are expressions, then so is $x \vee y$
6. If x and y are expressions, then so is $x \wedge y$
7. If x and y are expressions, then so is $x \rightarrow y$
8. If x and y are expressions, then so is $x \leftrightarrow y$

You can add more connectives if you like, this is just a standard set of connectives we use in this course.



Continuing from last code snippet[22]:

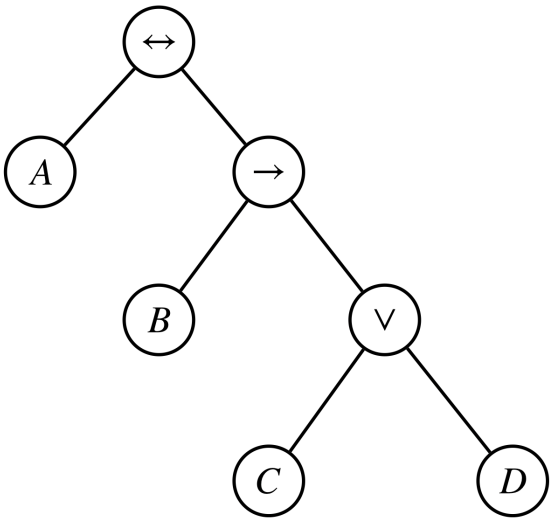
[22] Each case involves a **constructor**, which is either capitalised or an infix symbol. Infix constructors must start with a colon. We use colons on both side just for visual purposes.

```
data Exp = Var String
         | T
         | F
         | Not Exp      -- The negation of an expression is a new expression
         | Exp :|: Exp  -- The disjunction of two expressions is a new expression
         | Exp :&: Exp  -- etc...
         | Exp :->: Exp
         | Exp :<->: Exp
         deriving (Eq, Show, Ord)
```

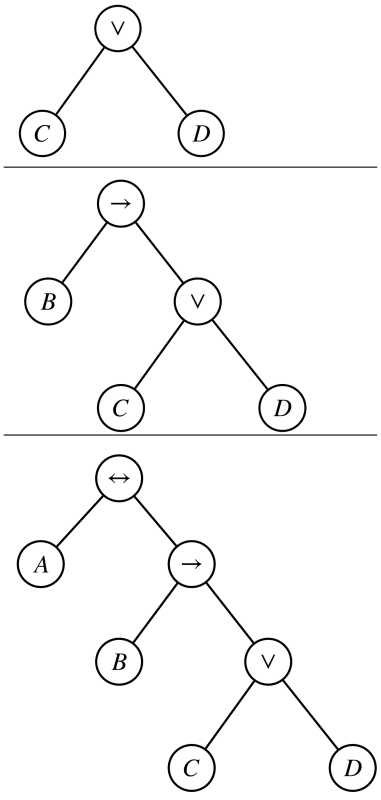
Now we have taught Haskell about well-formed logical expressions. We can write $A \leftrightarrow (B \rightarrow (C \vee D))$ as:

```
exp1 :: Exp
exp1 = Var "A" :<->: (Var "B" :->: (Var "C" :|: Var "D"))
```

We can visualize the `exp1` with a binary tree in which every subtree is an `Exp`:



Roughly how `exp1` is built from the bottom-up:



Show Your Expressions

If you want to see what `exp1` is in GHCi, you can do:

```
GHCi> exp1
Var "A" :<->: (Var "B" :->: (Var "C" :|: Var "D"))
```

It looks exactly like what we have typed in before, which is fine. Though we can do better by defining a function to pretty print it. There are a few objectives we want to achieve with the function:

1. If we see a variable(`Var String`), we want to show its content *i.e.* the `String`.
2. If we see either `T` or `F`, just show it.
3. If we see an expression of the form `Not Exp`, we want to replace the `Not` with '~'.
4. if we see an expression of the form `Exp :Connective: Exp`, we want to strip the colons around the connective.

The type declaration is:

```
showExp :: Exp -> String
```

The function definition is simply a matter of pattern matching, using ++ to join strings:

```
showExp (Var x)     = x       -- Objective 1
showExp (T)         = "T"     -- Objective 2
showExp (F)         = "F"     -- Objective 2
showExp (Not p)     = "(~" ++ showExp p ++ ")"    -- Objective 3
showExp (p :|: q)   = "(" ++ showExp p ++ " | "   ++ showExp q ++ ")"    -- Objective 4
showExp (p :&: q)   = "(" ++ showExp p ++ " & "   ++ showExp q ++ ")"    -- Objective 4
showExp (p :->: q)  = "(" ++ showExp p ++ " -> "  ++ showExp q ++ ")"    -- Objective 4
showExp (p :<->: q) = "(" ++ showExp p ++ " <-> " ++ showExp q ++ ")"    -- Objective 4
```

Test our new function in GHCi:

```
GHCi> showExp exp1
"(A <-> (B -> (C | D)))"
```

Evaluate Your Expressions

In order to evaluate an expression, we first need to prove a valuation which associates a boolean value with each variable in the expression. A valuation can look something like this in Haskell:

```
val :: [(String, Bool)]
val = [("A", True), ("B", False), ("C", True), ("D", False)]
```

We can make our code clearer by introducing a type synonym:

```
type Valuation = [(String, Bool)]
```

So we can simply write:


```
val1 :: Valuation
val1 = [("A", True), ("B", False), ("C", True), ("D", False)]
```

Our evaluation function should have the type declaration:

```
eval :: Valuation -> Exp -> Bool
```

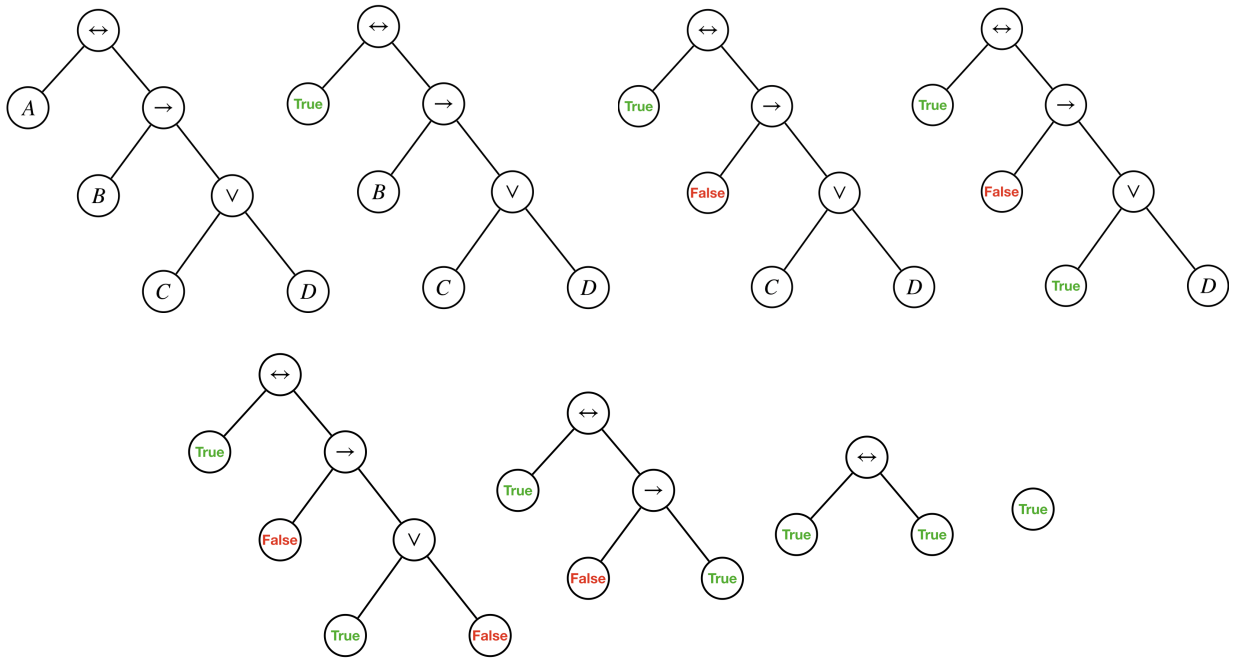
where it takes a valuation and an expression as inputs and outputs a boolean. Here is the definition of `eval`:

```
eval v (Var x)     = lookUp x v
eval v (T)         = True
eval v (F)         = False
eval v (Not p)     = not (eval v p)
eval v (p :|: q)   = eval v p || eval v q
eval v (p :&: q)   = eval v p && eval v q
eval v (p :->: q)  = not (eval v p) || eval v q
eval v (p :<->: q) = eval v p == eval v q
```

You might have noticed that the code of `showExp` and `eval` are similar and this is no coincidence because the order in which we traverse the expression tree is the same.

```
lookUp :: String -> Valuation -> Bool
lookUp var v = the [ b | (x, b) <- v, x == var ]
    where the [x] = x
          the _   = error "lookUp: variable missing or not unique"
```

It is probably best to demonstrate what `eval` does by showing a concrete example step-by-step. We will evaluate the expression `exp1` against the valuation `val1`. Below is (roughly) a series of steps that are taken (from left to right, top to bottom):



Obviously you will just get the final result if you just enter the command in Haskell:

```
GHCi> eval val1 exp1
True
```

Expression Manipulation

Now that Haskell knows what logical expressions are, we can teach it how to manipulate them using Boolean algebra. Remember the three steps for turning an expression into its CNF are:

1. Remove implications.
2. Push negations in and remove double negations *i.e.* there should only be negations of atomic propositions after this step.
3. Distribute $\vee$ over $\wedge$.

Let's write a function for each step.

Step 1: Remove implications.

```
removeImplication :: Exp -> Exp
```

The two identities we need to use here are:

$$x \leftrightarrow y = (x \rightarrow y) \wedge (y \rightarrow x)$$
$$x \rightarrow y = \neg x \vee y$$

It is important to realize that we have to recursively remove the implications in all of the sub-expressions:

```
-- No implication, no change
removeImplication (Var x)     = (Var x)
removeImplication (T)         = T
removeImplication (F)         = F

-- No implication at top level, try sub-expressions.
removeImplication (Not p)     = Not (removeImplication p)
removeImplication (p :|: q)   = removeImplication p          :|: removeImplication q
removeImplication (p :&: q)   = removeImplication p          :&: removeImplication q

-- Remove implication then try sub-expressions.
removeImplication (p :->: q)  = (Not (removeImplication p))  :|: removeImplication q
removeImplication (p :<->: q) = removeImplication (p :->: q) :&: removeImplication (q :->: p)
```

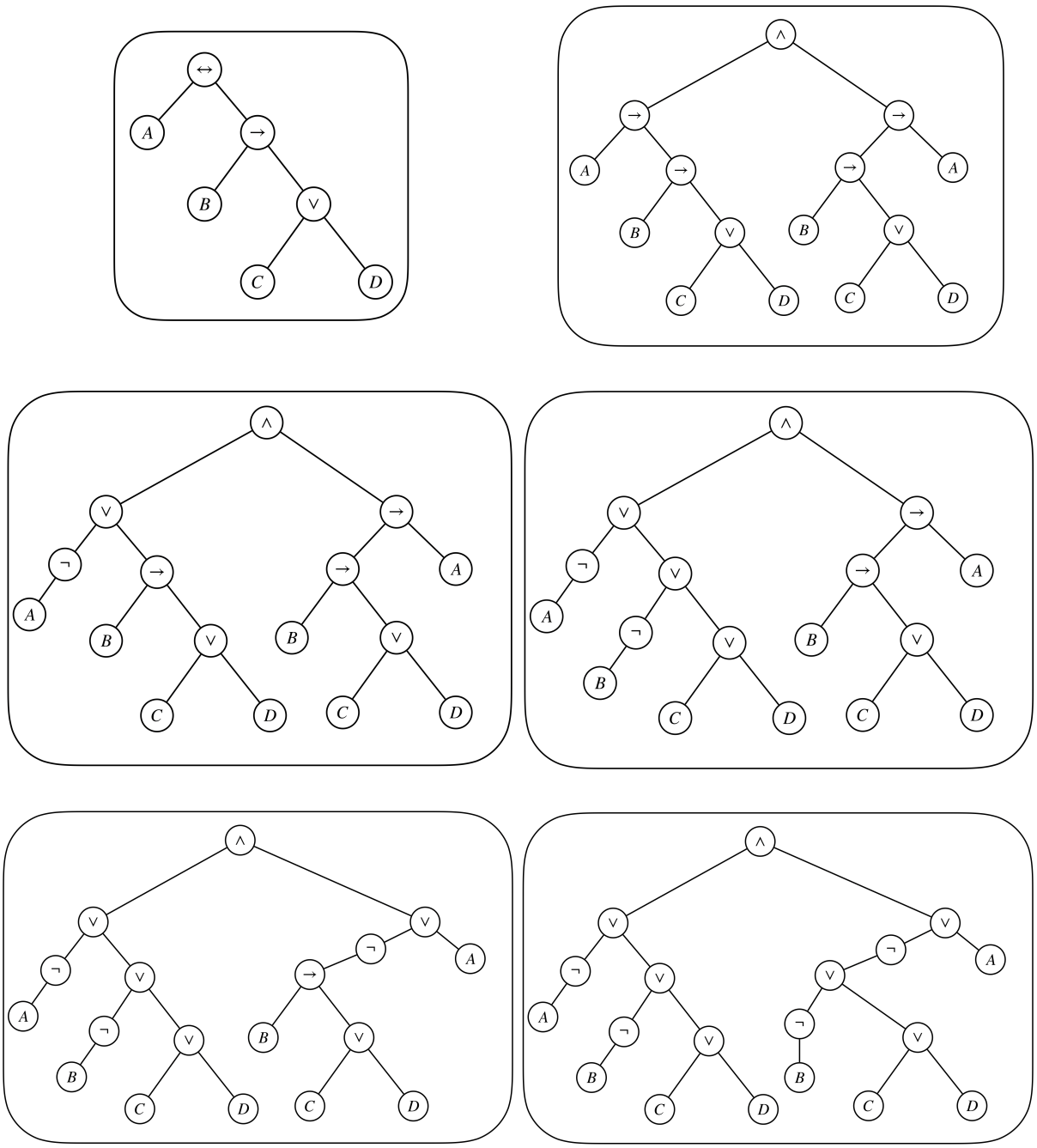
Test using GHCi, you can compare the output with what we did in 'CNF with Boolean Algebra' section:

```
GHCi> showExp $ removeImplication exp1
"(((~A) | ((~B) | (C | D))) & ((~((~B) | (C | D))) | A))"
```

The result is extensively bracketed to avoid ambiguity. Remember that although we know $A \vee B \vee C$, $A \vee (B \vee C)$, and $(A \vee B) \vee C$ are logically equivalent because $\vee$ is associative, computers don't and they get confused.

Please be aware that the visualization does not strictly represent what is really happening.

Here is the step-by-step visualization of implication removal (from left to right, top to bottom):



Step 2: Push negations in.

```
-- Pre-condition, Exp contains no implications
toNNF :: Exp -> Exp
```

We will use De Morgan's laws and the double negation identity:

$$\neg(x \wedge y) = \neg x \vee \neg y$$
$$\neg(x \vee y) = \neg x \wedge \neg y$$
$$\neg\neg x = x$$

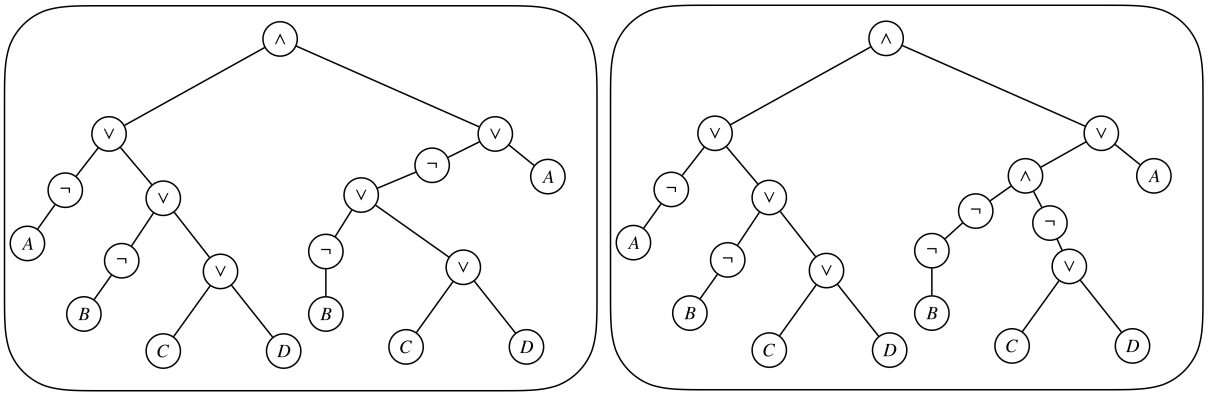
Same as before, we will have to take care of all the sub-expressions.

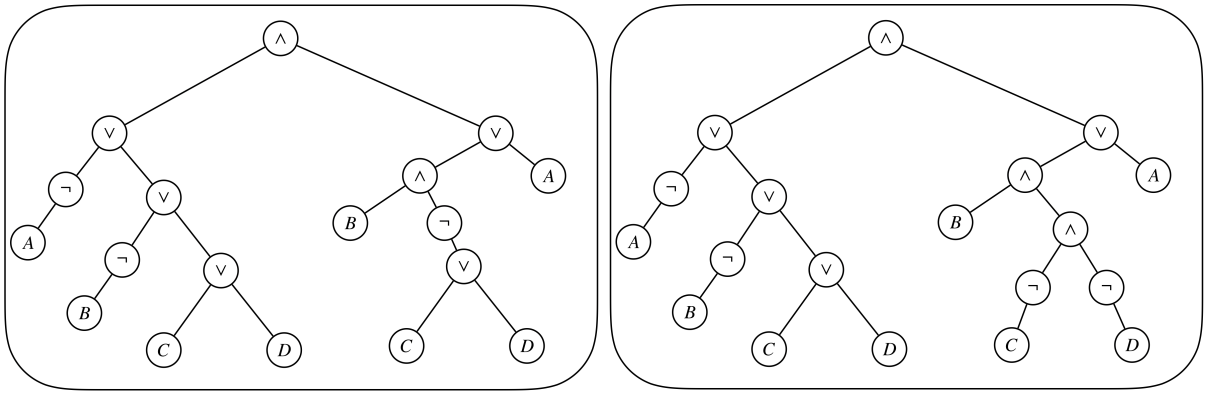
```
-- Literals, no change
toNNF (Var x)         = (Var x)
toNNF (Not (Var x))   = (Not (Var x))
toNNF (T)             = T
toNNF (F)             = F

-- Remove double negations then try sub-expressions
toNNF (Not (Not p))   = toNNF p

-- No negations at top level, try sub-expressions
toNNF (p :&: q)       = toNNF p :&: toNNF q
toNNF (p :|: q)       = toNNF p :|: toNNF q

-- Apply De Morgan's laws then try sub-expressions
toNNF (Not (p :&: q)) = toNNF (Not p) :|: toNNF (Not q)
toNNF (Not (p :|: q)) = toNNF (Not p) :&: toNNF (Not q)
```

Test using GHCi:

```
GHCi> showExp $ toNNF $ removeImplication exp1
"(((~A) | ((~B) | (C | D))) & ((B & ((~C) & (~D))) | A))"
```

Here is the step-by-step visualization for pushing negations in (goes from left to right, top to bottom):





Step 3: Distribute $\vee$ over $\wedge$.

```
-- Pre-condition, Exp is in NNF
distribute :: Exp -> Exp

-- Literals, no change
distribute (Var x)       = (Var x)
distribute (Not (Var x)) = (Not (Var x))
distribute (T)           = T
distribute (F)           = F

-- Conjunction, no change, try sub-expressions
distribute (p :&: q)     = distribute p :&: distribute q

-- Disjunction, first turn the disjuncts into CNF, then use helper to distribute
distribute (p :|: q)     = helper (distribute p) (distribute q)
    where
        helper (a :&: b) c = helper a c :&: helper b c
        helper a (b :&: c) = helper a b :&: helper a c
        helper a b         = a :|: b
```

This step is a bit tricker because it is not trivial to see what we should do with `:|:`. It is a lot easier if we know that both the left and the right hand sides of `:|:` are already in CNFs and we can just use the distribution laws:

$$(a \wedge b) \vee (c \wedge d) = (a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d)$$

This is exactly what `helper` does given that we have CNFs on both sides[23].

[23] This is made sure of by applying `distribute` to both sides first.

Test our functions out using GHCi:

```
GHCi> showExp $ distribute $ toNNF $ removeImplication exp1
"(((~A) | ((~B) | (C | D))) & ((B | A) & (((~C) | A) & ((~D) | A))))"
```

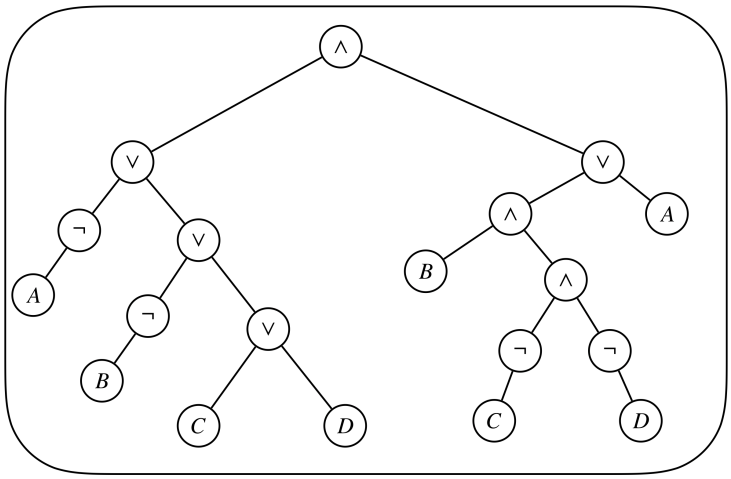
And it is the same as what we got in the Conjunctive Normal Form chapter, great.

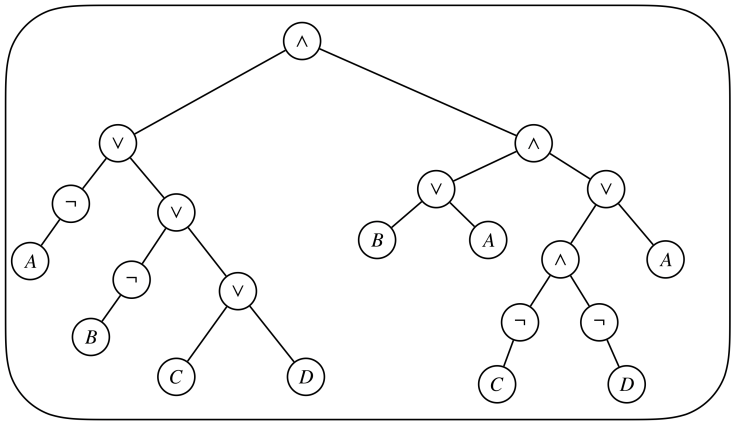
Finally, to tidy things up:

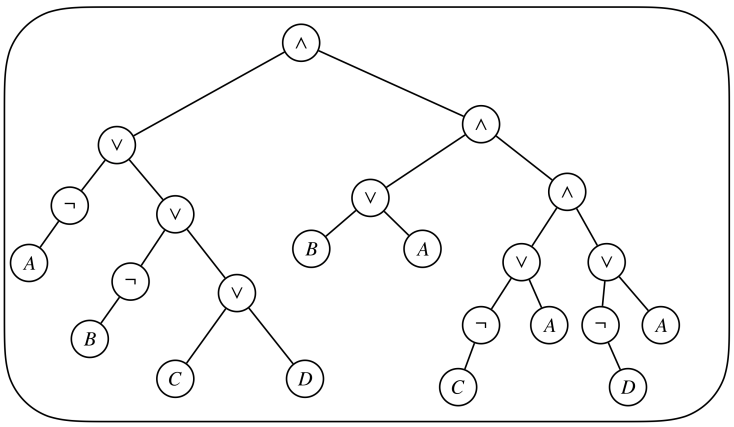
```
cnf :: Exp -> Exp
cnf = distribute . toNNF . removeImplication
```

```
GHCi> showExp $ cnf exp1
"(((~A) | ((~B) | (C | D))) & ((B | A) & (((~C) | A) & ((~D) | A))))"
```

Here is the step-by-step visualization of distribution:







Satisfiability – Introduction

Satisfiability problems are problems where: for a given logical expression, we try to find if there are any **valuations**[24] that can satisfy it *i.e.* if we can make a logical expression true. An expression is satisfiable if there is **at least** 1 satisfying valuation. Here are two simple examples:

- $A \wedge \neg A$ is clearly **not** satisfiable because no matter what truth values we give to A, $A \wedge \neg A$ is always false.
- $A \vee \neg B$ is satisfiable because it **can** be evaluated to true *i.e.* when A is true or B is false.

We usually write down satisfiability relations with entailments. Take $(A \wedge \neg B) \vee C$ for example, one of the satisfying valuations[25] is $A = 1, B = 0$. We put the valuation on the left of the turnstile (as the premise) and the satisfied expression on the right of the turnstile (as the conclusion):

$$A, \neg B \vdash (A \wedge \neg B) \vee C$$

You should think about why this is correct, go back to the definition of entailment if needed.

Sometimes, we want to know more than whether an expression is satisfiable. We might want to know if an expression is always true, we might want to know exactly which valuations satisfy a given expression, etc. Let's first remind ourselves of some keywords:

- **Tautology**: an expression that is always true no matter which valuation is given *e.g.* $A \vee \neg A$ is a tautology:
- **Contradiction**: an expression that is not satisfiable *e.g.* $A \wedge \neg A$ is a contradiction.
- **Contingency**: an expression that is neither a tautology nor a contradiction - an expression that is neither always true nor always false *e.g.* $A \vee \neg B$ is a contingency.

For the rest of this chapter, we will mainly be working with CNFs.

[24] A valuation is an assignment of truth values to propositional variables in a logical expression.

[25] The other one is $C = 1$: $C \vdash (A \wedge B) \vee C$.

Truth Table

The simplest method for checking the satisfiability of an expression is to draw out its truth table.

Show that ϕ is satisfiable where

$$\phi = (A \vee \neg C \vee \neg D) \wedge (A \vee C \vee \neg D) \wedge (A \vee D) \wedge (B \vee \neg C) \wedge A$$

Drawing a truth table should be an easy task for you by now:

A	B	C	D	$(A \vee \neg C \vee \neg D)$	$(A \vee C \vee \neg D)$	$(A \vee D)$	$(B \vee \neg C)$	A	ϕ
0	0	0	0	1	1	0	1	0	0
0	0	0	1	1	0	1	1	0	0
0	0	1	0	1	1	0	0	0	0
0	0	1	1	0	1	1	0	0	0
0	1	0	0	1	1	0	1	0	0
0	1	0	1	1	0	1	1	0	0
0	1	1	0	1	1	0	1	0	0
0	1	1	1	0	1	1	1	0	0
1	0	0	0	1	1	1	1	1	1
1	0	0	1	1	1	1	1	1	1
1	0	1	0	1	1	1	0	1	0
1	0	1	1	1	1	1	0	1	0
1	1	0	0	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Because there is at least one row where ϕ is true (in face, there are six in total), ϕ is satisfiable.

Similarly, if you are asked to show that an expression is a tautology, the last column of the corresponding truth table should only contain 1s (and vice versa for contradictions).

Truth Table using Haskell

Import `nub` and `sort` from `Data.List` module for this section:

```
import Data.List (nub, sort)
```

We will reuse the code from chapter 'CNF using Haskell' where we defined `Exp` type and function `eval`.

First encode ϕ in Haskell:

```
phi :: Exp
phi = (Var "A" :|: Not (Var "C") :|: Not (Var "D"))
  :&: (Var "A" :|: Var "C" :|: Not (Var "D"))
  :&: (Var "A" :|: Var "D")
  :&: (Var "B" :|: Not (Var "C"))
  :&: Var "A"
```

We used `eval` to evaluate an expression against a valuation. For example, we can evaluate `phi` against the valuation `val1`:

```
val1 :: Valuation
val1 = [("A", True), ("B", False), ("C", True), ("D", False)]

GHCi> eval val1 phi
False
```

We can now only evaluate `phi` against one valuation at a time but in order to determine whether an expression is satisfiable, we really want Haskell to do all of the work for us by evaluating `phi` against all the possible valuations. For Haskell to generate valuations, it first need to know what variables are present[26]. We can implement a function `findVariables` where:

[26] We know that A, B, C, D are present in ϕ when we provided the valuation `val1`. If we haven't told Haskell those variables are present, Haskell needs to find them itself.

```
findVariables :: Exp -> [String]
findVariables (Var x)     = [x]
findVariables (F)         = []
findVariables (T)         = []
findVariables (Not p)     = findVariables p
findVariables (p :|: q)   = nub (findVariables p ++ findVariables q)
findVariables (p :&: q)   = nub (findVariables p ++ findVariables q)
findVariables (p :->: q)  = nub (findVariables p ++ findVariables q)
findVariables (p :<->: q) = nub (findVariables p ++ findVariables q)

GHCi> findVariables phi
["A","C","D","B"]
```

The definition is by simple recursion similar to `showExp` and `eval`. One thing we have to keep in mind though, is that a variable can appear many times in an expression and we do not want duplicates in the output. Use `nub` to eliminate duplicates in a list.

With all the variables, let's generate the valuations. The main idea is that: **In each valuation, a variable is either true or false**.

```
generateValuations :: [String] -> [Valuation]
generateValuations []     =  [[]]
generateValuations (x:xs)  =  [(x, False):e  | e <- generateValuations xs ]
                           ++ [(x, True ):e  | e <- generateValuations xs ]
```

How `generateValuations` works is best demonstrated by an example. Generate all possible valuations given the list of variables `["A", "B", "C"]`:

```
generateValuations ("A":["B", "C"]) = [("A", False):e  | e <- generateValuations ("B":["C"])]
                                   ++ [("A", True ):e  | e <- generateValuations ("B":["C"])]
```

where

```
generateValuations ("B":["C"]) = [("B", False):e  | e <- generateValuations ("C":[])]
                               ++ [("B", True ):e  | e <- generateValuations ("C":[])]
```

where

```
generateValuations ("C":[]) = [("C", False):e  | e <- generateValuations []]
                            ++ [("C", True ):e  | e <- generateValuations []]
```


where

```
generateValuations [] = [[]]
```

Substitute back:

```
generateValuations ("C":[]) = [[("C", False)], [("C", True )]

generateValuations ("B":["C"]) = [[("B", False), ("C", False)], [("B", False), ("C", True )],
                                  [("B", True), ("C", False)],  [("B", True), ("C", True )]]

generateValuations ("A":["B", "C"]) = [[("A", False), ("B", False), ("C", False)],
                                       [("A", False), ("B", False), ("C", True )],
                                       [("A", False), ("B", True),  ("C", False)],
                                       [("A", False), ("B", True),  ("C", True )],
                                       [("A", True),  ("B", False), ("C", False)],
                                       [("A", True),  ("B", False), ("C", True )],
                                       [("A", True),  ("B", True),  ("C", False)],
                                       [("A", True),  ("B", True),  ("C", True )]]
```

We finally have every function we need to test if an expression is satisfiable:

```
isSatisfiable :: Exp -> Bool
isSatisfiable e = or [eval v e | v<-(generateValuations $ findVariables e)]
```

```
GHCi> isSatisfiable phi
True
```

Great.

A Brief Discussion of Complexity

We have discussed before that generating a truth table is not efficient because the number of possible valuations grows exponentially with respect to the number of variables. An exponential function is a function with the form:

$$f(x) = ab^x \quad \text{where } a \text{ and } b \text{ are constants}$$

In computer science, often $a = 1$ and $b = 2$:

$$f(x) = 2^x$$

If x is the number of variables, then there are 2^n possible valuations to check.

In the real world, we often have expressions that contain hundreds of variables and drawing their truth tables is simply not viable.

x	$f(x) = 2^x$
0	1
10	1,024
20	1,048,576
30	1,073,741,824
40	1,099,511,627,776
50	1,125,899,906,842,624
60	1,152,921,504,606,846,976
70	1,180,591,620,717,411,303,424
80	1,208,925,819,614,629,174,706,176

Table 1: Powers of 2

Satisfiability – 2-SAT

2SAT problems are satisfiability problems where each clause of the CNF in question contains at most two literals. Below is an example of such a CNF:

$$(A \vee \neg B) \wedge (\neg A \vee \neg C) \wedge (A \vee B) \wedge (D \vee \neg C) \wedge (D \vee \neg A)$$

Resolution

The first technique we are going to study for tackling 2-SAT problems is resolution. Consider this CNF:

$$(A \vee C) \wedge (B \vee \neg C)$$

A	B	C	$(A \vee C) \wedge (B \vee \neg C)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

In all of its satisfying valuations – if C is true, then B must be true; if C is false, then A must be true. So no matter what truth value C has, $A \vee B$ must be true. This yields the **resolution rule** below:[27]

$$\frac{A \vee C \qquad B \vee \neg C}{A \vee B} \text{ Resolution}$$

[27] C and $\neg C$ are called **complementary literals**.

Conversely, if $A \vee B$ is satisfiable, so is $(A \vee C) \wedge (B \vee \neg C)$[28] *e. g.* if we satisfied $A \vee B$ by setting A to true and B to false, we can choose to make C false in order to satisfy $(A \vee C) \wedge (B \vee \neg C)$.

[28] They are **equisatisfiable**.

Notice that we only need to keep track of one clause instead of two after resolution.This is essentially what we want to exploit, we keep resolving (applying the resolution rule to) our clauses until one of the two conditions holds:

- We get an empty clause which implies that the original expression is **unsatisfiable**. An empty clause arises when we resolve *e. g.* A and $\neg A$ which is clearly unsatisfiable.
- No clauses can be resolved (none of the clauses contain complementary literals) and the original expression is **satisfiable**.

We will discuss more about correctness at the end of the chapter. Just follow the procedure mechanically for now.

For compactness, we usually don't write down the whole CNF when doing resolution by hand. Instead, we only write down the clauses and we put them vertically in a table. Take the CNF at the top of this page for example:

$$\begin{aligned}&(A \vee \neg B) \wedge (\neg A \vee \neg C) \wedge (A \vee B) \wedge (D \vee \neg C) \wedge (D \vee \neg A) && \text{in CNF}\\ \equiv &\{\{A, \neg B\}, \{\neg A, \neg C\}, \{A, B\}, \{D, \neg C\}, \{D \neg A\}\} && \text{in clausal form}\end{aligned}$$

Take all the clauses and put them into a table like this:

Original	
1. $(A, \neg B)$	
2. $(\neg A, \neg C)$	
3. (A, B)	
4. $(D, \neg C)$	
5. $(D, \neg A)$	

Notice that I used round brackets instead of curly brackets in the table because I personally think it's easier to write round brackets by hand. You also don't need to number the clauses.

First resolve the clauses containing variable A. Clauses 1 and 3 contain A; clauses 2 and 5 contain $\neg A$. Apply the resolution rule to all of the pairs which are $(1,2), (1,5), (3,2), (3,5)$ and put the clauses produced[29] into the second column of the table. At the same time cross out the resolved clauses.

[29] A clause produced by resolution is called a **resolvent**.

Original	A	
~~1. $(A, \neg B)$~~	6. $(\neg B, \neg C)$	
~~2. $(\neg A, \neg C)$~~	7. $(\neg B, D)$	
~~3. (A, B)~~	8. $(B, \neg C)$	
4. $(D, \neg C)$	9. (B, D)	
~~5. $(D, \neg A)$~~		

Repeat the same procedure for clauses containing variable B. Clauses 8 and 9 contain B. Clauses 6 and 7 contain $\neg B$. Apply the rule to pairs $(8,6), (8,7), (9,6), (9,7)$:

Original	A	B	C	D
~~1. $(A, \neg B)$~~	~~6. $(\neg B, \neg C)$~~	10. $(\neg C)$		
~~2. $(\neg A, \neg C)$~~	~~7. $(\neg B, D)$~~	11. $(\neg C, D)$		
~~3. (A, B)~~	~~8. $(B, \neg C)$~~	12. (D)		
4. $(D, \neg C)$	~~9. (B, D)~~			
~~5. $(D, \neg A)$~~				

We discard duplicate clauses.

Now no clauses can be resolved. The resolution is completed and the original expression is satisfiable.

From the table, we can easily find out the satisfying valuation by trying to make every clause true. Notice that **unit clauses**[30] must be satisfied[31] *i.e.* $\neg C$ and D must be satisfied. Now with $\neg C$ and D being true, clauses that contain $\neg C$ or D *i.e.* $2, 4, 5, 6, 7, 8, 9, 10, 11, 12$ also become true and we don't need to check them any more. With clauses 1 and 3 left, it is clear that we have to make A true and B can be whatever. Finally, we have an satisfying valuation – $\mathbf{V} = \{A, B, \neg C, D\}$.

[30] Clauses with only one literal.

[31] For example, you have to satisfy A in $(A) \wedge (...) \vee (...)$ because the clauses are connected conjunctively.

Let's look at a CNF that is unsatisfiable.

$$(A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee C) \wedge (\neg A \vee \neg C)$$

Resolve the clauses containing variable A:

Original	A
~~1. (A, B)~~	5. (B, C)
~~2. $(A, \neg B)$~~	6. $(B, \neg C)$
~~3. $(\neg A, C)$~~	7. $(\neg B, C)$
~~4. $(\neg A, \neg C)$~~	8. $(\neg B, \neg C)$

Then the ones containing variable B:

Original	A	B
~~1. (A, B)~~	~~5. (B, C)~~	
~~2. $(A, \neg B)$~~	~~6. $(B, \neg C)$~~	9. (C)
~~3. $(\neg A, C)$~~	~~7. $(\neg B, C)$~~	10. $(\neg C)$
~~4. $(\neg A, \neg C)$~~	~~8. $(\neg B, \neg C)$~~	

We don't need to add trivial true clauses like $(C \vee \neg C)$.

Lastly C:

Original	A	B	C
~~1. (A, B)~~	~~5. (B, C)~~		
~~2. $(A, \neg B)$~~	~~6. $(B, \neg C)$~~	~~9. (C)~~	$()$
~~3. $(\neg A, C)$~~	~~7. $(\neg B, C)$~~	~~10. $(\neg C)$~~	
~~4. $(\neg A, \neg C)$~~	~~8. $(\neg B, \neg C)$~~		

We arrived at an empty clause and the original expression is unsatisfiable.

k-SAT with Resolution

We have used resolution to solve 2-SAT problems but it can be applied generally to k-SAT problems[32] as well. The generalized resolution rule is as follows:

$$\frac{A_1 \vee A_2 \vee A_3 \vee ... \vee C \qquad B_1 \vee B_2 \vee B_3 \vee ... \vee \neg C}{A_1 \vee A_2 \vee A_3 \vee ... \vee B_1 \vee B_2 \vee B_3 \vee ...} \text{ Resolution}$$

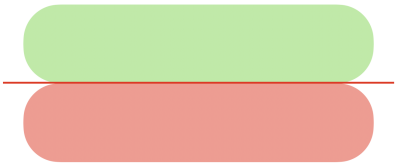
[32] $k \in \mathbb{Z}^+$, there are at most k literals in a clause.

It is applied in much of the same way so I am not going to talk much about it. For those who are interested in computational complexity, 2-SAT can be solved in polynomial time but only exponential time methods are known for $(k > 2)$-SAT.

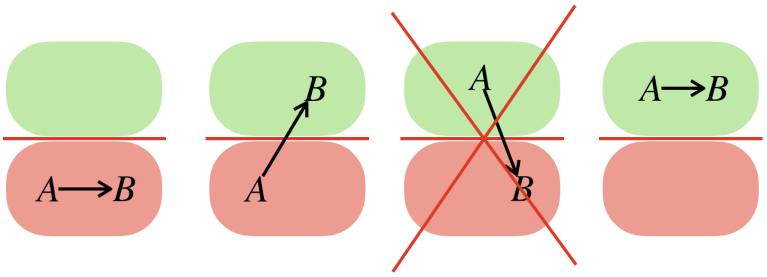
So basically another reason to why I'm not providing $(k > 2)$-SAT examples is that the tables get big and I don't feel like making them any more o_O.

Implication Graph

Let's transform our understanding of implication ($\rightarrow$) into graphs like this:



We put all the literals that are true above the red line and the ones that are false below the red line. Copying directly from the truth table of $A \rightarrow B$, we can draw the four different graph configurations:



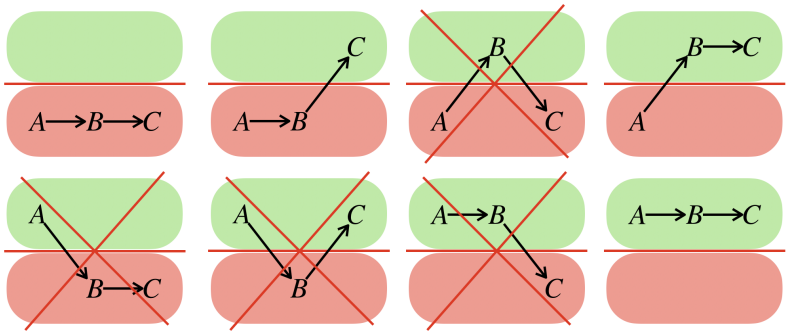
We use top $\top$ and bottom $\bot$ instead of 1 and 0 to represent true and false in this section because we are going to put true literals on top of and false literals below a line.

A	B	$A \rightarrow B$
$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$

Just as $\top \rightarrow \bot = \bot$, **an arrow going from green to red (true to false) is an illegal configuration** in our graphs. Let's verify this property with a slightly more complex example:

$$(A \rightarrow B) \wedge (B \rightarrow C)$$

A	B	C	$(A \rightarrow B) \wedge (B \rightarrow C)$
$\bot$	$\bot$	$\bot$	$\top$
$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$
$\top$	$\bot$	$\top$	$\bot$
$\top$	$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$	$\top$



Why is implication useful for 2-SAT? Well, first remember that we can turn disjunctions into implications by using the relation:

$$A \vee B = \neg A \rightarrow B = \neg B \rightarrow A$$

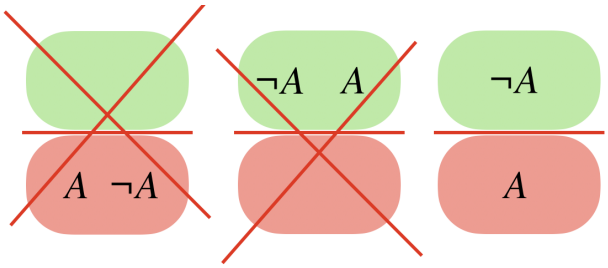
It means that we can then turn every 2-SAT CNF into its **implicative normal form** *e.g.*

$$\begin{aligned} &(\neg A \vee B) \wedge (\neg B \vee C) \wedge (\neg C \vee D) \\ =&(A \rightarrow B) \wedge (B \rightarrow C) \wedge (C \rightarrow D) \end{aligned}$$

We can construct the graph of an expression in implicative normal form and use the graph to quickly decide its satisfiability. In fact we can even obtain all of its satisfying valuations.

Before we dive in to the examples, I would like to emphasize that complimentary literals should never appear on the same side of a graph[33]. Although there's no problem with them appearing on different sides. *e.g.*:

[33] Complimentary literals can't have the same truth values *i.e.* A and $\neg A$ can't be both true.



Example 1 – $(\neg A \vee B) \wedge (\neg B \vee C) \wedge (\neg C \vee D)$

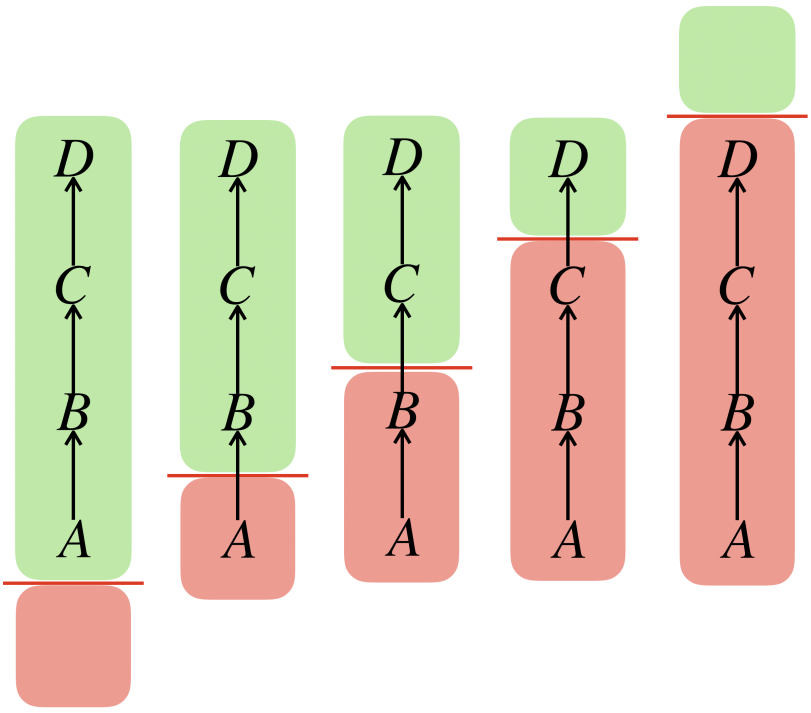
Turn the expression into its implicative normal form:

$$(\neg A \vee B) \wedge (\neg B \vee C) \wedge (\neg C \vee D)$$
$$= (A \to B) \wedge (B \to C) \wedge (C \to D)$$

Graph it with arrows going from bottom to top (graph in the margin).

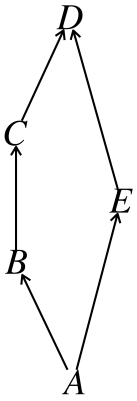
Then decide where to cut[34] so there aren't any arrows going from green to red (true to false), there are 5 options in this graph:

[34] Where to put the red line, it's like cutting the graph in half.



In conclusion, $(\neg A \vee B) \wedge (\neg B \vee C) \wedge (\neg C \vee D)$ is satisfiable and there are 5 satisfying valuations:

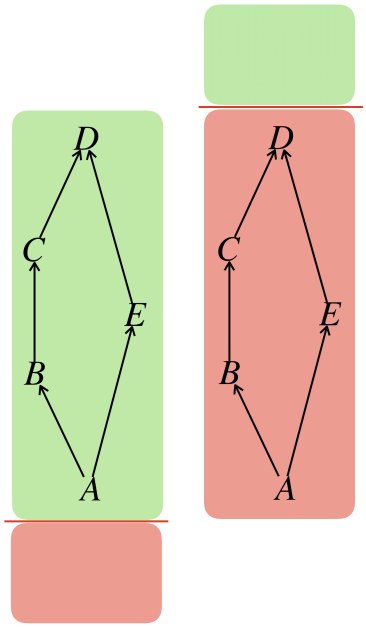
$$\mathbf{V_1} = \{A, B, C, D\}$$
$$\mathbf{V_2} = \{\neg A, B, C, D\}$$
$$\mathbf{V_3} = \{\neg A, \neg B, C, D\}$$
$$\mathbf{V_4} = \{\neg A, \neg B, \neg C, D\}$$
$$\mathbf{V_5} = \{\neg A, \neg B, \neg C, \neg D\}$$



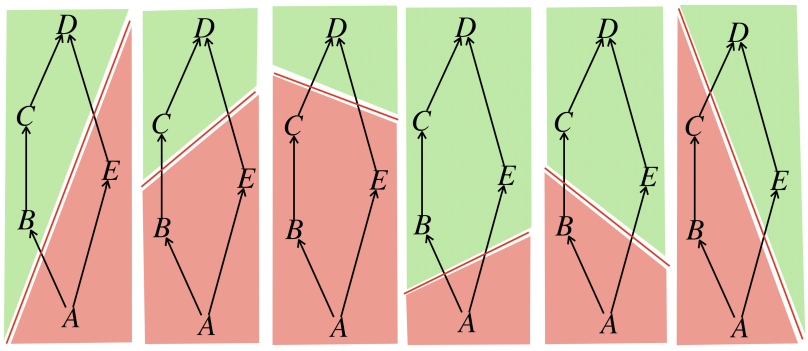
Example 2 – $(A \to B) \wedge (B \to C) \wedge (C \to D) \wedge (A \to E) \wedge (E \to D)$

The graph of the expression looks a bit different (graph in margin).

At first glance, it might look like there aren't any legal cuts beside the two trivial ones:



But we can indeed cut through the middle, we just need to keep in mind that **we don't allow arrows going from green to red**[35]:



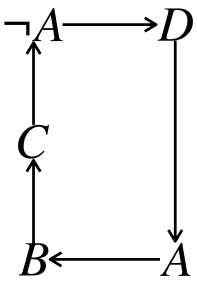
In conclusion, the original expression is satisfiable and there are 8 satisfying valuations.

[35] Tips: The number of cuts that can be made through a 'loop' similar to this is the product of the number of edges on one side and the number of edges on the other. In this case, it is simply $3 \times 2 = 6$ because there are 3 edges in $A \to B \to C \to D$ and 2 edges in $A \to E \to D$.

I know this sounds confusing, come ask me in person if you don't get it.

Example 3 – $(A \to B) \wedge (B \to C) \wedge (C \to \neg A) \wedge (\neg A \to D) \wedge (D \to A)$

Here is the graph:



The expression is unsatisfiable. Why?

Hint: The two rules we follow are:

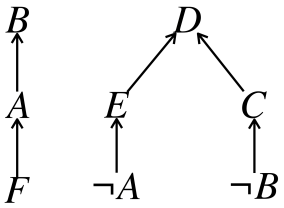
- No arrows go from true to false.
- Complimentary literals can't share the same truth value.

For people who know some graph theory: If complementary literals appear in the same strongly connected component of a graph, the expression the graph represents is unsatisfiable.

And for those who are interested in complexity: 2-SAT solving using implication graph can run in $O(n)$ time. Use adjacency lists to represent graphs and use DFS to search for strongly connected components.

Example 4 – $(A \to B) \wedge (\neg B \to C) \wedge (C \to D) \wedge (\neg A \to E) \wedge (E \to D) \wedge (F \to A)$

If we draw the graph directly without first modifying the expression, we will get a graph with two separate components:

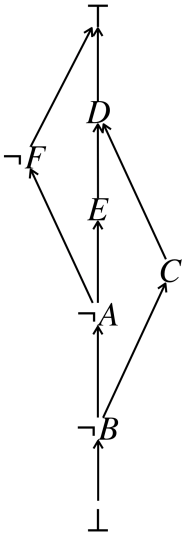


It is very hard for us to decide where to cut because we have to keep complementary literals separated when we are cutting through both components. **In this case**, we can actually pre-process the expression to eliminate complimentary literals and make the graph fully connected. We turn two of the implications to their contrapositives (marked in blue and red):

$$(A \to B) \wedge (\neg B \to C) \wedge (C \to D) \wedge (\neg A \to E) \wedge (E \to D) \wedge (F \to A)$$
$$\equiv (\neg B \to \neg A) \wedge (\neg B \to C) \wedge (C \to D) \wedge (\neg A \to E) \wedge (E \to D) \wedge (\neg A \to \neg F)$$

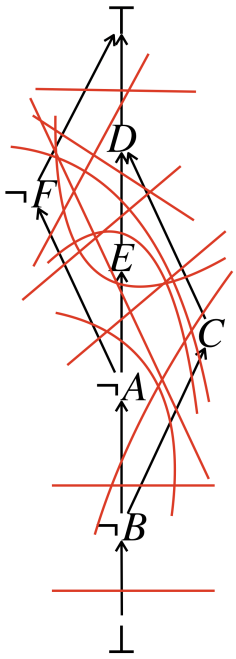
Use the 'new' expression to construct the graph (graph in the margin).

Notice that I have added $\top$ and $\bot$ to the endpoints of the graph. They don't affect the final result as both $(\bot \to ?)$ and $(? \to \top)$ are always true. They serve as indicators to the truth value of their respective sides[36].



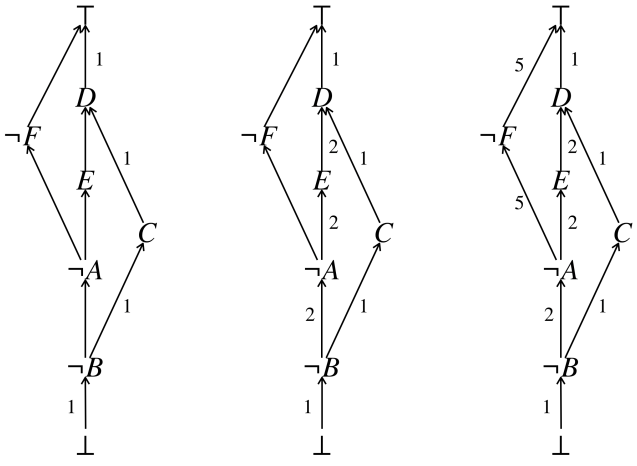
[36] So that every literal on the side which contains $\bot$ is false and vice versa.

We now can find the legal cuts as usual:

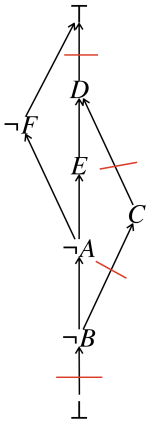


To make things simpler, I have taken the coloured boxes away and made every legal cut in one graph. You can still imagine the left side of the arrow being red and right side being green though.

Hmmm.. it doesn't look good, does it? It seems like we need a different notation for graphs with many cuts. We can count the possible cuts instead:



Step 1 would look something like this if we do the cuts:



Suppose we are cutting from the right to the left of the graph. We cut in steps so that in each step, we only cut through 1 edge (thus the cut may be incomplete and stuck in a 'loop'). In the first step, there is always only 1 way to cut through any of the edges between $\bot \to \neg B \to C \to D \to \top$. Notice that the cut between $\bot \to \neg B$ is complete so we put the 1 on the other side.

In step 2, continue cutting with the incomplete cuts from last round, there are $1 + 1 = 2$ ways to cut through the edges between $\neg B \to \neg A \to E \to D$. Also notice that the cut between $\neg B \to \neg A$ is complete.

Do the same procedure in step 3 and we will have the last 2 edges cut through.

Lastly, count the sum of the numbers on the left side and that would be the number of satisfying valuations. So in conclusion, there are $1 + 2 + 5 + 5 = 13$ satisfying valuations for the original expression.

This cutting / numbering process is really best explained with pen and paper. If you had no clue what I was talking about, you are welcome to ask me in person.

Make Literals Pure!

If an implicative normal form contains complementary literals, it is usually quite difficult for us to find all the legal cuts in its implication graph because we have to keep the complimentary literals separated.

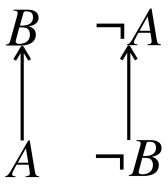
We saw in Example 4 that it was possible to make all the literals pure[37] by turning some implications to their contrapositives using the rule $A \to B = \neg B \to \neg A$. You might have wondered how did we know that all the literals can be made pure?

[37] A literal is pure if its compliment does not occur *i.e.* there aren't complimentary literals.

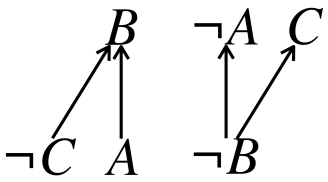
For every implication in the expression, we draw two arrows: one for the implication and one for its contrapositive. Take the expression from Example 4 for example:

$$(A \to B) \wedge (\neg B \to C) \wedge (C \to D) \wedge (\neg A \to E) \wedge (E \to D) \wedge (F \to A)$$

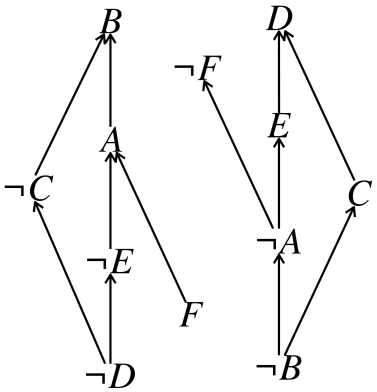
For $(A \to B)$, we draw both $A \to B$ and $\neg B \to \neg A$:



Then for $(\neg B \to C)$, we draw both $\neg B \to C$ and $\neg C \to B$:



Repeat the same procedure for the rest and you will eventually get two separate components that are dual (graph in margin, if you reverse all the arrows and negate all the literals in the component on the left, you will get the component on the right) to each other.



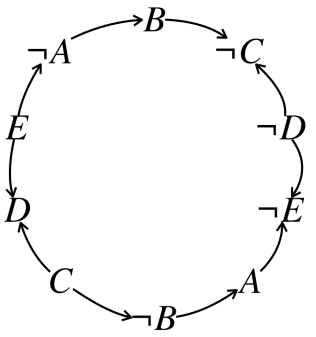
For any expression, if its literals can be made pure, the graph (in which you draw both the implication and its contrapositive) necessarily separates into two components[38]. You can use either component to do your cuts.

Sometimes, it is just impossible to make the literals pure and then it all becomes a little tricky. We will look at one of those in the next and final example.

[38] Why is this true?

Example 5 – $(\neg A \to B) \wedge (B \to \neg C) \wedge (C \to D) \wedge (A \to \neg E) \wedge (E \to D)$

There are complimentary literals so we start by drawing the implications and their contrapositives and see if we get two separated components:



(°-°) Nope... It means that we can't make all the literals pure. You can still try to cut as usual but it really is just a nightmare doing that.

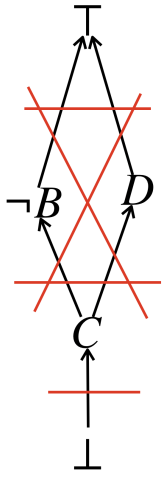
Fear not, we can still reason our way through! We can start by making A true[39]:

$$(\neg A \to B) \wedge (B \to \neg C) \wedge (C \to D) \wedge (A \to \neg E) \wedge (E \to D)$$

A being true forces $\neg E$ to be true[40]:

$$(\neg A \to B) \wedge (B \to \neg C) \wedge (C \to D) \wedge (A \to \neg E) \wedge (E \to D)$$

With only $(B \to \neg C) \wedge (C \to D)$ left to satisfy, we can make its implication graph:



So if we set A to true, there are 5 satisfying valuations. Now we just need to set A to false instead, repeat the process, and sum the results, this is left as an exercise for you[41].

There should be 8 satisfying valuations in total.

[39] Of course $\neg A$ becomes false at the same time and $(\neg A \to B)$ becomes true.

[40] Again, E becomes false at the same time and $(E \to D)$ becomes true.

[41] I hate it when authors leave exercises to the readers but I believe that you are better readers than I am ;D

Intuition on Correctness of Resolution

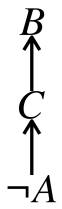
Intuition on resolution's correctness is very easy to establish using implication graphs. Let me remind you of the resolution rule:

$$\frac{A \vee C \qquad B \vee \neg C}{A \vee B} \text{ Resolution}$$

If we turn the disjunctions into implications:

$$\frac{\neg A \to C \qquad C \to B}{\neg A \to B}$$

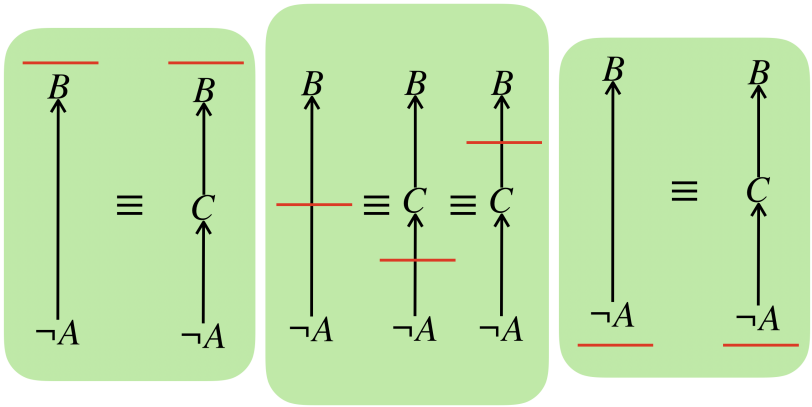
Plot the implication graph of the premises:



And the graph of the conclusion:

$(A \vee C) \wedge (B \vee \neg C)$ and $(A \vee B)$ are equisatisfiable because no matter how we cut the graph of $(A \vee B)$, you can cut at the 'same' place in the graph of $(A \vee C) \wedge (B \vee \neg C)$[42].

[42] If the cut is illegal in one graph, it'd be illegal in the other one as well.



This is why we could discard the resolved clauses during resolution and keep only the resolvent. It is sufficient to satisfy the clauses that are left at the end.

Resolved clauses are often kept during resolution which isn't necessary.

Satisfiability – DPLL Algorithm

This algorithm was introduced by Martin **D**avis, Hilary **P**utnam, George **L**ogemann, and Donald **L**oveland. It is a **crucial** algorithm for SAT solving to date.

Unit Propagation

Unit propagation (UP)[43] is the core procedure in DPLL. Consider the following expression in clausal normal form:

[43] Sometimes called Boolean constraint propagation (BCP).

$$\phi = \{\{\neg A, \neg C, \neg D\}, \{A, C, \neg D\}, \{A, D\}, \{B, \neg C\}, \{A\}\}$$

It doesn't take long to for us to realize that – In a conjunction, literals in **unit clauses**[44], *i.e.* A in $\{A\}$, **must** be satisfied.

[44] Clauses that contain only one literal.

To satisfy $\{A\}$, we make A true in our **partial valuation**[45]:

[45] The valuation is partial because it doesn't tell us the truth value of all the variables

$$\mathbf{V} = \{A\}$$

With A being true:

- All the clauses that contain A are true and we can remove those clauses because "$\top \wedge$ something" is the something:

$$\top \wedge (A \vee B \vee ...) \wedge (C \vee D \vee ...) \wedge ... = (A \vee B \vee ...) \wedge (C \vee D \vee ...) \wedge ...$$

- $\neg A$ becomes false and we can remove $\neg A$ from every clause because "$\bot \vee$ something" is the something:

$$(\bot \vee A \vee B \vee ...) = (A \vee B \vee ...)$$

So ϕ can be simplified to:

$$\phi = \{\{\neg C, \neg D\}, \{B, \neg C\}\}$$

DPLL uses this procedure repetitively with **backtracking**[46] until either:

[46] If a partial valuation leads to contradiction (an empty clause), you go back and try again. We will demonstrate this with an example.

- A full satisfying valuation is found and the expression is SAT.
- All valuations are rejected and the expression is UNSAT.

[47] Dpll algorithm, May 2019b. URL https://en.wikipedia.org/wiki/DPLL_algorithm

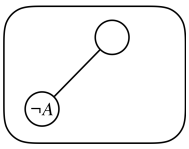
Example – Unit Propagation and Backtracking

Let's apply DPLL to the following expression taken from Wikipedia[47]:

$$\{\{\neg A, B, C\}, \{A, C, D\}, \{A, C, \neg D\}, \{A, \neg C, D\}, \{A, \neg C, \neg D\}, \{\neg B, \neg C, D\}, \{\neg A, B, \neg C\}, \{\neg A, \neg B, C\}\}$$

If there is no unit clause, we just choose a literal to propagate. For now, we can simply make the first literal we see true *i.e.*:

$$\mathbf{V} = \{\neg A\}$$



Remove the clauses that contain $\neg A$:

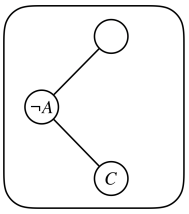
$$\{\{A, C, D\}, \{A, C, \neg D\}, \{A, \neg C, D\}, \{A, \neg C, \neg D\}, \{\neg B, \neg C, D\}\}$$

Remove A from remaining clauses:

$$\{\{C, D\}, \{C, \neg D\}, \{\neg C, D\}, \{\neg C, \neg D\}, \{\neg B, \neg C, D\}\}$$

There still no unit clause, we will make C true as well:

$$\mathbf{V} = \{\neg A, C\}$$

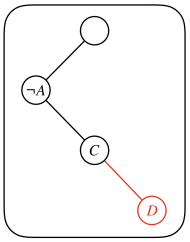


Remove clauses that contain C and remove literals $\neg C$:

$$\{\{D\}, \{\neg D\}, \{\neg B, D\}\}$$

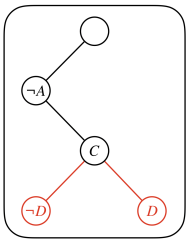
There is a unit clause $\{D\}$, **unit propagate** D:

$$\mathbf{V} = \{\neg A, C, D\}$$
$$\{\{\}\}$$



We get an empty clause which makes the expression false and we need to **backtrack** by assigning the opposite truth value to the last variable in the partial valuation:

$$\mathbf{V} = \{\neg A, C, \neg D\}$$
$$\{\{\}\}$$

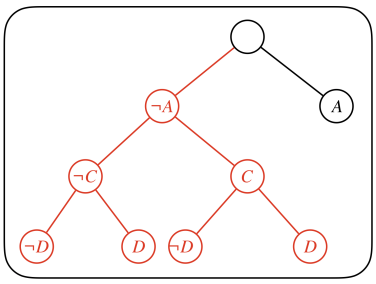


We get an empty clause again and we need to **backtrack further**:

$$\mathbf{V} = \{\neg A, \neg C\}$$
$$\{\{D\}, \{\neg D\}\}$$

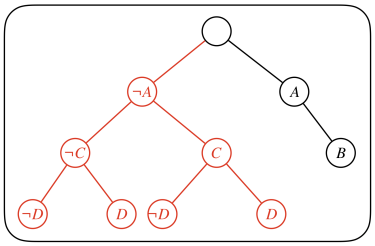
This clearly won't work either (we skip some steps here), we need to **backtrack even further back**:

$$\mathbf{V} = \{A\}$$
$$\{\{B, C\}, \{\neg B, \neg C, D\}, \{B, \neg C\}, \{\neg B, C\}\}$$



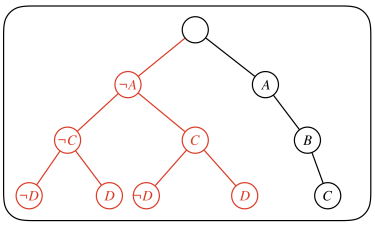
Add B to the valuation:

$$\mathbf{V} = \{A, B\}$$
$$\{\{\neg C, D\}, \{C\}\}$$



Unit propagate C:

$$\mathbf{V} = \{A, B, C\}$$
$$\{\{D\}\}$$



Finally, **unit propagate** D:

$$\mathbf{V} = \{A, B, C, D\}$$
$$\{\}$$

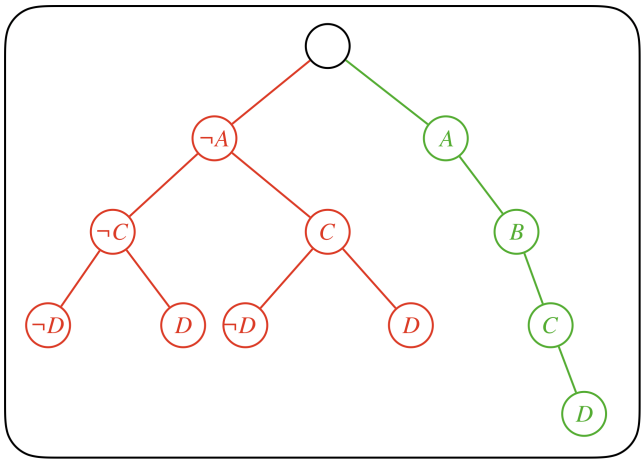


Figure 2: Completed DPLL Tree

DPLL found a satisfying valuation and the algorithm halts.

Haskell Implementation

Preparation

In the Syntax of Propositional Logic in Haskell chapter, we defined the `Exp` type and wrote a script to transform arbitrary expressions into CNFs:

```
GHCi> cnf (Var "A" :<->: (Var "B" :->: (Var "C" :|: Var "D")))
(Not (Var "A") :|: (Not (Var "B") :|: (Var "C" :|: Var "D"))) :&:
((Var "B" :|: Var "A") :&:
((Not (Var "C") :|: Var "A") :&:
(Not (Var "D") :|: Var "A")))
```

We can certainly still work with CNFs in the form of an `Exp`. But we can (and should) find a different data structure to represent CNFs because unlike an arbitrary expressions, we know exactly what a CNF "looks like" – a CNF is a **conjunction** of **disjunctions** of **literals**.

Let's tackle the bold keywords one by one:

- A **literal** is just an atom or its negation. Let's introduce a new data type `Literal`:

  ```
  data Literal a = P a | N a deriving (Eq, Show, Ord)
  ```

  It states that a literal is either a **P**ositive "thing" or an **N**egative "thing". The type of that "thing" can be whatever you want (depending on your use case) as long as it can be tested for equality.

- We represent a **disjunction** of literals, *i.e.* a clause, as a list of literals:

  ```
  type Clause a = [Literal a]
  ```

- And a **conjunction** of disjunctions of literals, *i.e.* a CNF, as a list of lists of literals:

  ```
  type CNF a = [Clause a]
  ```

We can write the CNF from the previous example

$$\{\{\neg A, B, C\}, \{A, C, D\}, \{A, C, \neg D\}, \{A, \neg C, D\}, \{A, \neg C, \neg D\}, \{\neg B, \neg C, D\}, \{\neg A, B, \neg C\}, \{\neg A, \neg B, C\}\}$$

down as:

```
e :: CNF String
e = [[N "A", P "B", P "C"],
     [P "A", P "C", P "D"],
     [P "A", P "C", N "D"],
     [P "A", N "C", P "D"],
     [P "A", N "C", N "D"],
     [N "B", N "C", P "D"],
     [N "A", P "B", N "C"],
     [N "A", N "B", P "C"]]
```

Before we implement the algorithm, let's also define a type synonym for valuations:

```
type Valuation a = [Literal a]
```

and a helper function that negates literals will come in handy:

```
neg :: Literal a -> Literal a
neg (P atom) = N atom
neg (N atom) = P atom
```

Implementation

Below is a straightforward Haskell implementation of DPLL:

```
dpll :: Eq a => CNF a -> Valuation a -> Valuation a
dpll e v
    | e == [] = v
    | elem [] e = []
    | units /= [] = dpll (propagate unitLit e) (v ++ [unitLit])
    | otherwise = dpll (propagate lit e) (v ++ [lit])
            >||< dpll (propagate (neg lit) e) (v ++ [(neg lit)])
       where
           units = filter (\x -> (length x) == 1) e
           unitLit = head $ head units
           lit = head $ head e
           propagate n e = map (\\ [neg n]) $ filter (notElem n) e
           (>||<) x y = if x /= [] then x else y
```

\\ is the list difference which is in Data.List

Given a CNF expression and a partial valuation, we first check whether the CNF is satisfied or contradicted by the valuation. If it is satisfied, we output the valuation. If it is contradicted, we output an empty valuation.

Test satifaction/contradiction

If the CNF is neither satisfied nor contradicted by the partial valuation, we first check if there are any unit clauses. If there are, we propagate a unit literal and then call `dpll` with the new expression and valuation.

Unit propagate

If there aren't any unit clauses, we propagate the first literal we see. If we end up getting an empty valuation, we try again with the literal's negation.

Backtrack

Finally, let's find a satisfying valuation of the example CNF:

```
GHCi> dpll e []
[P "A",P "B",P "C",P "D"]
```

:-Þ

Case Study: Sudoku

TL;DR: Go to the last page of this chapter where we play with the code.

Problem Description

Sudoku is a puzzle where you fill numbers in a 9x9 grid. Of course, you don't just fill in numbers randomly and there are some constraints. Below is a typical Sudoku puzzle:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

We use a list of lists of `Int` to represent a Sudoku grid:

```
type Grid = [[Int]]
```

So the example grid looks like:

```
exampleGrid :: Grid
exampleGrid = [[5,3,0, 0,7,0, 0,0,0],
               [6,0,0, 1,9,5, 0,0,0],
               [0,9,8, 0,0,0, 0,6,0],

               [8,0,0, 0,6,0, 0,0,3],
               [4,0,0, 8,0,3, 0,0,1],
               [7,0,0, 0,2,0, 0,0,6],

               [0,6,0, 0,0,0, 2,8,0],
               [0,0,0, 4,1,9, 0,0,5],
               [0,0,0, 0,8,0, 0,7,9]]
```

0 means that the cell is not filled.

You are given a partially filled grid and your task is to fill the whole grid under these constraints:

- Every cell must contain exactly one value from 1 to 9.
- Every row must contain all the numbers from 1 to 9.
- Every column must contain all the numbers from 1 to 9.
- Every block (those smaller 3x3 grids) must contain all the numbers from 1 to 9.

Below is the puzzle completed:

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Encode Sudoku into CNF

We index the Sudoku grid in the way you would normally index a 2D array using two indices i and j where i is the row number and j is the column number. Both i and j range from 1 to 9. To assert that a number n is filled in the cell at (i, j), we write:

```
P (i, j, n)
```

To assert that n is **not** filled in cell (i, j), we write:

```
N (i, j, n)
```

Encode each of the following constraints in Haskell:

- Every cell must contain exactly one value from 1 to 9. We can break this constraint down into two smaller constraints:
  - Every cell must contain **at least one** value from 1 to 9.
    Take cell $(1,1)$ for example, we need the nine-ary clause
    $\{\mathrm{P}\ (1,1,1),\ \mathrm{P}\ (1,1,2),\ \mathrm{P}\ (1,1,3),\ \mathrm{P}\ (1,1,4),\ \mathrm{P}\ (1,1,5),\ \mathrm{P}\ (1,1,6),\ \mathrm{P}\ (1,1,7),\ \mathrm{P}\ (1,1,8),\ \mathrm{P}\ (1,1,9)\}$
    We can easily achieve this using list comprehension:

    ```
    atLeastOne' = [[P (i, j, n) | n <- [1..9]] | i <- [1..9], j <- [1..9]]
    ```

  - Every cell must contain **no more than one** value from 1 to 9.
    This means all the literal pairs in each clause from the previous step can not be true at the same time. For example, $\mathrm{P}\ (1,1,1)$ and $\mathrm{P}\ (1,1,2)$ cannot both be true, which gives the clause $\mathrm{N}\ (1,1,1),\ \mathrm{N}\ (1,1,2)$.

```
    noMoreThanOne' = [[neg x, neg y] | row <- atLeastOne', (x:xs) <- tails row, y <- xs]
```

- Every row must contain all the numbers from 1 to 9.
- Every column must contain all the numbers from 1 to 9.
- Every block (those smaller 3x3 grids) must contain all the numbers from 1 to 9.

The other three constraints are encoded in the same fashion and they are left as exercises for the reader... If you don't want an exercise, the complete code snippet is on the next page.

There is actually one more constraint we haven't talked about – we need to encode the pre-filled numbers as well. That is pretty trivial to do and it really is a good exercise to test your functional thinking.

Code Snippet

Below is a tidied-up version of the Sudoku puzzle encoding:

```
module Encode where
import Data.List (tails)
import Grid
import DPLL (Literal (..), CNF (..), neg)

enc :: Grid -> CNF (Int, Int, Int)
enc g = prefilled g ++ noMoreThanOne ++ atLeastOne

prefilled :: Grid -> CNF (Int, Int, Int)
prefilled g = [[P (i, j, n)] |
               (i, row) <- zip [1..] g,
               (j, n) <- zip [1..] row,
               n /= 0]

atLeastOne :: CNF (Int, Int, Int)
atLeastOne = [[P (i, j, n) | n <- [1..9]] | i <- [1..9], j <- [1..9]] -- Cells
          ++ [[P (i, j, n) | j <- [1..9]] | i <- [1..9], n <- [1..9]] -- Rows
          ++ [[P (i, j, n) | i <- [1..9]] | j <- [1..9], n <- [1..9]] -- Columns
          ++ [[P (i, j, n) | i <- [ii..ii + 2], j <- [jj..jj + 2]] |
              ii <- [1,4,7], jj <- [1,4,7], n <- [1..9]] -- Blocks

noMoreThanOne :: CNF (Int, Int, Int)
noMoreThanOne = [[neg x, neg y] | row <- atLeastOne, (x:xs) <- tails row, y <- xs]
```

We have talked about `atLeastOne` and `noMoreThanOne` before and mentioned `prefilled`.

`enc` puts all the sub-CNFs together.

The clauses in `prefilled`, `noMoreThanOne`, `atLeastOne` are respectively unary, binary, nine-ary. It is better to put the smaller clauses at the beginning. Why?

`solve` the problem!

Click here to access all the files and play with them interactively online.[48]

[48] https://repl.it/@haoranpeng/Introduction-to-Computation

Put everything together in another file:

```
import Encode
import DPLL
import Grid
import Data.List (sort)

solve :: Grid -> IO ()
solve g = mapM_ print $ format $ solve' g

solve' :: Grid -> Valuation (Int, Int, Int)
solve' g = sort [n | n <- dpll (enc g) [], isPos n]
           where
            isPos (P _) = True
            isPos (N _) = False

format :: Valuation (Int, Int, Int) -> Grid
format [] = []
format v  = let
                (l, r) = splitAt 9 v
            in
                map (getNum) l : format r
            where
                getNum (P (_,_,x)) = x
```

These functions are for pretty printing, don't worry if you don't understand them. To solve the example Sudoku:

```
GHCi> :l Sudoku
[1 of 4] Compiling Grid             ( Grid.hs, interpreted )
[2 of 4] Compiling DPLL             ( DPLL.hs, interpreted )
[3 of 4] Compiling Encode           ( Encode.hs, interpreted )
[4 of 4] Compiling Main             ( Sudoku.hs, interpreted )
Ok, modules loaded: DPLL, Encode, Grid, Main.
GHCi> solve exampleGrid
[5,3,4,6,7,8,9,1,2]
[6,7,2,1,9,5,3,4,8]
[1,9,8,3,4,2,5,6,7]
[8,5,9,7,6,1,4,2,3]
[4,2,6,8,5,3,7,9,1]
[7,1,3,9,2,4,8,5,6]
[9,6,1,5,3,7,2,8,4]
[2,8,7,4,1,9,6,3,5]
[3,4,5,2,8,6,1,7,9]
```

For more advanced users, add the following line to Sudoku.hs

```
main = solve exampleGrid
```

and compile using:

```
$ ghc -o Sudoku Sudoku.hs
```

And run using:

```
$ ./Sudoku
```

How big is the problem?

Basic counting tells us that there are $9 \times 9 \times 9 = 729$ variables.

Also, there are 11988 clauses in total (not counting `prefilled`), of which 11664 are binary and 324 are nine-ary.

```
GHCi> length (noMoreThanOne ++ atLeastOne)
11988
GHCi> length noMoreThanOne
11664
GHCi> length atLeastOne
324
```

Regular Expressions

A **reg**ular **ex**pression (regex for short) is a sequence of characters that define a search pattern[49]. Regular expressions are used regularly[50] by us computer people. In its simplest form, a regex can be a string like:

"regex"

We can use it to search for any string that matches with "regex", in a webpage for example:

Regex is though much more powerful and we are going to look at the most essential bits of regex[51] later in this chapter.

We will use regex to search through the text below which is also saved in my home directory as `example.txt`.

```
If music be the food of love, play on;
Give me excess of it, that, surfeiting,
The appetite may sicken, and so die.
That strain again! it had a dying fall:
O, it came o'er my ear like the sweet sound,
That breathes upon a bank of violets,
Stealing and giving odour! Enough; no more:
'Tis not so sweet now as it was before.
O spirit of love! how quick and fresh art thou,
That, notwithstanding thy capacity
Receiveth as the sea, nought enters there,
Of what validity and pitch soe'er,
But falls into abatement and low price,
Even in a minute: so full of shapes is fancy
That it alone is high fantastical.
```

[49] Regular expression, Jun 2019d. URL https://en.wikipedia.org/wiki/Regular_expression

[50] See what I did there?

[51] POSIX Extended Standard

Regular Characters and Metacharacters in Regex

Much of this section is copied from Wikipedia[52].

Each character in a regular expression (that is, each character in the string describing its pattern) is either a metacharacter, having a special meaning, or a regular character that has a literal meaning. For example, in the regex "a.", 'a' is a literal character which matches just 'a', while '.' is a meta character that matches every character except a newline.

[52] Regular expression, Jun 2019d. URL https://en.wikipedia.org/wiki/Regular_expression

We use the below command to match a regex in a file:

```
$ egrep -o "YOUR REGEX HERE" FILENAME
```

If we match "a." in `example.txt`, we get all the two-character strings that starts with an 'a'.

```
$ egrep -o "a." example.txt
ay
...
a
...
a,
...
```

Here are the metacharacters and their descriptions:

Metacharacter	Description
^	Matches the starting position of any line.
.	Matches any character except for newline. Acts as a regular character in bracket expressions [ ].
[ ]	A bracket expression. Matches any of the single characters that are contained within the brackets.
[^]	Matches any single character other than the ones that are contained within the brackets.
( )	Groups characters together.
$	Matches the ending position of any line.
*	Matches the preceding element zero or more times.
{m, n}	Matches the preceding element at least m and not more than n times.
?	Matches the preceding element zero or one time.
+	Matches the preceding element one or more time.
|	Matches either the expression before or the expression after the operator. Similar to a disjunction.

The table is probably too much information to process. Let's have a look at some examples.

- Get all the lines that **contain** an apostrophe.

```
$ egrep "'" example.txt
O, it came o'er my ear like the sweet sound,
'Tis not so sweet now as it was before.
Of what validity and pitch soe'er,
```

We use GNU grep throughout, Mac users might find certain regex not working.

- Get all the lines that **start** with an apostrophe.

```
$ egrep "^'" example.txt
'Tis not so sweet now as it was before.
```

- Get all the seven-character **strings** that start with 'v' or 'p' and do not end with a vowel.

```
$ egrep -o "[vp][a-z][a-z][a-z][a-z][a-z][^aeiou]" example.txt
petite
violets
validit

$ egrep -o "(v|p)[a-z]{5}[^aeiou]" example.txt
petite
violets
validit
```

The `-o` flag tells `egrep` to only print the part of the line that matches the pattern, instead of the whole line.

- Get all the **words** (in lower case) that contain at least one 'b'.

```
$ egrep -o "[a-z]*b[a-z]*" example.txt
be
breathes
bank
before
abatement
```

- Get all the lines that contain "love".

```
$ egrep -o ".*love.*" example.txt
If music be the food of love, play on;
O spirit of love! how quick and fresh art thou,
```

The examples above are just the tip of an iceberg of what regex can do but an understanding of just those examples will certainly go a long way.

This chapter is designed to be a practical introduction to regex so that you can comfortably use `egrep` and regex in the terminal (and possibly in programming languages). The kind of regex we will see in future chapters will be less complicated.

Useful Regex Resources

There are a lot of good regex resources online[53]. If you want to know more about `grep` and regex, below are some of them:

Where GREP Came From - Computerphile
Regular Expressions (Regex) Tutorial by Corey Schafer
Regex Quickstart Guide
POSIX Section of Regex Wikipedia Page

[53] One of the reasons why this chapter is quite brief.

Regex and Machines

Illustrations of machines are created using FSM Workbench made by former Edinburgh University student Matthew Hepburn.

Matthew Hepburn. URL https://hepburn.codes/fsm-workbench/create.html

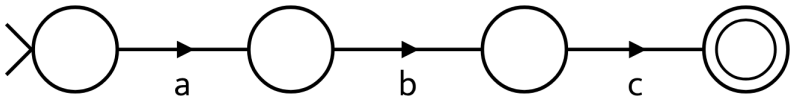
Below are the metacharacters we will use in subsequent chapters:

Metacharacter	Name	Description
*	Kleene star	Matches the preceding element zero or more time.
|	The pipe	Matches either the expression before or after the operator. You can interpret it as 'or'
( )	Round brackets	For grouping characters together.

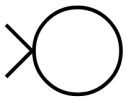
This minimal set of metacharacters doesn't look very expressive but as you will soon recognize, it achieves basically everything the previous large set of metacharacters can achieve.

Regex is great and all but now the question for your curious minds is that how does `grep` figure out what to return when we feed it a regex such as "(a|b)a*(b|c)"?

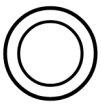
`grep` **converts the regex into an equivalent machine then use the machine to match strings.** Starting with the basics, below is the transition diagram of a machine that matches the same strings as the regex "abc":



Each circle represents a **state** that the machine can be in and each arrowed line represent a **transition**. You might have noticed that some circles are different: circles that look like the one below are **start states**:

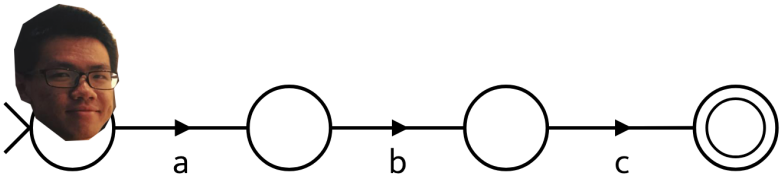


and circles that look like this one are **accept states**:

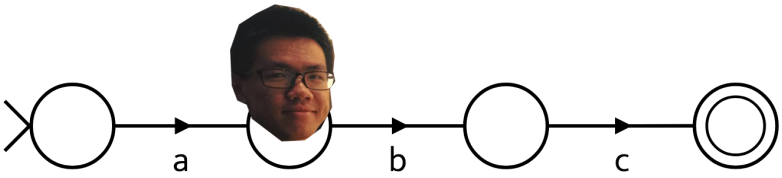


Let's test if "aba" matches with our regex "abc"[54].

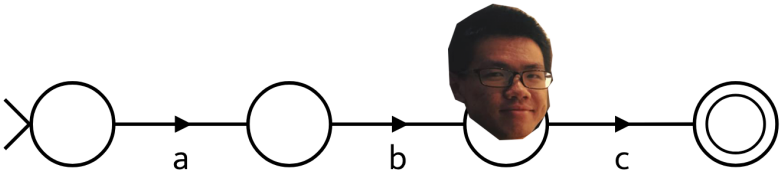
Initially, our machine is in its start state (the position of my face corresponds to our machine's state):



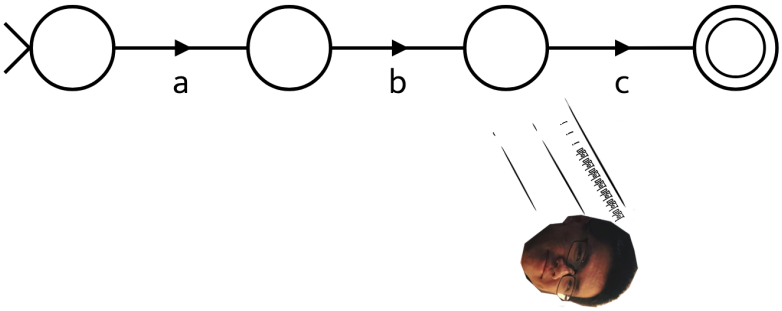
We input the characters one by one starting with "**a**":



Then input "**b**":

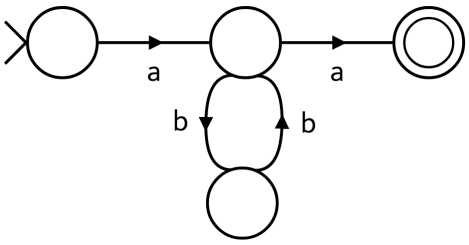


Finally input "**a**". Observe that there's no transitions for "a":



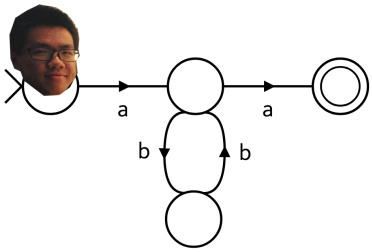
We did not arrive at an **accept state** after inputting all the characters which means "aba" does not match with the regex "abc".

Let's look at a slightly more involved regex "a(bb)*a". Below is the corresponding machine's transition diagram:

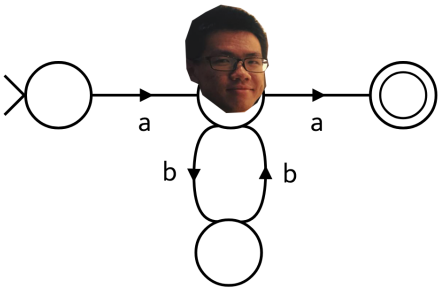


[54] Obviously it doesn't but we want to see how our machine rejects it.

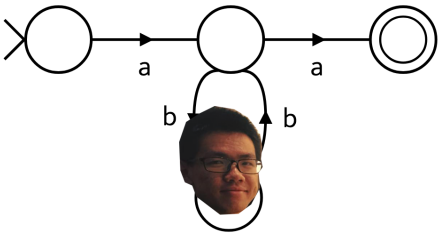
The regex matches all the strings that **start and end with an "a" with possibly an even number of "b"s in between**. The machine should accept "abba", let's try it out. The machine is initially in its **start state**:



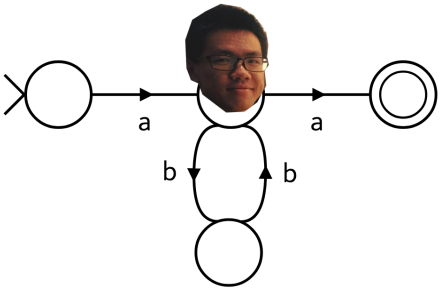
Input "**a**":



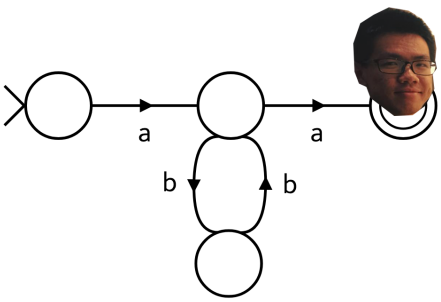
Input "**b**":



Input the second "**b**":



Finally input "**a**":



This time, we did arrive at an **accept state** which means that "abba" matches with the regex "a(bb)*a".

Deterministic Finite Automata

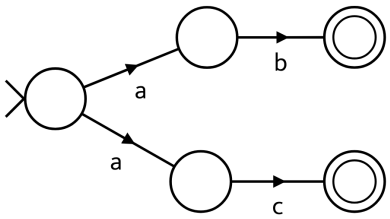
The machines we saw in the previous chapter have many names: **D**eterministic **F**inite **A**utomata, Deterministic Finite State Machines, Deterministic Finite State Automata etc. They all refer to the same thing and we will mainly be using the abbreviated name – DFA.

DFA has two important properties:

Reminder: When we drew machines in the previous chapter, each circle represents a state and each arrowed-line represents a transition.

1. It has **finitely** many states (we don't concern ourselves too much with this property).
2. Its state transitions must be **deterministic**. This means that given the state the machine is in, and an input symbol, there is one and only one state the machine can transition to. Below is a machine that is **NOT** deterministic[55] because from its start state, inputting "a" can lead to two different states:

[55] We will study nondeterministic machines in-depth later.



DFA Formal Specification

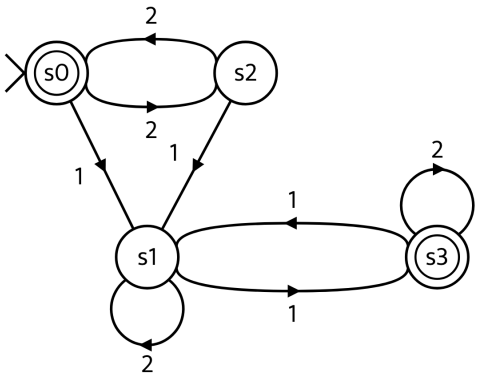
We discussed informally about how DFA works in the previous chapter and we have omitted some details that aren't important for our intuition.

In this section, we are going to formally specify a DFA so that we can easily implement a DFA in Haskell. To formally specify a DFA we have to specify its five parts. Mathematically, a DFA is a 5-tuple, $(Q, \Sigma, \delta, s, F)$, consisting of:[56]

[56] Deterministic finite automaton, Apr 2019a. URL https://en.wikipedia.org/wiki/Deterministic_finite_automaton

- a finite set of states Q
- a finite set of input symbols called the alphabet Σ
- a transition function $\delta : Q \times \Sigma \to Q$
- a start state $s \in Q$
- a set of accept states $F \subseteq Q$

Now let's make sure that we understand what each of the parts are using this machine:



To specify this machine, the five parts needing specifying are as follows:

- the set of states $Q = \{s0, s1, s2, s3\}$
- the set of input symbols $\Sigma = \{1, 2\}$
- a transition function δ where:

$$\begin{aligned} \delta(s0, 1) &= s1 \quad \delta(s0, 2) = s2 \\ \delta(s1, 1) &= s3 \quad \delta(s1, 2) = s1 \\ \delta(s2, 1) &= s1 \quad \delta(s2, 2) = s0 \\ \delta(s3, 1) &= s1 \quad \delta(s3, 2) = s3 \end{aligned}$$

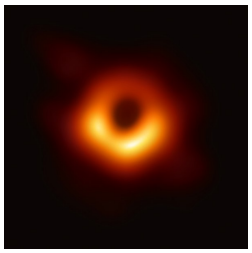
Take some time to understand the correspondence between the drawing and the formal specification. The amount of Greek symbols might be somewhat overwhelming.

- the start state $s = s0$
- the set of accept states $F = \{s0, s3\}$

Note that a DFA can have **only one start state but multiple accept states**[57].

[57] Why? The answer relates to the deterministic property of DFAs.

Black Hole Convention

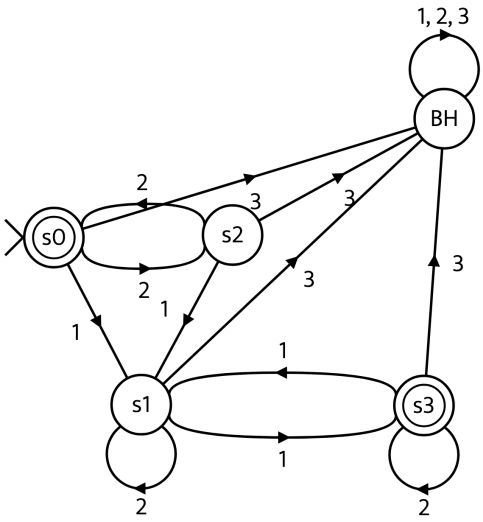


Take the DFA on this page and modify the set of input symbols Σ to be:

$$\Sigma = \{1, 2, 3\}$$

Now say if we input a 3 when the machine is in state $s0$ (or in any other states), what transition will the machine take? For a machine to be deterministic, every legal input[58] should lead to exactly one state transition.

We know that our DFA would never accept a string that contains a 3. In other words, input 3 should immediately be rejected. From this observation, we can make the machine correct by adding one more state – a **non-accepting** state where the machine can't leave once it is there (it behaves like a **B**lack **H**ole):



Black hole states make the diagram a lot **messier** and cumbersome to draw, so we are going to be lazy and **NOT**[59] draw them in the future. Just remember that not drawing them doesn't mean they are not there.

[58] That is, the input symbol is in the alphabet.

New specification:

$Q = \{s0, s1, s2, s3, BH\}$

$\Sigma = \{1, 2, 3\}$

$\delta(s0,1) = s1 \quad \delta(s0,2) = s2 \quad \delta(s0,3) = BH$

$\delta(s1,1) = s3 \quad \delta(s1,2) = s1 \quad \delta(s1,3) = BH$

$\delta(s2,1) = s1 \quad \delta(s2,2) = s0 \quad \delta(s2,3) = BH$

$\delta(s3,1) = s1 \quad \delta(s3,2) = s3 \quad \delta(s3,3) = BH$

$\delta(BH,1) = BH \quad \delta(BH,2) = BH \quad \delta(BH,3) = BH$

start state $s = s0$

$F = \{s0, s3\}$

[59] In the exam, you can still choose to draw them though.

DFA in Haskell

First make some type synonyms for DFA[60]:

[60] Let Haskell know what a DFA is / consists of.

```
type State = String
type Symbol = Char
type DFA = (
    [State],  -- Q
    [Symbol], -- Sigma
    State -> Symbol -> State, -- delta
    State,    -- s
    [State]   -- F
    )
```

Encode the DFA on the previous page in Haskell:

```
exampleDFA :: DFA
exampleDFA = (
    ["s0", "s1", "s2", "s3", "BH"],
    ['1', '2', '3'],
    delta,
    "s0",
    ["s0", "s3"]
    ) where
        delta "s0" '1' = "s1"
        delta "s0" '2' = "s2"
        delta "s1" '1' = "s3"
        delta "s1" '2' = "s1"
        delta "s2" '1' = "s1"
        delta "s2" '2' = "s0"
        delta "s3" '1' = "s1"
        delta "s3" '2' = "s3"
        delta _    _   = "BH"
```

For comparison, here is the mathematical specification:

$Q = \{s0, s1, s2, s3, BH\}$
$\Sigma = \{1, 2, 3\}$
$\delta(s0,1) = s1 \quad \delta(s0,2) = s2 \quad \delta(s0,3) = BH$
$\delta(s1,1) = s3 \quad \delta(s1,2) = s1 \quad \delta(s1,3) = BH$
$\delta(s2,1) = s1 \quad \delta(s2,2) = s0 \quad \delta(s2,3) = BH$
$\delta(s3,1) = s1 \quad \delta(s3,2) = s3 \quad \delta(s3,3) = BH$
$\delta(BH,1) = BH \quad \delta(BH,2) = BH \quad \delta(BH,3) = BH$
start state $s = s0$
$F = \{s0, s3\}$

We can then write functions to make `exampleDFA` accept / reject strings. Acceptances / rejections are dependent only on the final state the machine is in after consuming all the input symbols so we need a function that outputs the final state:

```
finalState :: DFA -> State -> [Symbol] -> State

-- If input symbols are exhausted, output the input state
finalState _ currentState [] =  currentState

-- Use a symbol and transition once, do this recursively until inputs are all used
finalState dfa@(_,_,delta,_,_) currentState (symbol:symbols) =
    finalState dfa (delta currentState symbol) symbols
```

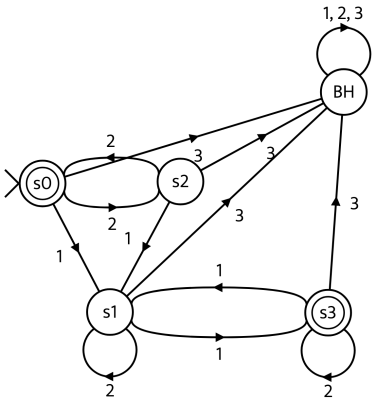
Confused about @? See Haskell/Pattern matching

For the function that accepts / rejects strings, we just need to use `finalState` to get the final state, output `True` if it is an accept state and vice versa.

```
accept :: DFA -> [Symbol] -> Bool
accept dfa@(_,_,_,startState,acceptStates) symbols =
    finalState dfa startState symbols `elem` acceptStates
```

Test the code out, use the transition diagram in the margin to verify:

```
GHCi> accept exampleDFA "221212"
True
GHCi> accept exampleDFA "3"
False
GHCi> accept exampleDFA "2212"
False
```

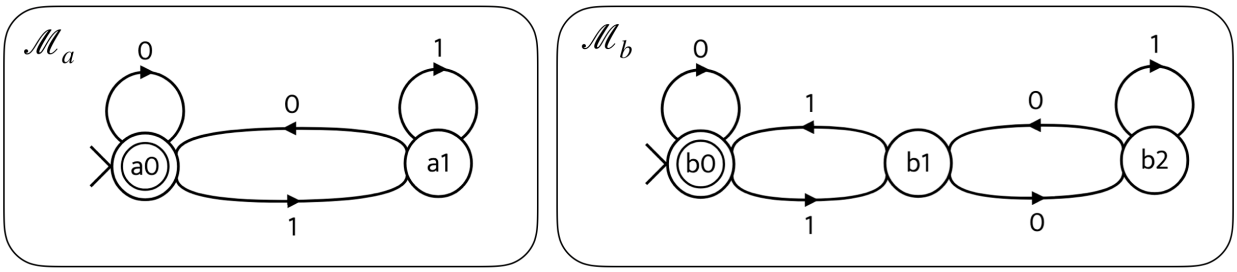


Product DFA and Sum DFA

Given a deterministic finite state machine $\mathcal{M}$, we call the set of strings accepted by $\mathcal{M}$ **the language accepted by** $\mathcal{M}$, denoted as:

$$L(\mathcal{M})$$

Now suppose we are given two DFAs – $\mathcal{M}_a$ and $\mathcal{M}_b$:



The languages accepted by $\mathcal{M}_a$ and $\mathcal{M}_b$ are respectively:

$$L(\mathcal{M}_a) = \text{set of binary numbers that are divisible by 2}$$
$$L(\mathcal{M}_b) = \text{set of binary numbers that are divisible by 3}$$

Don't see why? Hint: think about modular arithmetic. What happens to the current value after each transition.
$110_2 = 6_{10}$
$1100_2 = 12_{10}$
$1101_2 = 13_{10}$

Product Construction

Often, we would like to construct a DFA which accepts a language that is an intersection of two other languages. For example, we want a machine $\mathcal{M}_{a\times b}$ where:

We are only talking about regular languages here.

$$\begin{aligned} L(\mathcal{M}_{a\times b}) &= L(\mathcal{M}_a) \cap L(\mathcal{M}_b) \\ &= \text{set of binary numbers that are divisible by both 2 and 3} \end{aligned}$$

Intuitively, we can simulate $\mathcal{M}_{a\times b}$ by running both $\mathcal{M}_a$ and $\mathcal{M}_b$ in parallel. If both machines arrive at accept states for an input string, that string is also accepted by $\mathcal{M}_{a\times b}$.

For example, we know that $110_2 = 6_{10}$ is divisible by 2 and 3 and it should be accepted by $\mathcal{M}_{a\times b}$. First input 1 to $\mathcal{M}_a$ and $\mathcal{M}_b$, $\mathcal{M}_a$ transitions to state $a1$ and $\mathcal{M}_b$ transitions to state $b1$. Then input another 1, $\mathcal{M}_a$ transitions to state $a1$ and $\mathcal{M}_b$ transitions to state $b0$. Finally input 0, $\mathcal{M}_a$ transitions to state $a0$ and $\mathcal{M}_b$ transitions to state $b0$. Since both $a0$ and $b0$ are accept states, 110 is accepted by $\mathcal{M}_a$ and $\mathcal{M}_b$ which means that it is also accepted by $\mathcal{M}_{a\times b}$.

We call $\mathcal{M}_{a\times b}$ a **product** of $\mathcal{M}_a$ and $\mathcal{M}_b$. Before we bring in the formalism, we shall have a look at how we can construct a product DFA with a **transition table**. We label the first column with the states and the first row with the input symbols. Label only the start state[61] for now:

[61] Remember that we are sort of running two machines in parallel. The start state of the product DFA should be a combination of the two original start states.

	0	1
$(a0, b0)$		

Input 0 causes $\mathcal{M}_a$ to transition from $a0$ to $a0$; input 0 causes $\mathcal{M}_b$ transition from $b0$ to $b0$:

	0	1
$(a0, b0)$	$(a0, b0)$	

Input 1 causes $\mathcal{M}_a$ transition from $a0$ to $a1$; input 1 causes $\mathcal{M}_b$ transition from $b0$ to $b1$:

	0	1
$(a0, b0)$	$(a0, b0)$	$(a1, b1)$

Now copy the newly-found states to the first column:

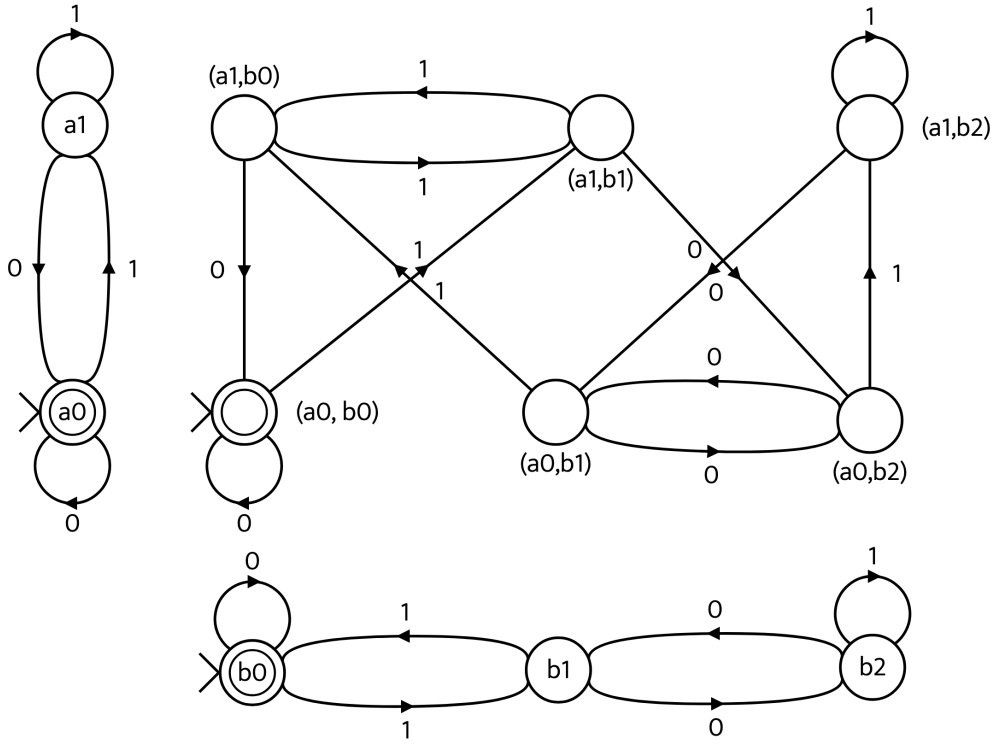
	0	1
$(a0, b0)$	$(a0, b0)$	$(a1, b1)$
$(a1, b1)$		

Continue the process until no new states are found and finally label the accept states[62] (I used bold here but do whatever you want):

[62] The product DFA accepts a string when both its factors accept the string.

	0	1
$(\mathbf{a0}, \mathbf{b0})$	$(a0, b0)$	$(a1, b1)$
$(a1, b1)$	$(a0, b2)$	$(a1, b0)$
$(a0, b2)$	$(a0, b1)$	$(a1, b2)$
$(a1, b0)$	$(a0, b0)$	$(a1, b1)$
$(a0, b1)$	$(a0, b2)$	$(a1, b0)$
$(a1, b2)$	$(a0, b1)$	$(a1, b2)$

With the transition table ready, we can easily draw the $\mathcal{M}_{a\times b}$'s transition diagram as well:



Formal Specification of Product DFA

Suppose we have two DFAs:

The alphabets are identical for convenience. What would you do if they are not?

$$\mathcal{M}_a = (Q_a, \Sigma, \delta_a, q_a, F_a)$$
$$\mathcal{M}_b = (Q_b, \Sigma, \delta_b, q_b, F_b)$$

Let their product DFA be $\mathcal{M}_{a\times b}$:

$$\mathcal{M}_{a\times b} = (Q_{a\times b}, \Sigma, \delta_{a\times b}, q_{a\times b}, F_{a\times b})$$

where

$$Q_{a\times b} = Q_a \times Q_b$$
$$\delta_{a\times b} : Q_{a\times b} \times \Sigma \to Q_{a\times b}$$
$$q_{a\times b} = (q_a, q_b)$$
$$F_{a\times b} = F_a \times F_b$$

The transition table corresponds with the function $\delta_{a\times b}$

$\times$ is the Cartesian product *e.g.*: $A \times B = \{(a, b) \mid a \in A, b \in B\}$

Compare this with the transition table and diagram carefully, these cryptic symbols should become apparent. If you want a small challenge, try and implement product DFA in Haskell.

Sum DFA

A product DFA accepts a language that is the intersection of two other languages. In contrast, a sum DFA accepts a language that is the **union** of two other languages. Constructing a sum DFA is largely the same as constructing a product DFA with one small modification. I will leave this for you to think about.

Hint: You only need to modify the accept states.

Nondeterministic Finite Automata

Shaking things up a little bit, we will start by introducing the formal specification of an NFA (**N**ondeterministic **F**inite **A**utomaton).

Before start reading this chapter, you should have acquired a solid understanding of DFA and its formal specification.

NFA Formal Specification

An NFA is a 5-tuple: $(Q, \Sigma, \Delta, S, F)$, consisting of:[63]

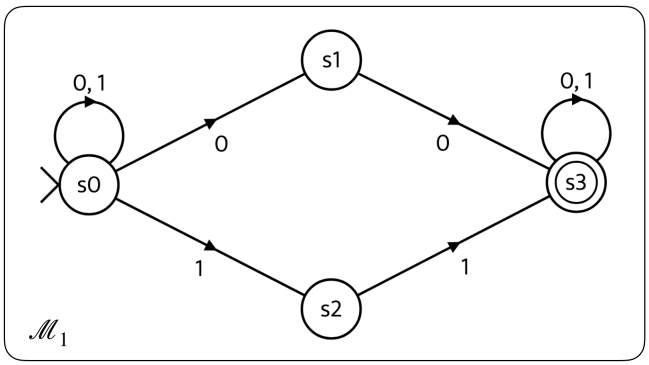
[63] Dexter Kozen. *Automata and Computability*. Springer, 1997

- a finite set of states Q
- a finite set of input symbols called the alphabet Σ
- a transition function $\Delta : Q \times \Sigma \to \wp(Q)$
- a set of start states $S \subseteq Q$
- a set of accept states $F \subseteq Q$

NFA's specification differs from DFA's in two places:

1. NFA has a transition function $\Delta : Q \times \Sigma \to \wp(Q)$

   Recall that $\wp(Q)$ is the powerset of Q. This declares that the output of the transition function is now **a set of states** instead of just a single state. Take NFA $\mathcal{M}_1$ for example:



$Q_1 = \{s0, s1, s2, s3\}$
$\Sigma_1 = \{0, 1\}$
$S_1 = \{s0\}$
$F_1 = \{s3\}$

   Its transition function is:

$$\Delta_1(s0, 0) = \{s0, s1\} \qquad \Delta_1(s0, 1) = \{s0, s2\}$$
$$\Delta_1(s1, 0) = \{s3\} \qquad \Delta_1(s1, 1) = \emptyset$$
$$\Delta_1(s2, 0) = \emptyset \qquad \Delta_1(s2, 1) = \{s3\}$$
$$\Delta_1(s3, 0) = \{s3\} \qquad \Delta_1(s3, 1) = \{s3\}$$

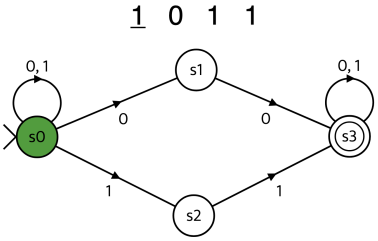
You might have asked: **What does it mean for an NFA to transition from a state to a set of states?**

We can gain an **intuition**[64] by playing with $\mathcal{M}_1$. First establish that $\mathcal{M}_1$ accepts all the binary numbers that contain two consecutive identical numbers *i.e.*

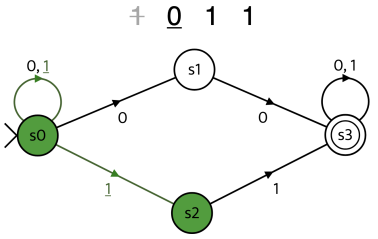
$$L(\mathcal{M}_1) = L((0|1) * (00|11)(0|1)*)$$

Let's test the binary number 1011 which should be accepted.

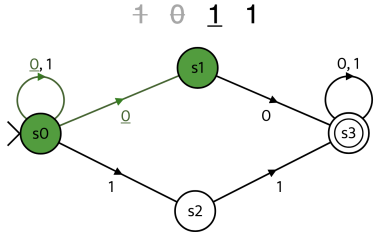
$\mathcal{M}_1$ is initially in state $s0$:



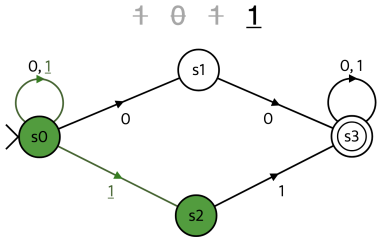
Inputting a 1 causes $\mathcal{M}_1$ to transition to $s0$ and $s2$. $\mathcal{M}_1$ is in two states simultaneously.



Input the next symbol, *i.e.* 0, to $\mathcal{M}_1$. From state $s0$, $\mathcal{M}_1$ transitions to $s0$ and $s1$. From state $s2$, $\mathcal{M}_1$ can't transition to anywhere.

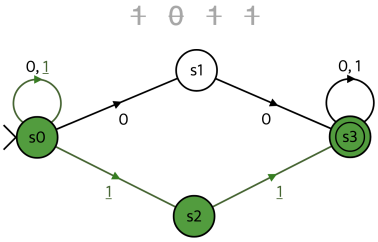


Input the next symbol again. From state $s0$, $\mathcal{M}_1$ transitions to $s0$ and $s2$. From state $s1$, $\mathcal{M}_1$ can't transition to anywhere.



Input the final symbol. From state $s0$, $\mathcal{M}_1$ transitions to $s0$ and $s2$. From state $s2$, $\mathcal{M}_1$ transitions to $s3$.

Finally, the machine is in three states simultaneously. One of them is an accept state, *i.e.* $s3$, thus 1011 is accepted. **Note that an input string is accepted if and only if there is at least one accept state in the set of final states.**



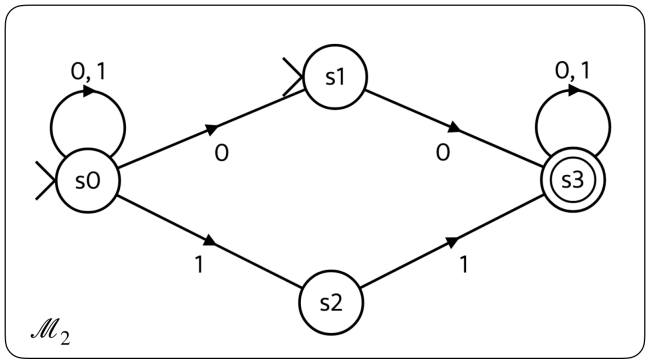
[64] In other words, not formally correct. The intuition I'm giving here is what I call the **many-worlds interpretation of NFA**. In contrast, the commonly given intuition in other textbooks – **guess and verify** – is more like the Copenhagen interpretation.

2. NFA has a **set of start states** $S \subseteq Q$

The NFA $\mathcal{M}_1$ is not a very good example to demonstrate this property as it only has one start state:

$$S_1 = \{s0\}$$

But a NFA can have multiple start states like this:



$\mathcal{M}_2$'s specification is identical to that of $\mathcal{M}_1$ except:

$$S_2 = \{s0, s1\}$$

$\mathcal{M}_2$ in turn accepts all the binary numbers with two consecutive identical numbers and all binary numbers that start with a 0 (assuming that leading zeroes are not ignored).

ϵ-Transitions

ϵ epsilon /ˈɛp.sɪˌlɒn/ represents an **empty string** *e.g.* (let ++ be the concatenation operator):

ϵ is the identity for the string monoid.

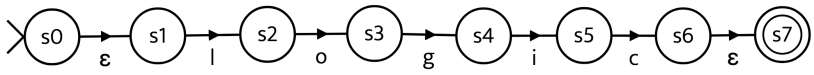
$$\epsilon \text{ ++ "logic"} = \text{"logic" ++ } \epsilon = \text{"logic"}$$

Hmmm, ϵ seems like a pretty useless thing? We will learn how it functions in this section. In future sections, we shall see how it brings convenience and more flexibility to NFAs.

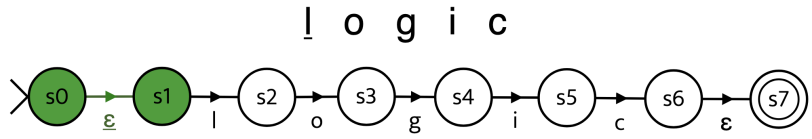
ϵ is treated just like other input symbol in the alphabet *e.g.*:[65]

[65] Formally, the transition function should be modified so that:

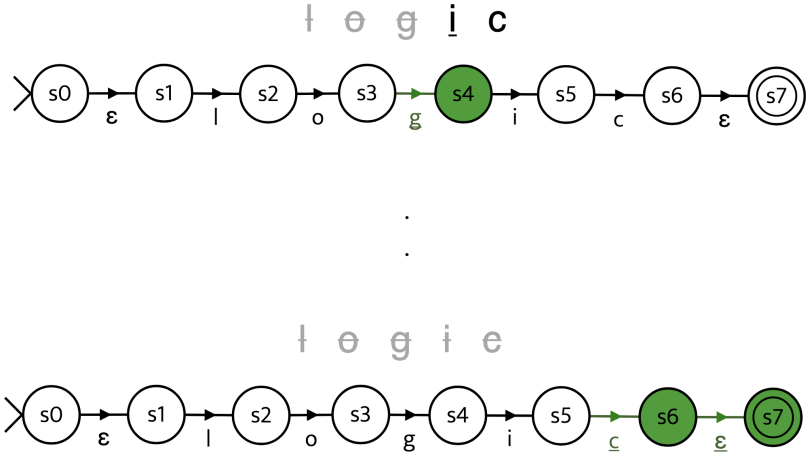
$$\Delta : Q \times (\Sigma \cup \{\epsilon\}) \to \wp(Q)$$



ϵ – transitions are spontaneous meaning that if you are in state $s0$, the machine can transition to $s1$ itself without consuming any input (it can be viewed that it is "**in two states simultaneously**"). Thus it accepts the string "logic":



.
.



.
.

Nondeterministism

At this point, you should have sensed the differences between DFAs and NFAs. NFA's nondeterminism comes from the fact that there may be multiple transitions given an input symbol.

There are two points I want you to think about:

1. Although we can't deterministically predict the transitions an NFA takes, the final result is deterministic *i.e.* either an acceptance or a rejection depending only on your input.

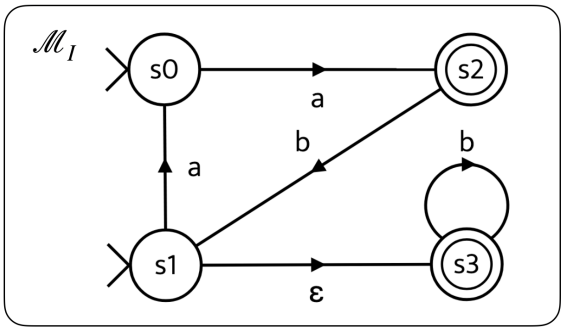
2. If we can tolerate an NFA to be in multiple states simultaneously, we can actually deterministically predict what **states** it will be in next (like the NFA intuition I gave before).

Equivalence to DFA

Every NFA can be converted to an equivalent DFA. The subset construction is the standard algorithm we use for doing such conversions.

Subset Construction

This algorithm constructs an equivalent DFA from an NFA. The procedure and concept are nearly identical to the one we used in constructing product DFAs. Here is an example:



First figure out the set of start states, $s0$ and $s1$ are the obvious ones but also notice there is an ϵ – transition from $s1$ thus $s3$ is "basically a start state" as well. Put the states and the input symbols into a table like so:

	a	b
$\{s0, s1, s3\}$		

To reduce clutter, we use "State $\xrightarrow{\text{symbol}}$ {State}" to represent transitions.

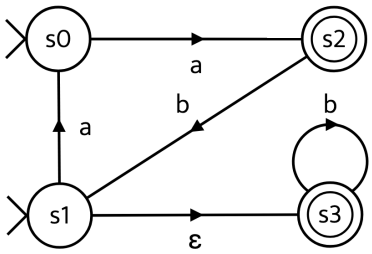
Now input an "a" and record all the transitions:

$$s0 \xrightarrow{a} \{s2\}$$
$$s1 \xrightarrow{a} \{s0\}$$
$$s3 \xrightarrow{a} \{\}$$
$$\{s2\} \cup \{s0\} \cup \{\} = \{s0, s2\}$$

Fill the table with the union of the outputs:

	a	b
$\{s0, s1, s3\}$	$\{s0, s2\}$	

Do the same for input "b":



$$s0 \xrightarrow{b} \{\}$$
$$s1 \xrightarrow{b} \{s3\}$$
$$s3 \xrightarrow{b} \{s3\}$$
$$\{\} \cup \{s3\} \cup \{s3\} = \{s3\}$$

	a	b
$\{s0, s1, s3\}$	$\{s0, s2\}$	$\{s3\}$

Copy those newly-found superstates[66] into the state column:

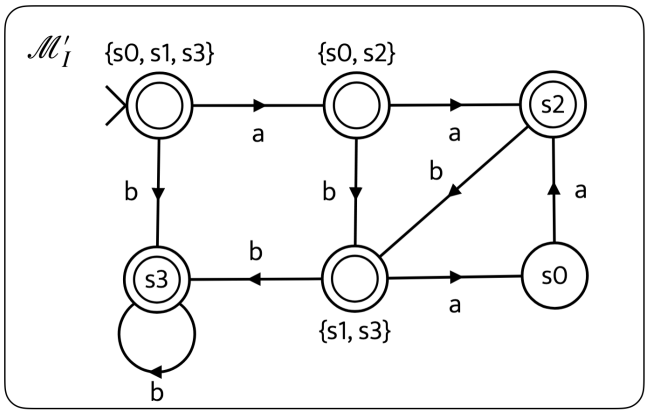
[66] sets of states

	a	b
$\{s0, s1, s3\}$	$\{s0, s2\}$	$\{s3\}$
$\{s0, s2\}$		
$\{s3\}$		

Keep doing this until no new superstates appear. Finally mark (I used bold here but do whatever you want) all the **superstates that contain at least one accept state – these are the "accept superstates".**

	a	b
$\{\mathbf{s0}, \mathbf{s1}, \mathbf{s3}\}$	$\{s0, s2\}$	$\{s3\}$
$\{\mathbf{s0}, \mathbf{s2}\}$	$\{s2\}$	$\{s1, s3\}$
$\{\mathbf{s3}\}$	$\{\}$	$\{s3\}$
$\{\mathbf{s2}\}$	$\{\}$	$\{s1, s3\}$
$\{\mathbf{s1}, \mathbf{s3}\}$	$\{s0\}$	$\{s3\}$
$\{s0\}$	$\{s2\}$	$\{\}$

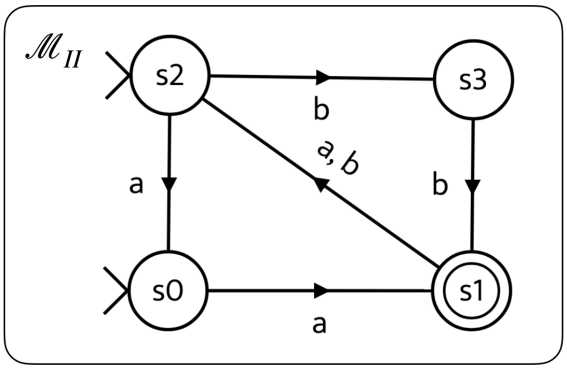
Draw the transition diagram according to the transition table:



Try out some strings and see that it is indeed the case that $L(\mathcal{M}_I) = L(\mathcal{M}'_I)$

I will quickly go through a second example by filling in the transition table in stages[67]:

[67] Roughly how I do subset construction by hand.



Use the margin to carry out the procedure yourself (if you have this page printed) and compare with what I did.

Stage 0:

	a	b
$\{s0, s2\}$		

Stage 1:

	a	b
$\{s0, s2\}$	$\{s0, s1\}$	$\{s3\}$
$\{s0, s1\}$		
$\{s3\}$		

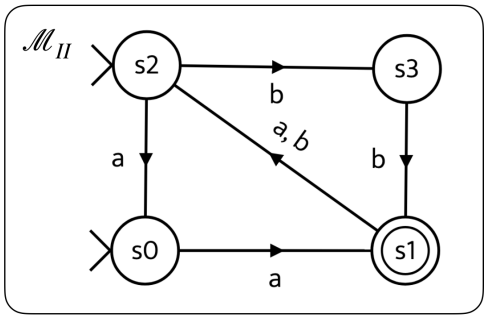
Stage 2:

	a	b
$\{s0, s2\}$	$\{s0, s1\}$	$\{s3\}$
$\{s0, s1\}$	$\{s1, s2\}$	$\{s2\}$
$\{s3\}$	$\{\}$	$\{s1\}$
$\{s1, s2\}$		
$\{s2\}$		
$\{s1\}$		

Stage 3:

	a	b
$\{s0, s2\}$	$\{s0, s1\}$	$\{s3\}$
$\{s0, s1\}$	$\{s1, s2\}$	$\{s2\}$
$\{s3\}$	$\{\}$	$\{s1\}$
$\{s1, s2\}$	$\{s0, s2\}$	$\{s2, s3\}$
$\{s2\}$	$\{s0\}$	$\{s3\}$
$\{s1\}$	$\{s2\}$	$\{s2\}$
$\{s2, s3\}$		
$\{s0\}$		

Stage 4:

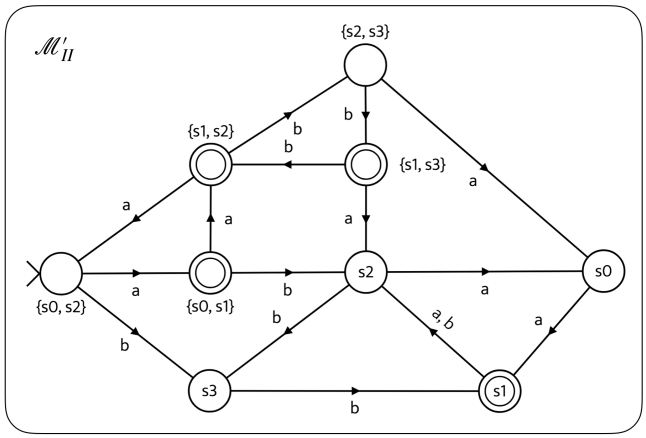


	a	b
$\{s0, s2\}$	$\{s0, s1\}$	$\{s3\}$
$\{s0, s1\}$	$\{s1, s2\}$	$\{s2\}$
$\{s3\}$	$\{\}$	$\{s1\}$
$\{s1, s2\}$	$\{s0, s2\}$	$\{s2, s3\}$
$\{s2\}$	$\{s0\}$	$\{s3\}$
$\{s1\}$	$\{s2\}$	$\{s2\}$
$\{s2, s3\}$	$\{s0\}$	$\{s1, s3\}$
$\{s0\}$	$\{s1\}$	$\{\}$
$\{s1, s3\}$		

Stage 5:

	a	b
$\{s0, s2\}$	$\{s0, s1\}$	$\{s3\}$
$\{\mathbf{s0}, \mathbf{s1}\}$	$\{s1, s2\}$	$\{s2\}$
$\{s3\}$	$\{\}$	$\{s1\}$
$\{\mathbf{s1}, \mathbf{s2}\}$	$\{s0, s2\}$	$\{s2, s3\}$
$\{s2\}$	$\{s0\}$	$\{s3\}$
$\{\mathbf{s1}\}$	$\{s2\}$	$\{s2\}$
$\{s2, s3\}$	$\{s0\}$	$\{s1, s3\}$
$\{s0\}$	$\{s1\}$	$\{\}$
$\{\mathbf{s1}, \mathbf{s3}\}$	$\{s2\}$	$\{s1, s2\}$

Construct the transition diagram accordingly:



Note About Complexity

For both our examples, there are more states in the equivalent DFAs than the NFAs. Most of the time, we'd get more states after conversion and in the worst case we will get 2^n states in the DFA if the NFA has n states[68]. For bigger NFAs, the subset construction may be infeasible.

[68] The complexity is $\Theta(2^n)$

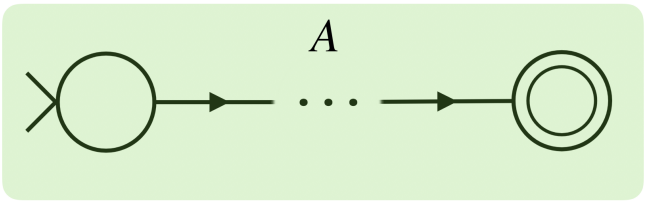
Regex to Machines

Following from the previous chapter where we converted machines to regex, we will convert regex to machines in this chapter. Again, for many regex that we will see in this course, you should be able to construct equivalent machines by observing and thinking. For more complicated regex, you can use the systematic method introduced below.

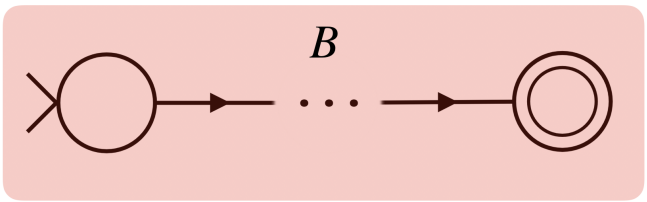
Thompson's Construction

Given two regex and their equivalent machines:

- Regex `A`:

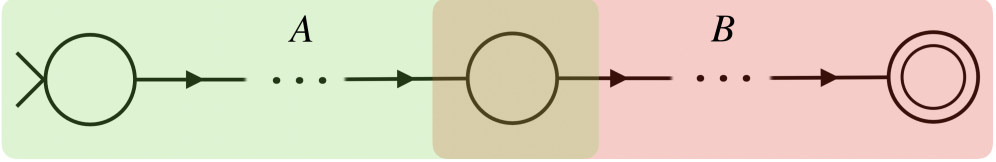


- Regex `B`:

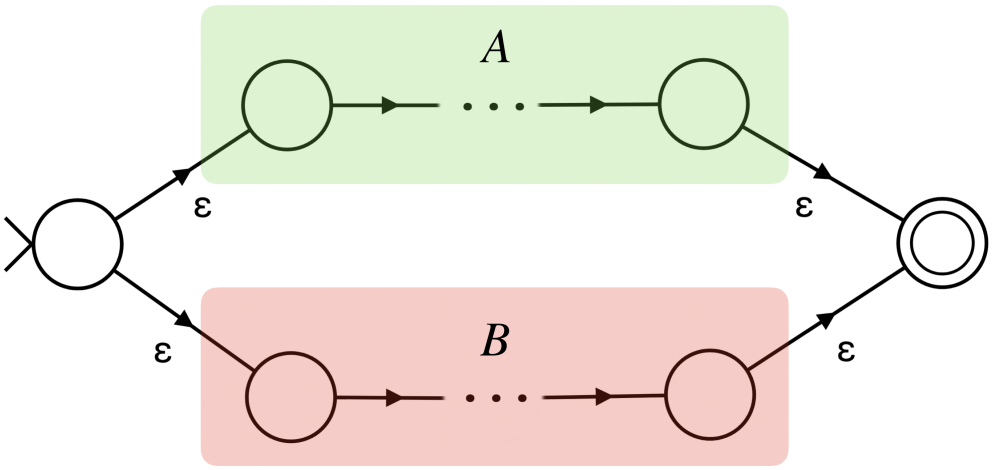


There are three operations we can do to regex to form new regex:

1. Concatenation between two regex *i.e.* `AB`. The equivalent machine is simple to construct:

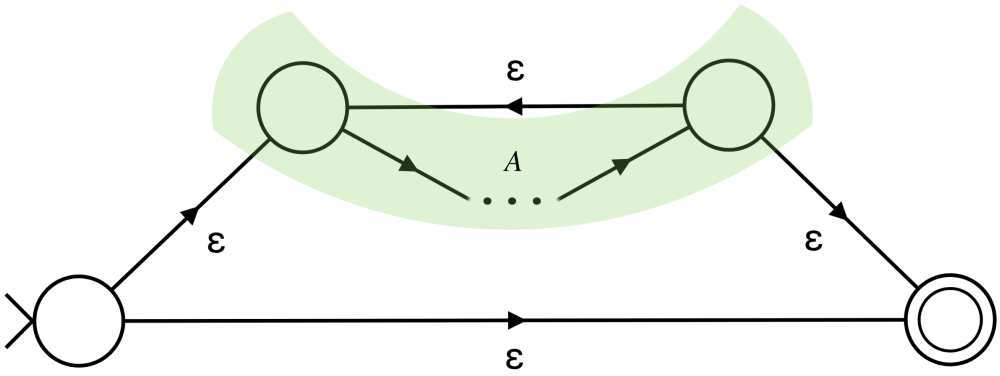


2. Disjunction between two regex *i.e.* `A|B`. We can use ϵ – transitions to build two paths, one for `A` and one for `B`:



3. Kleene star on one regex *i.e.* `A*`. Remember `A*` means the pattern `A` repeated for zero or more times, it is the same as ϵ or repeating `A` at least once[69]:

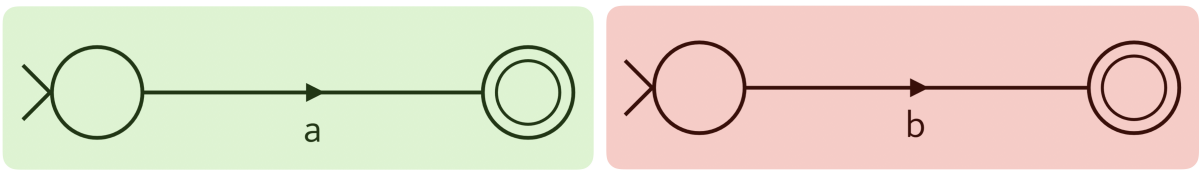
[69] Same as ϵ|`A+` although we don't use `+` in our regex.



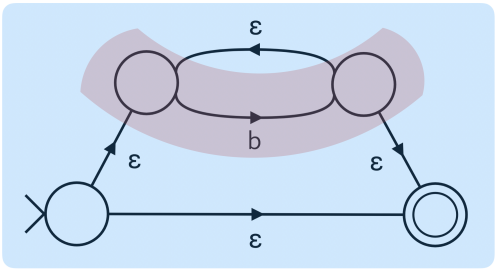
With the above knowledge under our belt, we can construct an equivalent NFA bottom-up from every regex. Let's go through an example.

Example 1 – `ab*a|b*`

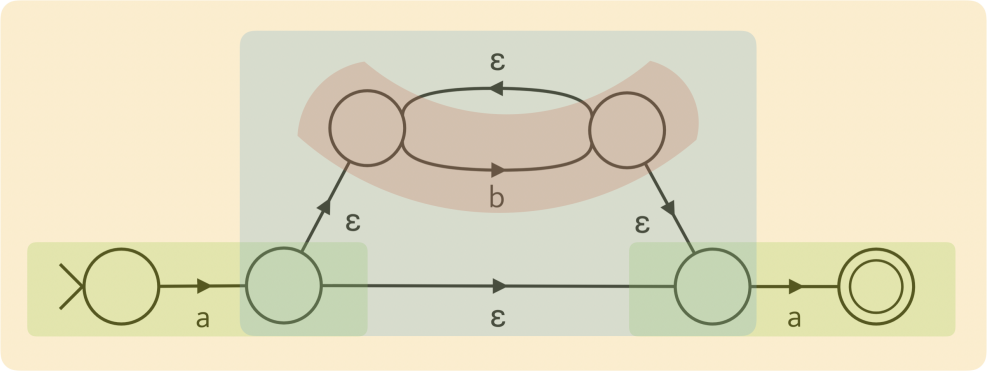
Step 1 – Construct machines for `a` and `b`:



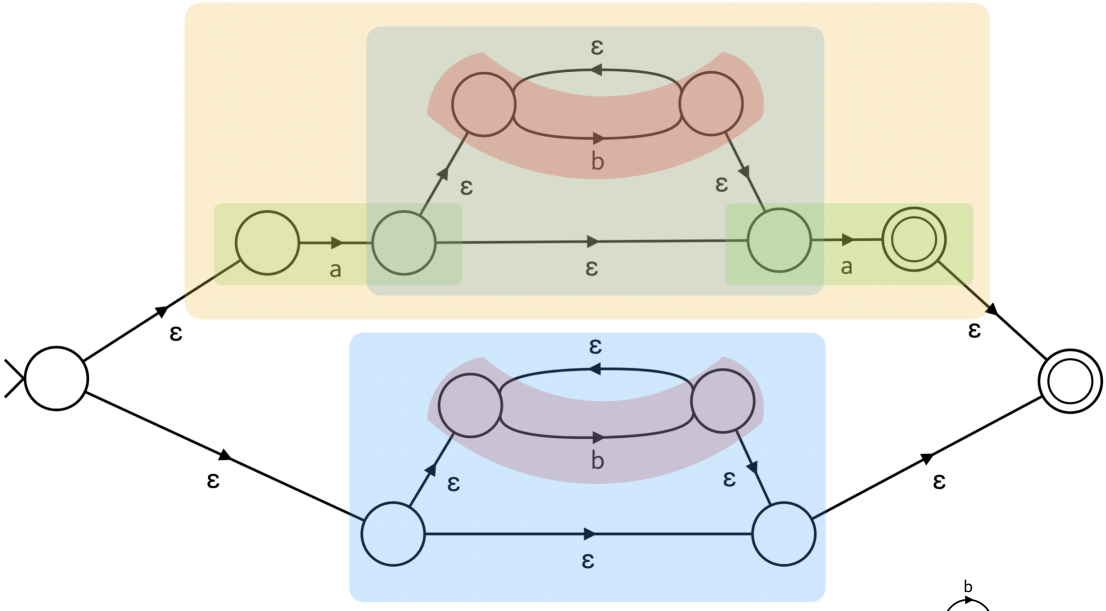
Step 2 – Construct a machine for `b*` based on `b`:



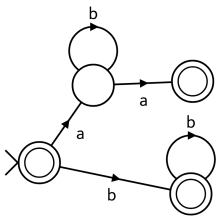
Step 3 – Construct a machine for `ab*a` based on `a` and `b*`:



Step 4 – Finally construct a machine for `ab*a|b*` based on `ab*a` and `b*`:



The NFA we have just constructed is correct but way too complicated for a simple regex like `ab*a|b*`. The machine in the margin is also equivalent to `ab*a|b*` but much much smaller.



Bibliography

Infix operator, Jan 2008. URL https://wiki.haskell.org/Infix_operator.

Deterministic finite automaton, Apr 2019a. URL https://en.wikipedia.org/wiki/Deterministic_finite_automaton.

Dpll algorithm, May 2019b. URL https://en.wikipedia.org/wiki/DPLL_algorithm.

Karnaugh map, May 2019c. URL https://en.wikipedia.org/wiki/Karnaugh_map.

Regular expression, Jun 2019d. URL https://en.wikipedia.org/wiki/Regular_expression.

Barbara Hall-Partee, Alice G. B. ter Meulen, and Robert Eugene Wall. *Mathematical methods in linguistics*. Kluwer Academic, 1990.

Matthew Hepburn. URL https://hepburn.codes/fsm-workbench/create.html.

Dexter Kozen. *Automata and Computability*. Springer, 1997.

Robert Roth Stoll. *Sets, logic, and axiomatic theories*. Freeman and Company, 1974.

Index