# *Notes on Finite State Machines and Regular Languages*

*J. Todd Wilson*

*Andrew Clifton*

*Department of Computer Science*

*California State University, Fresno*

*Fresno, CA 93740*

*January 15, 2016*

`twilson@csufresno.edu`, `andyclifton@csufresno.edu`

As the major concerns of computer science are *computation* and the nature and variety of the *machines* that support it, in all their glorious but unwieldy variety, a particularly productive strategy in the study of computer science is to focus on special cases of computations (and machines) that exhibit two opposing qualities: they are special enough to make a detailed study feasible, yet they are general enough to provide useful insight into many broader questions. In the whole of computer science, there is no better example of a subject that meets these two requirements than the one involving Finite State Machines and Regular Languages.

These notes present the standard definitions, techniques, and results of this subject, although in occasionally new ways. They are currently in draft form and contain parts in various stages of completion, from bare outlines to polished text, so please do not distribute them without permission of the author.

## *Introduction*

Suppose we are given a string of characters, say `twilson@csufresno.edu`, and need to determine whether this string represents a valid email address. Or we are given a string and need to determine whether it represents a valid date, or zip code, or system log entry, or programming-language statement. These are all examples of *string recognition problems*, and these problems form the context of our study.

What do we mean by "valid"? Clearly we must have some definition in mind of what strings are, and hence, are not, part of the class we are interested in. At the most basic level, we need to know what *characters* or, more generally, what *symbols* are allowed in our strings. We refer to the set of all valid symbols as the *alphabet* and write it $\Sigma$. For example, for email addresses $\Sigma$ includes not just alphanumeric characters, but also `.` `@` and any other characters allowed in email addresses.

See RFC ... for all the gruesome details on what characters are permissible in email addresses.

Sometimes we need to refer to the set of *all* strings, of any length $\geq 0$ over an alphabet. We write this as $\Sigma^*$.

A *language* is some subset of $\Sigma^*$ containing only the strings we are interested in. For example, the language $L_{\text{email}}$ contains only the valid email addresses. Formally speaking, a language is a possibly-infinite set of strings, but as we will see, you can also think of a language as a mechanism for either *recognizing* strings in that set, or for *generating* all strings in that set.

Again, suppose we are interested in the language $L_{\text{email}}$. How can we describe what constitutes a valid email address? By looking at the above example `twilson@csufresno.edu`, we can see that an email consists some *user identifier*, follow by an `@`-sign, follow by a hostname. We can formalize this by giving it as a *grammar* $G_{\text{email}}$:

$$\begin{aligned} email &\to user\,identifier@hostname \\ user\,identifier &\to \ldots \\ hostname &\to \ldots \end{aligned}$$

(Obviously, we still need to fill in the definitions of $user\,identifier$ and $hostname$.)

Formally speaking, a grammar is a four-tuple $(V, \Sigma, P, S)$ where $V$ is a finite set of *non-terminals* (e.g., *email*, *hostname*), $\Sigma$ is the alphabet, the set of terminals, $P \subset V \times (V \cup \Sigma)^*$ is a finite set of *rules*, and $S \in V$ is the *start symbol*.

This definition actually describes *context-free* grammars, those in which the left-hand-side of every rule is restricted to a single nonterminal. There are grammars, which we will consider later, in which this restriction does not hold; e.g., $\mathtt{a}B\mathtt{c} \to \mathtt{abc}$ is a valid, but not context-free, rule.

Informally, a grammar is defined by its rules, and the nonterminals and terminals that occur in it (the above definition allows for grammars to have nonterminals and terminals that are never used by the rules, but since these have no effect, we can safely ignore them). Every rule describes the *structure* of a particular nonterminal, as a sequence of nonterminals and/or terminals. Note that it *is* valid for this sequence to be empty! A rule $A \to \varepsilon$ indicates that the definition of $A$ is empty, and is refered to as an *epsilon rule*. Also note that there is no restriction requiring each nonterminal to have a single defining rule. For example, this is a perfectly valid grammar:

$$\begin{aligned} A &\to \mathtt{a} \\ A &\to \mathtt{b} \end{aligned}$$

This expresses the fact that the nonterminal $A$ can recognize or match *either* a literal `a` *or* a literal `b`.

When presenting grammars, we will normally assume that the *first* rule is the start rule.

## *String recognition with a grammar*

Here we informally build up a sketch of an algorithm for doing string recognition using a grammar. Consider, again, the example of an email address. We want to use the grammar $G_{\text{email}}$ to determine whether `"twilson@csufresno.edu"` is a valid email address.

1. We begin with the start rule *email*. Its definition tells us that a valid email address consists of a $user\,identifier$ followed by a literal `@` followed by a *hostname*.

2. So we recursively ask whether *user identifier* matches some prefix of `"twilson@csufresno.edu"`. Presumably (hopefully!) it matches `twilson`, leaving the remainder of the string as `@csufresno.edu`.

3. We now ask whether the literal `@` matches some prefix of `@csufresno.edu`; it does, with the remainder of the string now being `csufresno.edu`.

4. We recursively ask whether *hostname* matches a prefix of `csufresno.edu`. It does, and all that's left is the empty string $\varepsilon$.

5. Since the start rule accepted the string given, and since there's nothing left of the string, the string is accepted as a valid email address.

Note that *both* the final conditions are important: the entire definition of the start rule must have been successful (e.g., if the input string was `twilson|csufresno.edu` the `@` would have failed to match), *and* there must be nothing left of the string when we are finished.

Above we assumed that the definition of a rule could be followed from left to right, but this is not required. It's not hard to construct grammars for which this method will result in an infinite loop:

$$A \rightarrow A\texttt{a}$$
$$A \rightarrow \varepsilon$$

More on derivations...

## *Preliminaries – Haskell Review*

### *Syntax*

Comments in Haskell take two forms:

- Single-line comments (similar to `//`-style comments in C/C++/Java) start with `--` and extend to the end of the line. E.g.,

  ```
  -- This comment extends to the end of the line.
  ```

  Note that in Haskell you can define your own operators, and it is perfectly acceptable to define an operator that starts with `--`, as long as it continues with some other operator-like character. E.g., we could define

  ```
  (-->) :: Int -> Int -> Int
  a --> b   = a*a + b*b
  ```

  Some editors, however, will not be aware of this and may show everything after the start of the `-->` as a comment.

- Multi-line comments (similar to `/* ... */`) start with `{-` and end with `-}`. Note that multi-line comments *can* nest, unlike in C/C++/Java. E.g., this is perfectly valid:

  ```
  {-
     This is commented out
     {- So is this -}
     This is still commented out
  -}
  ```

Identifiers (functions, variables, types, etc.) are subject to a few rules regarding their names:

- Variable and function names must start with a lowercase letter, but may be followed by upper- and lower-case letters, numbers, underscores, or the apostrophe. The latter is commonly used to show that one variable is a slightly different version of another. E.g.,

  ```
  if a == a' then ...
  ```

  The exception to this rule are operator-style functions, which, as shown above, have to start with some kind of symbol character (`-`, `*`, etc.).

- Identifiers starting with uppercase letters are reserved for module names, types, type classes, and type constructors.

  Note that this conflicts with our usual mathematical custom of writing the names of sets in uppercase. Usually we will get around this by writing sets or lists with a plural 's' at the end. E.g., the set A will become `as` ("more than one a"). Bear this in mind when translating math into Haskell; you'll have to do some renaming, so be consistent about it.

- The two type constructors you will probably see most frequently are `True` and `False`, both of type `Bool`. These are, as you might expect, the boolean constants. Note that they must start with uppercase T and F! If you write `true` by accident you will get an `undefined identifier` error.

  Literal values:

- Integer and floating-point values look like you'd expect:

  ```
  1
  105
  0.5
  -12.4
  ```

  But note that there is an unfortunate ambiguity with the unary minus, so often it's safer to write `(-12.4)` with explicit parentheses.

- Character literals are enclosed in single-quotes (forward quotes; backquotes do something different):

  ```
  'a'
  'b'
  ```

  Note that Haskell supports Unicode, so you can use fancy characters if you want.

- String literals are enclosed in double-quotes:

  ```
  "Hello, world!\n"
  ```

  As shown, the usual backslash escapes for special characters are supported. In Haskell, strings are just lists of characters (i.e., of type `[Char]`), so technically a string literal is just another form of a list.

- List literals have two forms, explained in detail below, in the section on lists. Some examples:

  ```
  [1,2,3,4]
  1:2:3:4:[]        -- Same as the previous
  1:(2:(3:(4:[]))) -- Also the same
  1:2:[3,4]         -- Still the same
  ```

- Tuples consists of multiple values, of possibly different types, in parentheses. Tuples are explained more fully below, but here are some examples:

  ```
  (1,"hello")                 -- A "pair" of type (Int, [Char])
  (3.14,[True,False],"potato") -- A "triple" of type (Float, [Bool], [Char])
  ```

- Function values are explained in detail below, but they begin with a backslash, followed by arguments, followed by `->`, followed by the body:

  ```
  (\x -> x)   -- The identity function
  (\v -> v+1) -- The successor function
  (\x y -> x + y) -- This is the same as the function (+)
  ```

Any function can be treated as an infix operator, and any infix operator can be treated as a function, as you find it convenient:

- If f is a function of two arguments then

  ```
  f a b
  ```

  is exactly the same as writing

```
a `f` b
```

- If + is any infix operator, then

  ```
  a + b
  ```

  is exactly the same as writing

  ```
  (+) a b
  ```

  This is mostly useful in situations not where you are calling `(+)` as a function, but where you are storing it in a variable, or passing it as an argument (see the section on "First Class Functions" below for examples).

Layout: Indentation is *significant* in Haskell, meaning that it affects the meaning of your code. Generally speaking, if a line is indented more than the previous line, then it is treated as being part of a new block (this is similar to the layout rule in Python). The block ends with the next line that is unindented (indented *less* than its parent). An example:

```
f x = x + 1
g y = y * 2
```

This would not work if we wrote it as

```
f x = x + 1
 g y = y * 2
```

because now the definition of `g` appears to be nested inside that of `f`, somehow. This is actually a common problem, where the definitions are separated by enough space to make the extra indentation less noticable. Always check your definitions to make sure they are flush left!

If you find the layout-based structure problematic, Haskell also has an explicit block syntax that should be more familiar:

```
let {
x = 1;
y = 2;
} in
  x + y
```

(Note that the expression following the `in` must still be indented, or you could put it on the same line as the `in`.)

The only places where layout really matters are:

- After `let` but before `in`
- After `of` (i.e., after `case ... of`)

- After `where`

Thus, these are the only situations where the block syntax will really help. Some additional examples of block syntax:

```
case x of {
1 -> "Hello";
2 -> "Goodbye";
}

f x = x + y + z
  where {
y = 4;
z = 2;
  }
```

(The crazy indentation is just to show that it doesn't matter in a block; in reality you should try to make your code readable.)

Note that you can use a semicolon anywhere where a newline would normally occur. E.g., you can put multiple definitions on a single line if you like:

```
x = 1; y = 2; z = 3;
```

## *Haskell File Structure*

A Haskell file typically has the extension .hs. A Haskell file can optionally begin with some module imports, followed by *definitions*. (Note that, as in Java, any module imports must appear before all definitions; you cannot mix and match imports and definitions throughout the file.) A module import looks like this

```
import Data.List
```

Haskell uses a hierarchical module structure: here we are importing the List module, which is nested inside the Data module. An unqualified import will load in *every* definition provided by the file. We can qualify the import if we only want to load in specific definitions:

```
import Data.List (permutations, subsequences)
```

This will only import the two functions `permutations` and `subsequences`.

Definitions in Haskell have the (very) general form of some identifier, optionally some argument pattern(s), a literal `=`, and then the body of the definition. For example,

```
x = 10
f x = x + 2
first_two (x1:x2:xs) = x1 + x2
squared_dist a b = a^2 + b^2
```

A definition can optionally be preceded by a type declaration. This consists of the name of the definition, followed by `::`, followed by a type:

```
x :: Int
x = 10

f :: Integer -> Integer
f x = x + 2

first_two :: [Int] -> Int
first_two (x1:x2:xs) = x1 + x2

squared_dist :: Num a => a -> a -> a
squared_dist a b = a^2 + b^2
```

If the type is omitted, Haskell will figure it out for you. But if you give a type, Haskell will still figure out the type, and then check it against the type you gave. This is a good way of "checking your work"; if you and Haskell disagree about the types, probably something went wrong someplace.

The forms that a definition can take are quite varied:

- A single definition can have multiple *clauses*, each matching a different *pattern*. We've already seen this in recursive list functions: we have a clause for the empty list, and then another clause for the cons. Haskell will try to match the actual argument(s) against the clauses' patterns in the order they are given; the first one to match is used.
- A single clause of a definition can have *guards*. Guards allow a single clause to be split into multiple cases, with the case chosen depending on some boolean conditions. For example, here is a function which clamps the value of its first argument to be >= its second and <= its third:

  ```
  clamp :: Int -> Int -> Int -> Int
  clamp x a b | x <= a    = a
              | x >= b    = b
              | otherwise = x
  ```

  `otherwise` is just a synonym for True, acting as an "else" case. If *none* of the guards succeed, then the entire clause is treated as a failed pattern match.
- A clause can have *local definitions* via `where`:

  ```
  f x = x + x2 - z
      where
          x2 = 2 * x
          z = 12
  ```

These local definitions are full definitions in their own right: they can have types, multiple clauses, guards, even nested `where`s! The only difference is that definitions in a `where` are only visible within the body of their attached definition (e.g., above, you cannot refer to `x2` and `z` anywhere but in the definition of f).

- If Haskell gets to the end of the list of clauses and none of them has matched, then it will throw an "inexhaustive match" error.

### *Patterns*

Patterns are what follow the name of a definition on the left-hand side. Although simple argument patterns like

```
f x y = ...
```

are not too hard to understand (f takes two arguments, named x and y within its body), argument patterns can in fact be quite complex and expressive. (Note that for functions with multiple arguments, each argument gets its own pattern.) The forms of patterns are:

- A variable, e.g., `f x y` as above. A variable matches anything, and will result in the matched value being bound to the name of the variable, within the body of the definition.

- The wildcard variable `_`. `_` matches anything, but does *not* bind any name to the value. You can use this for arguments that you don't care about; e.g., in our length function, we did not use the value of `x` in the cons case, so we could have written it as

  ```
  mylen (_:xs) = 1 + mylen xs
  ```

- A literal value, which must match exactly, and does not bind any names. E.g., in the factorial function, the base case is

  ```
  fact 1 = 1
  ```

  This clause will only match if the actual argument is 1.

- A data constructor. We've already seen one of these, the cons constructor:

  ```
  mylen (x:xs) = ...
  ```

  This will match the non-empty list case, and will also split the list into its head and tail, and bind x to the head and xs to the tail. This can be done with any data constructor, including those for data types you create yourself:

```
data NameOrNumber = Name String | Number Int

isName :: NameOrNumber -> Bool
isName (Name _) = True
isName (Number _) = False
```

Data constructors can also contain nested patterns. As shown above, we can use _ within a constructor. You can even next data constructors within data constructors:

```
addFirstTwo :: [Int] -> Int
addFirstTwo (x1:(x2:_)) = x1 + x2
```

Remember that `[a,b,c]` is just shorthand for `a:b:c:[]` (which in turn is equivalent to `a:(b:(c:[]))`), so you can use patterns of the form `[a,b,c,...]` to match a list of specific length. (And again, the elements of this list pattern could themselves be nested patterns!)

- An "as" pattern. Sometimes you want to break up a data constructor (e.g., extract the head and tail of a list) but *also* have access to the complete original value. An `@` pattern does just this:

  ```
  f l@(x:xs) = ...
  ```

  Here, x and xs will be bound to the head and tail as usual, but l will also be bound to the entire original list.

### *Expressions*

While Haskell files consist of definitions, the *body* of every definition must be an expression. Hence the structure of expressions is very important. (Note that when we talk about "builtin" operators and functions, we are actually refering to the set of operators/functions defined in the *Haskell Prelude*. The Prelude is a special module that is automatically imported by every Haskell file; it is also automatically available in GHCi.)

Operators: Haskell supports the usual collection of arithmetic operators:

```
+ - * / ^
```

These are defined on all "numeric" types (i.e., types implementing the `Num` typeclass; see below for a description of what typeclasses are).

Comparison operators are similarly available:

```
> < >= <= == /=
```

The ordering operators (less-than, etc.) are defined on types implementing `Ord`, while the (in)equality operators are defined on types implementing `Eq`. (Note that all numeric and character types implement both of these; structured types like lists and products also implement `==` and `!=`.) All of these return a `Bool` result.

Two other pseudo-comparison functions are `min` and `max`. These do what you'd expect, returning the minimum/maximum of their two arguments (which must be Ord-erable).

Built-in boolean operators are as you would expect:

```
&& || not
```

Note that `not` is just a normal one-argument function, not a special operator. All of these take `Bool` arguments, and return a `Bool` as well.

Signalling errors: If something does not make any sense whatsoever, you can throw an `error`: `error` is a built-in function of type `String -> a`. Note that it's return type is `a`, completely unspecified. This means that you can use `error` *anywhere*, in any type of expression. As soon as it is evaluated, the `error` will print the String you give it and then abort your program. E.g.,

```
x = 1 + error "Whoops!"
```

Evaluating x will cause the error to be thrown.

The other magical error value is `undefined`. We use this in labs to signal parts of the file which you are supposed to fill in. `undefined` has type `a`, so you can use it anywhere, but like `error`, attempting to evaluate it will abort your program and print an error message.

## *Control Structures*

In a language like C/C++/Java, control structures are procedural in nature: they affect the order in which things happen. In Haskell, control structures are expressions: they return values.

if-then-else:

```
if x == 12 then "Hello" else "Goodbye"
```

The general form is

```
if condition then
  true_expression
else
  false_expression
```

The `condition` must be of type Bool, and both the `true_expression` and the `false_expression` must be of the same type. Note that if-then-else is "lazy": only one of `true_expression` and `false_expression` will be evaluated; the unused branch is not evaluated.

Note that since if-then-else is an expression you can do things like

```
12 + (if odd x then 1 else 2) * y
```

case:

```
case x of
  12 -> "Hello"
  _  -> "Goodbye"
```

(This has exactly the same effect as the example if-then-else above.) `case` is roughly Haskell's equivalent to `switch` in C/C++/Java. It solves the problem of nested if's becoming cumbersome, by allowing multiple branches. All of the patterns (to the left of the `->`) must have the same type, the type of x, and all of the return values (to the right of `->`) must have the same type.

Note that the "conditions" on each branch (12 and _ above) can actually be arbitrary patterns, so you can do something like

```
case l of
  [] -> 0
  (x:_) -> x
```

If you want to write a case on a single line, you'll have to use semicolons to separate the cases:

```
case x of 12 -> "Hello" ; _ -> "Goodbye"
```

As with if-then-else, `case` is an expression and can be used anywhere where a value or expression is needed:

```
1 + (case x of 'A' -> 0 ; 'B' -> 1 ) * z
```

let-in:

`let..in` is Haskell's version of local variable definitions, but with a significant twist. `let` lets you bind some names to some expressions (and do pattern-matching in the process, if you like), and thus is useful for either labeling some values according to their function, or abstracting out repeated calculations for efficiency:

```
let x = huge_calculation in x*x + x
```

Rather than perform the `huge_calculation` three times, we perform it once, call the result x, and then compute `x*x + x` (which would otherwise require *three* evaluations of `huge_calculation`). But again, `let..in` is still an expression, so you can do things like

```
x * (let x' = x + 12 in x*y) + z
```

`let..in` can bind more than one name:

```
let x = 12
    y = length "Potato"
    z = [1..]
in
    x + y + head z
```

Later names can refer to earlier ones:

```
let x = 12
    y = x + 1
    z = y ^ 2
in
    x + y + z
```

You can even use `let..in` to bind *functions* locally:

```
let f x = x^2
in
    f 5
```

## *Lists*

Lists are so useful in Haskell that they have a number of different forms. The "cons" form of a list looks like this:

```
1:2:3:4:[]
```

Note that because the : operator associates to the *right*, this is equivalent to

```
1:(2:(3:(4:[])))
```

If you want to put a single element on the front of a list, you can "cons" it on:

```
1 : [2,3,4]
```

(try typing this into GHCi!) evaluates to

```
[1,2,3,4]
```

If you want to treat a list like a stack, then this is your "push" operation.

A lot of times we want to construct a list from a range of values. For example, `[1,2,3,4]` is the list of Int values between 1 and 4 (inclusive). We can write this more succintly as just

```
[1..4]
```

This will work with any element type that supports Enum. For example:

```
['a'..'h']
```

gives

```
"abcdefgh"
```

We can vary the "step" if we like:

```
[1,3..10]
```

gives

```
[1,3,5,7,9]
```

Note that if you want to count "down", you *must* provide a decreasing step:

```
[4..1]
```

gives

```
[]
```

What you really want is

```
[4,3..1]
```

We can even use this to construct *infinite* lists:

```
[1..]
```

gives the (infinite) list

```
[1,2,3,4,...]
```

Similarly,

```
[1,1..]
```

gives the infinite list of 1s:

```
[1,1,1,1...]
```

(A better way to construct an infinite list of a single value is to use `repeat`:

```
repeat 1
```

gives

```
[1,1,1,1...]
```

The difference is that `[a,a..]` requires the type of a to support Enum, while `repeat` can be used to repeat values of *any* type.)

Sometimes we want to build a list out of another list, by applying some operation to its elements. For example suppose we want the list of the squares of the integers from 1 to 4. I.e., we want to take `[1..4]` and from it square each element, producing `[1,4,9,16]`. We can do this with a *list comprehension*:

```
[x^2 | x <- [1..4]]
```

The left-hand side (to the left of the vertical bar) is the expression that is used to compute the elements of the new list. The right-hand side specifies where the original elements come from. So here, x will be bound to an element of `[1..4]`, x^2 will be computed, and the result saved in the corresponding position of the output list.

If we want, we can filter the values according to some criteria:

```
[x^2 | x <- [1..10], even x]
```

This will give the squares of only the *even* numbers between 1 and 10. But note that Haskell will keep "running" the list comprehension as long as the list generating x produces values. E.g., you might think you could do something like this:

```
[x^2 | x <- [1,2..], x <= 10]
```

and the list would stop after `x == 10`, but in fact it will run forever. Haskell doesn't know that, after `x == 10`, there won't, eventually, be another x that is <= 10, so it keeps on trying. Running forever makes Haskell sad; do you want to make Haskell sad?

You can also "drive" a list comprehension with more than one generator list:

```
[x+y | x <- [1,2], y <- [10,20]]
```

gives

```
[11,21,12,22]
```

(Can you see why?) You can think of this as a generalization of the notion of a "cross product" over all the input lists. Strangely enough, you can even "drive" a list comprehension with *no* generators:

```
[x | x < 10]
```

In this case, x must already be defined. If the condition is True, this will evaluate to `[x]`; if it is False it will evaluate to `[]`. Sometimes it may be useful to construct a zero-or-one element list, based on some condition; this is an easy way to do just that.

"Collapsing" lists: often we will want to take a list and collapse it down to a single value. For example, we might want to find the sum or product of a list of numbers, or maybe, given a list of Bools, determine whether they are all True. Haskell has a number of "aggregate" functions that do things like this:

- `sum` – Sums the elements of the list
- `product` – Finds the product of the elements of the list

- `maximum` – Returns the largest element of the list
- `minimum` – Returns the smallest element of the list
- `and` – Returns True if *every* element of the input list is True (i.e., it `&&`s all the elements of the input list together).
- `or` – Returns True if *any* element of the input list is True (i.e., it `||`s all the elements together)

## *Product Types*

We mentioned tuples above and showed have they have a type built from `,`; the `,` type is called a *product type*. A product type can be thought of as somewhat like a `struct` in C/C++: it aggregates together multiple values of different types, but the overall *structure* (the component types, their number, and order) is fixed at compile time. A product type is like a `struct` in which the elements are unnamed, they just have their relative ordering:

```
(1,"Hello") -- A value of product type (Int,String)
```

Tuples are useful when you want to pass around multiple values as if they were a single object. For example, you can use a tuple to return two values from a function:

```
minMax :: [Int] -> (Int, Int)
minMax l = (minimum l, maximum l)
```

In the arguments to a function, you can pattern-match against a tuple to extract the components:

```
f :: (Int, String) -> Int
f (i,s) = i + length s
```

Although this looks like a "normal" function in C/C++/Java, do not be deceived (and don't write all your functions to take tuples, just because they look familiar). A tuple is still a single value, so we can do the following:

```
f x  -- Provided that x has a value of type `(Int, String)
```

But of course, we can also construct the required tuple on-the-fly, from values of the component types:

```
f(userid,username)
```

One useful function that combines tuples and lists is `zip`:

```
zip [1,2,3,4] "ABCD" -- Gives [(1,'A'), (2,'B'), (3,'C'), (4,'D')]
```

This is useful if you want to process two lists in parallel with each other:

```
[x+y | (x,y) <- zip [1..4] [4..8]]
```

(In this case, there is another function, `zipWith`, that handles the problem of pairing up elements of two lists and then applying some binary operation to them. E.g. this is equivalent to the previous:

```
zipWith (+) [1..4] [4..8]
```

Note that if one of the lists is longer that the other, then `zip` will only work to the end of the shorter lists. This means you can use an infinite list safely:

```
numberElems :: [a] -> [(a,Int)]
numberElems l = zip l [0..]
```

There are two builtin functions that operate on pairs:

- `fst` returns the first element of a pair
- `snd` returns the second element of a pair

Note that these *only* work on pairs: for higher-dimensional tuples you will have to use pattern matching. E.g.,

```
let (x,y,z) = triple_thing in ...
```

## *First-class Functions*

Haskell is a *functional* language, which mostly means that functions exist as values: they can be stored in variables, passed into and returned out of functions, and even built-up from other functions. For example, we can do

```
plusone :: Int -> Int
plusone x = x+1

twice :: (Int -> Int) -> Int -> Int
twice f x = f(f(x))  -- Could also be written as f $ f x
```

We can now do something like

```
twice plusone 4
```

and the result will be 6 (i.e., `plusone(plusone(4))`).

There are a whole suite of builtin "higher order functions", functions that, like `twice`, take another function as an argument. Some examples:

```
map plusone [1,2,3,4]  -- Gives [2,3,4,5]
filter odd [1..10]     -- Gives [1,3,5,7,9]
foldr (+) 0 [1,2,3,4]  -- Gives 1+2+3+4+0 = 10
```

`foldr` can be thought of as performing a search-and-replace on a list. E.g., in the example above, the input list is `1:2:3:4:[]`. `:` gets replaced with `+`, while `[]` gets replaced with 0.

The `(+)` syntax for turning an operator into a "normal" function is actual just a fragment of the *sectioning syntax* that lets you leave off one argument to an operator and get back a function

```
(+1)  -- Same as the function plusone
(^2)  -- The function that squares its argument
(<10) -- Returns True if its argument is less than 10
(0==) -- Returns True if its argument is exactly 0
```

(But note that `(-1)` is not the function that decrements its argument, but just the literal numeric value -1. If you want the decrement function, you have to write `(+ (-1))`.) The comparison operator sections are useful with `filter`:

```
filter (>=0) list_of_numbers -- Keep only the positive values
```

*Currying* is an extension of sections to all functions, even those you write. It means that you can leave off the later arguments of a function, and you'll get back a new function. For example, suppose we have

```
f :: Int -> Float -> String -> Char
```

(where `a,b,c,d` are some types). If we call

```
f 1 3.5 "hello"
```

we will get back a Char. But if we call

```
f 1 3.5
```

we will get back a *function*, a function that takes a String and "finishes up", returning an Int. Similarly, if we leave off 3.5 we get a function that takes a Float and a String, and so forth. We can get a feeling for why this works by looking at the type of the function. In fact, the `->` type associates *to the right*, so in reality the type is

```
f :: Int -> (Float -> (String -> Char))
```

I.e., f takes an Int and returns a function. That function takes a Float and returns a function. *That* function (finally!) takes a String and returns a Char. In reality, all Haskell functions are unary; they take only one argument. But later arguments will be automatically passed to functions that are returned, so we can "fake" multiple argument functions. This ability is what lies behind the otherwise inexplicable function call syntax:

```
f 1 2.2 "3"
```

makes more sense if you imagine that the call will actually proceed like

```
((f 1) 2.2) "3"
```

## *Lazy Evaluation*

You may have heard that Haskell is a "lazy" language. As a way of introduction to what this means, take another look at the syntax for functions:

```
f x y z = ... -- three arguments
g x y = ...   -- two arguments
h x = ...     -- one argument
```

Under Haskell's syntax, what would a zero-argument function look like?

```
x = ...
```

In Haskell, a zero-argument function is indistinguishable from a variable. In particular, *using* a variable is semantically equivalent to "calling" a zero argument function. This means that definitions like this

```
x = x + 1
```

are perfectly valid. If you ever "call" x, then your program will go into an infinite loop, but the definition itself fine, albeit useless.

A more useful zero-argument function is something like this:

```
x = 1:x
```

In order to figure out what this means exactly, let's try to figure out the type. We know `1 :: Int`, and we also know that `(:) :: a -> [a] -> [a]`. Since the first argument to `:` is 1, `a` must be `Int`, which means that the type of the second argument (i.e., `x`) must be `[Int]`. So we actually have

```
x :: [Int]
x = 1:x
```

Let's evaluate out a couple of terms. After substituting the definition of `x` into its body once we have

```
x = 1:(1:x)
```

Do it again and we get

```
x = 1:(1:(1:x))
```

In fact, x is the (lazy) infinite list of 1s. Each occurence of x within the ever-expanding definition will be evaluated lazily; not when it is used, but only when it is actually needed. This is how we can deal with infinite lists. We can use the built-in `take` function to get a fragment of the list safely:

```
take 5 x  -- will output [1,1,1,1,1]
```

### *Typeclasses*

A Haskell typeclass is roughly akin to an interface in Java, or an abstract base class in C++. It defines a set of operations, but does not specify how they are implemented. For example, any type that supports the `Eq` typeclass supports both equality (==) and inequality (/=) but the actual implementation of these operators is left up to the type.

The most useful typeclasses to know about are

- Eq – supports (in)equality
- Ord – supports comparison operators
- Num – supports artithmetic operators (implies support for Eq and Ord as well)
- Show – supports conversion to String (i.e., for printing)
- Enum – supports enumeration over a range (i.e., we can ask for all the values of this type between a and b). The list syntax `[a..b]` requires that the type of a and b support Enum.

Note that product and list types support Eq, Ord, and Show, provided that the component types support them. I.e., because Int supports Ord, so does `[Int]`, so we can do:

```
[1,2,3] < [3,4,5]
```

This kind of comparison is done *lexicographically*; the first two components are compared, if they are equal then the second two, and so forth. (This is the kind of comparison you would do when looking a word up in the dictionary.)

In the type of a function, any type classes are shown before a `=>`:

```
f :: Eq a => [a] -> Bool
f (x1:x2:_) =   x1 == x2
```

We won't ask you to write functions with typeclass constraints, however. Knowledge of typeclasses is mostly useful for when you want to *look up* a function; most Haskell functions are polymorphic, so although you might expect to see a function of type `Int -> Int` it will probably have a type more like `Num a => a -> a` so that it works on any numeric type.

## *Preliminaries – Set Theory Review*

Here we are interested in the properties of *sets*. A set is an unordered collection of objects, all sharing some important property or properties, with an important restriction: *no duplicates* are present in a set. For example, this is a set:

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

But this is not:

Sets are normally written between curly-braces.

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

Since writing out the elements of a set is often tedious (for large sets), and sometimes impossible (for infinite sets), we also use *set builder notation*:

$$\{x^2 \mid x \text{ is a prime number }\}$$

would give us the set of all squares of primes (i.e., $\{4, 9, 25, 49, \dots\}$).

Some sets are so common that they have special names (we've seen some of these before); these are listed in figure 1.

| Set | Description |
|---|---|
| $\mathbb{N}$ | Natural numbers (i.e., whole numbers $\geq 0$) |
| $\mathbb{Z}$ | Integers |
| $\mathbb{Q}$ | Rational numbers (fractions) |
| $\mathbb{R}$ | Real numbers |
| $\mathbb{C}$ | Complex numbers |

Figure 1: Special sets

Sometimes you'll see the notation $\mathbb{Z}^*$ for "integers $\geq$ 0" and $\mathbb{Z}^+$ for "integers $\geq$ 1". Likewise, $\mathbb{N}_k$ is sometimes used for natural numbers modulo $k$, i.e., $\{0, 1, \dots, k-1\}$.

When we need to write a variable for an unknown set, we'll usually write it as an uppercase letter.

Often when we talk about sets we need to make it clear what kinds of things they can contain. We do this by stating what the *universe of discourse* is. The universe is just another set, denoted $\mathscr{U}$, but every other set we talk about must draw its elements from $\mathscr{U}$.

I.e., as described below, every set must be a *subset* of $\mathscr{U}$.

**Haskell note:**

In Haskell, we'll use lists as sets, so a set of elements of type `a` will have type `[a]`. We'll have to take care to ensure that our lists are always sorted and de-duplicated. We will provide you with a function `distinct` which will take an arbitrary list and transform it into a valid set:

```
ghci> distinct [1,3,2,1,3,5]
[1,2,3,5]
```

We will use the `elem` function as the equivalent to $\in$:

```
ghci> 3 `elem` [1,2,3,5]
True
```

And finally, we can use list comprehensions as an analogue to set builder notation. For example,

$$\{x^2 \mid x \in \{1 \dots 10\}\}$$

becomes

```
[x^2 | x <- [1..10]]
```

## *Operations on sets*

Perhaps the fundamental operation on sets is *membership*. The logical proposition $e \in S$ states "$e$ is a member of set $S$". Likewise, $e \notin S$ states "$e$ is not a member of $S$". Since $\in$ is a proposition, we can make logical statements using it:

$$e \in S \to e \in \mathscr{U}$$

This expresses our statement above that all the elements of the sets we are talking about must be elements of the universe.

If we want to relate two sets to each other, we can ask, to what extent do they overlap? If, given two sets $A$ and $B$, we find that every element of $A$ is also an element of $B$ then we say that "$A$ is a *subset* of $B$" and write this as $A \subseteq B$. Logically,

$$A \subseteq B \Leftrightarrow \forall a \in A : a \in B$$

If $A \subseteq B$ then we can also say that $B$ is a *superset* of $A$.

Sometimes we write $A \subset B$ to mean that $A$ is a "proper" subset of $B$, that is, $A \subseteq B \wedge A \neq B$.

There are a few easy-to-verify tautologies about set membership and the subset relation:

$$a \in A \wedge A \subseteq B \to a \in B \qquad \text{(Definition of } \subseteq\text{)}$$
$$A \subseteq B \wedge B \subseteq A \to A = B \qquad \text{(Symmetry)}$$
$$A \subseteq B \wedge B \subseteq C \to A \subseteq C \qquad \text{(Transitivity)}$$
$$A \subseteq A, \text{ for any set } A \qquad \text{(Reflexivity)}$$

Given two sets, there are a number of operations we can perform that will result in a new set:

$$A \cup B \qquad \text{(Union)}$$
$$A \cap B \qquad \text{(Intersection)}$$
$$A \setminus B \qquad \text{(Set difference)}$$
$$A \ominus B \qquad \text{(Symmetric difference)}$$

Each of these can be defined by a set builder on $\mathscr{U}$ and the logical equivalent to the above descriptions:

| Operation | Set builder expression |
|---|---|
| $A \cup B$ | $\{e \mid e \in \mathscr{U} \wedge (e \in A \vee e \in B)\}$ |
| $A \cap B$ | $\{e \mid e \in \mathscr{U} \wedge (e \in A \wedge e \in B)\}$ |
| $A \setminus B$ | $\{e \mid e \in \mathscr{U} \wedge (e \in A \wedge e \notin B)\}$ |
| $A \ominus B$ | $\{e \mid e \in A \cup B \wedge e \notin A \cap B\}$ |

Figure 2: Set operations

The *cardinality* of a set $A$, written $|A|$ roughly defines the "size" of the set. For finite sets, this is simply the number of (unique) elements in the set.

Interestingly, there are different "sizes" of infinite sets. For example, $|\mathbb{N}| < |\mathbb{R}|$; the set of real numbers has *more* elements than the set of natural numbers, even though both are infinite!

If $A \subseteq B$ then we have $|A| \leq |B|$. Similarly, if $A \subset B$ then $|A| < |B|$.

Some properties relating cardinality and the above set operations are easy to derive:

Figure 3: Cardinality of set operations

$$\begin{gathered}
|A \cup B| = |A| + |B| - |A \cap B| \\
|A \cap B| \leq |A| \wedge |A \cap B| \leq |B| \\
\text{or equivalently} \\
|A \cap B| \leq \max |A|, |B| \\
|A \setminus B| = |A| - |A \cap B| \\
|A \ominus B| = |A \cup B| - |A \cap B|
\end{gathered}$$

For sets constructed via the set builder notation, note that we have

$$|\{\dots \mid s_1 \in S_1, s_2 \in S_2, \dots, s_n \in S_n, \dots\}| \leq \prod_{i=1}^{n} |S_i|$$

*Powersets:* The *powerset* of a set $A$ is the *set of all subsets* of $A$. That is

$$\mathscr{P}(A) = \{A' \mid A' \subseteq A\}$$

For example,

$$\mathscr{P}(\{1,2\}) = \{\varnothing, \{1\}, \{2\}, \{1,2\}\}$$

Note that the powerset of *any* set always includes the empty set. This is true even of the empty set itself:

$$\mathscr{P}(\varnothing) = \{\varnothing\}$$

Note that $\{\varnothing\} \neq \varnothing$. The former is a set containing a single element, while the later contains no elements.

One way to view the construction of the powerset is as if each element of the input set had a "switch" attached to it, labeled IN/OUT. We can construct any particular subset by flipping the switches: those whose elements are IN will be in the subset, and those which are OUT will not. I.e., we have something like

$$\{1_{\text{IN}}, 2_{\text{OUT}}, 3_{\text{OUT}}, 4_{\text{IN}}\} \longrightarrow \{1, 4\}$$

By looking at all possible configurations of switches, we can see all the possible subsets.

Here, for example, are all the subsets of $\{1,2,3\}$:

| 1 | 2 | 3 | Subset |
|---|---|---|---|
| Out | Out | Out | $\emptyset$ |
| In | Out | Out | $\{1\}$ |
| Out | In | Out | $\{2\}$ |
| In | In | Out | $\{1,2\}$ |
| Out | Out | In | $\{3\}$ |
| In | Out | In | $\{1,3\}$ |
| Out | In | In | $\{2,3\}$ |
| In | In | In | $\{1,2,3\}$ |

We can also generate the powerset by an inductive formulation:

$$\mathscr{P}(\emptyset) = \{\emptyset\} \qquad \text{(Base case)}$$

Let $A = \{x\} \cup A'$ with $x \notin A'$. Then

$$\mathscr{P}(A) = \mathscr{P}(\{x\} \cup A') = \mathscr{P}(A') \cup \{\{x\} \cup P \mid P \in \mathscr{P}(A')\} \qquad \text{(Inductive case)}$$

That is, at each inductive step, we choose an arbitrary element of $A$, remove it producing $A'$, recursively construct $\mathscr{P}(A')$, and then add $x$ to every set in $\mathscr{P}(A')$ and union that with the (unchanged) $\mathscr{P}(A')$. At some point we will remove the last element, at which point $A' = \emptyset$ and the recursive powerset will use the base case.

*Cardinality of $\mathscr{P}$:* How many subsets does a given set have? Looking at the above table, we see that for a set with three elements, there are eight possible subsets, including the empty set. For each element in the original set, it's "switch" can be in one of two states. Thus, by the rule of product, the total number of possible configurations is $2 \times 2 \times 2 = 2^3 = 8$. In general, if $n = |A|$ then $2^n = |\mathscr{P}(A)|$. Figuring out how many subsets to expect is a good way to check your work, if you are trying to generate all the subsets.

*Equality between sets:* Since sets are unordered, we define equality between sets to mean "containing the same elements". That is,

$$A = B \text{ iff } (\forall a \in A \colon a \in B) \land (\forall b \in B \colon b \in A)$$

or equivalently,

$$A = B \text{ iff } A \subseteq B \land B \subseteq A$$

Two sets with the same cardinality (possibly infinite, but of the "same size" infinite) are *isomorphic* even if they are not equal. Isomorphism just means that we can construct a pairing of elements between the two sets, such that each element from $A$ is paired with a single unique element from $B$, and each element of $B$ is paired with a single unique element of $A$. Because we can always move "back and forth" between the two sets, we can convert any operation on $A$ into an operation on $B$, and vice versa, without loss of information. Thus, two isomorphic sets can be regarded as being "extensionally equivalent", because anything we can do on the one can be done on the other.

## *Strings, Languages, and Grammars*

*A note on notation:* As before, we will use uppercase letters $A, B, C, \dots$ to refer to unknown sets. $L$ will be used to refer to languages, for example, $L_{\text{email}}$ the language of valid email address, or $L_{\text{prime}}$ the language of prime numbers (expressed in some suitable base). We will use lowercase $a, b, c, \dots$ for unknown symbols (i.e., elements of $\Sigma$) and lowercase $s, t, u, \dots$ for strings (elements of $\Sigma^*$). Thus, $as$ signifies a string that starts with the (unknown) character $a$ and continues with the string $s$. Literal characters and strings will be written in typewritter font: $\Sigma = \{\texttt{a}, \texttt{b}\}$, $s = \texttt{ababa}$, etc.

*Strings:* As always when dealing with sets, we must define our universe of discourse. Here, we begin with $\Sigma$, the *alphabet* under consideration. For example, if $\Sigma = \{\texttt{a}, \texttt{b}\}$ then the only symbols we are interested in are `a` and `b`. We define $\mathscr{U} = \Sigma^*$, the set of all *strings* over the alphabet. A string is a possibly-empty sequence of symbols. For example, given $\Sigma = \{\texttt{a}, \texttt{b}\}$, some possible strings are `aa`, `b`, and `ababa`. The empty string is denoted $\varepsilon$. If we wish to formalize the structure of a string, we can define it inductively by:

$$\text{Eps}\,\frac{}{\varepsilon \in \Sigma^*} \qquad \text{Sym}\,\frac{a \in \Sigma \qquad s \in \Sigma^*}{as \in \Sigma^*}$$

We overload the notation $|s|$ to denote the *length* of the string $s$. For example, we have $|\texttt{ababa}| = 5$ while $|\varepsilon| = 0$. We can define $|s|$ inductively by

$$\text{Len-Eps}\,\frac{}{|\varepsilon| = 0} \qquad \text{Len-Sym}\,\frac{|s| = l}{|as| = l + 1}$$

We can *concatenate* strings to form larger strings. To make this explicit, we will use the $\cdot$ operator: $\texttt{aaa} \cdot \texttt{bbb} = \texttt{aaabbb}$. When it is unambiguous, we will indicate concatenation by juxtaposition; i.e., if $s_1$ and $s_2$ are strings, then $s_1 s_2 = s_1 \cdot s_2$. Again, for future reference, we give an inductive definition:

$$\text{Cat-Eps}\,\frac{}{\varepsilon \cdot s = s} \qquad \text{Cat-Sym}\,\frac{s_1 \cdot s_2 = s}{as_1 \cdot s_2 = as}$$

Sometimes we will find it useful to reverse a string; we denote the reversal of a string as $S^R$ and define it inductively as

$$\text{Rev-Eps}\,\frac{}{\varepsilon^R = \varepsilon} \qquad \text{Rev-Sym}\,\frac{s^R = s'}{(as)^R = s'a}$$

Sometimes we will need to express that a string $s_1$ is a *prefix* of another string $s$. For example, `aba` is a prefix of `ababbb`. We write this as $\texttt{aba} \sqsubseteq \texttt{ababbb}$.

Inductively, for a string $s \in \Sigma^*$

$$\text{Pre-Eps}\,\frac{}{\varepsilon \sqsubseteq s} \qquad \text{Pre-Sym}\,\frac{s_1 \sqsubseteq s}{as_1 \sqsubseteq as}$$

Note that for every string, it is the case that $s \sqsubseteq s$. If we want to express "proper" prefixes (i.e., a prefix but *not* equal to the entire string) we will write $s_1 \sqsubset s$.

Example 0.1 *Prove that*

$$\forall s_1, s_2 : |s_1 \cdot s_2| = |s_1| + |s_2|$$

PROOF. The proof proceeds by induction on the length of $s_1$.

- Base case: $s_1 = \varepsilon, |s_1| = 0$. Then, by rule Cat-Eps, $\varepsilon \cdot s_2 = s_2$ and we have $|\varepsilon \cdot s_2| = |s_2| = 0 + |s_2|$.
- Inductive case: $s_1 = as'_1$ for some $s'_1$. Then by rule Cat-Sym we have $|as'_1 \cdot s_2| = 1 + |s'_1 \cdot s_2|$. But by the IH we have $|s'_1 \cdot s_2| = |s'_1| + |s_2|$. By rule Len-Sym $|as'_1| = 1 + |s'_1|$, so we have $|as'_1 \cdot s_2| = 1 + |s'_1| + |s_2|$. QED.

*Languages:* A *language* is some (possibly empty) subset of $\Sigma^*$; a selection of strings that have some property we are interested in. For example, $\{\mathtt{aa}, \mathtt{bb}\}$, $\{\mathtt{ababa}\}$, $\{\varepsilon\}$ and $\{\} = \varnothing$ are all valid languages.

Note that there is an important difference between $\{\varepsilon\}$ and $\varnothing$; the former contains only the empty string, while the latter contains no strings at all.

Since enumerating the strings in a language is impractical for all but the smallest sets, we will often use either set-builder notation or inductive definitions to define infinite languages. For example:

The language with an equal number of a's and b's, with all b's following all a's:

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

The language $L_{\mathtt{a}+}$ of strings of a's of length $\geq 1$:

$$\frac{}{\mathtt{a} \in L_{\mathtt{a}+}} \qquad \frac{s \in L_{\mathtt{a}+}}{\mathtt{a}s \in L_{\mathtt{a}+}}$$

As languages are sets, the setwise union, intersection, complement (relative to $\Sigma^*$), etc. are all defined as one would expect. Two additional set operations are available on sets of strings, language *concatenation* and the *Kleene star*.

Concatenation for languages is defined as the cross-wise concatenation of all their strings. That is, for two languages $L_1$ and $L_2$ we have

$$L_1 \cdot L_2 = \{s_1 \cdot s_2 \mid s_1 \in L_1, s_2 \in L_2\}$$

As with string concatenation, we will omit the $\cdot$ when this is unambiguous. Some properties of language concatenation:

- $\varnothing \cdot L = L \cdot \varnothing = \varnothing$

- $\{\varepsilon\} \cdot L = L \cdot \{\varepsilon\} = L$
- If $\varepsilon \in L_2$ then $L_1 \subseteq L_1 \cdot L_2$ and similarly, if $\varepsilon \in L_1$ then $L_2 \subseteq L_1 \cdot L_2$.

The *Kleene star* operation $L^*$ generates all possible self-concatenations of a language. That is,

$$L^* = \bigcup_{i \in \mathbb{N}} L^i$$

Some properties of $L^*$:

- $\varnothing^* = \{\varepsilon\}^* = \{\varepsilon\}$
- If $L \neq \varnothing$ and $L \neq \{\varepsilon\}$ then $|L^*| = \infty$
- $L \subseteq L^*$ for any $L$.

We summarize the properties of these operations in figure 4.

| Identity | Description |
|---|---|
| $\varnothing \cup L = L \cup \varnothing = L$ | $\varnothing$ is the identity of $\cup$ |
| $\varnothing \cdot L = L \cdot \varnothing = \varnothing$ | $\varnothing$ is the zero of $\cdot$ |
| $\{\varepsilon\} \cdot L = \{\varepsilon\} \cdot L = L$ | $\{\varepsilon\}$ is the identity of $\cdot$ |
| $\varnothing^* = \{\varepsilon\}$ | From the definition of the Kleene star |
| $\{\varepsilon\}^* = \{\varepsilon\}$ | From the definition of the Kleene star |
| $L \subseteq L^*$ | |

Figure 4: Identies of language operations

Note that we do not, at this stage, place any restriction on how a language may be defined. For example, this is a perfectly acceptable language:

$$L_{\text{prime}} = \{p \mid p \text{ is prime}\}$$

In fact, the notions of languages and strings provides a framework for thinking about all sorts of problems. Any problem in which we are given some input, expressible as a sequence of symbols, and have to determine whether it satisfies some criteria, can be phrased as "is the string $s$ a member of a suitably defined language?".

## *Grammars*

A *grammar* gives a way of defining a language by constructing a collection of mutually-inductive sets which serve as the building-blocks of the language of interest. Formally, a grammar is a four-tuple $G = (V, \Sigma, R, S)$ where

- $V$ is the set of *non-terminals* (sometimes called variables).
- $\Sigma$ is the alphabet, with $\Sigma \cap V = \varnothing$.
- $R$ is the set of *rules* (or productions), with $R \subseteq V \times (V \cup \Sigma)^*$.

- $S$ is the *start symbol*, a distinguished element of $V$ (i.e., $S \in V$).

  Less formally, a grammar looks like:

$$A \to \varepsilon$$
$$A \to \mathtt{a}A$$
$$A \to A\mathtt{b}$$

where we assume that $S$ is the first non-terminal defined in a rule, and the sets $V$ and $\Sigma$ consist of the non-terminals and terminals used in the grammar.

This grammar generates the language $L_{\mathrm{ab}} = \{a^m b^n \mid m, n \in \mathbb{N}\}$.

We denote the language generated by a grammar $G$ as $L(G)$. The language generated by a grammar can be constructed by following rules, starting with the start symbol. To present this more formally, we will extend our notion of strings to allow

- *Terminal* strings $\in \Sigma^*$.
- *Nonterminal* strings $\in (V \cup \Sigma)^*$.

Nonterminal strings are allowed to have non-terminals in them. For example, $\mathtt{ab}A\mathtt{ba}$ is a nonterminal string. Our process will intuitively consist of starting with the string $S$ (i.e., the start symbol) and "applying" rules to the non-terminals in it until we reach a terminal string. We define application of a rule to a string as

$$\frac{s = s_1 A s_2 \qquad (A \to D) \in R}{s \Longrightarrow s_1 D s_2}$$

(I.e., the possibly-nonterminal string on the right-hand side of the rule is "spliced in" to the string in place of $A$.)

We extend $\Longrightarrow$ to the idea of *derivability*: a string $s_2$ is *derivable* from $s_1$ (written $s_1 \overset{*}{\Longrightarrow} s_2$) if there is a sequence of zero-or-more rule applications $s_1 \Longrightarrow s_1' \Longrightarrow s_1'' \Longrightarrow \ldots \Longrightarrow s_2$. We can define $\overset{*}{\Longrightarrow}$ as

$$\frac{}{s \overset{*}{\Longrightarrow} s} \qquad \frac{s_1 \Longrightarrow s_1' \qquad s_1' \overset{*}{\Longrightarrow} s_2}{s_1 \overset{*}{\Longrightarrow} s_2}$$

Then we say that the language $L(G)$ generated by $G$ is given by

$$L(G) = \{s \mid S \overset{*}{\Longrightarrow} s, s \in \Sigma^*\}$$

(I.e., it is the set of terminal strings that are derivable from the start symbol $S$.)

## *Proof of grammar/language equivalence*

We can often intuitively see that the language of a grammar matches that of some other (e.g., set-builder) specification. However, we wish to prove this more rigorously. In order to prove $L = L(G)$ we must show that $L \subseteq L(G)$ (i.e., every string in $L$ is in $L(G)$) and $L(G) \subseteq L$ (i.e., that every string in $L(G)$ is in $L$).

In this sections we will develop the tools for proving these inclusions. We will use the grammar

$$\begin{aligned} \mathrm{r}_1 : &\quad S \to \varepsilon \\ \mathrm{r}_2 : &\quad S \to \mathtt{a}S \\ \mathrm{r}_3 : &\quad S \to S\mathtt{b} \end{aligned}$$

where $L(G) = \{\mathtt{a}^m\mathtt{b}^n \mid 0 \le m, n\}$.

To show $L \subseteq L(G)$ we must construct a derivation in $G$ for every string in $L$. Since $L$ is infinite we obviously cannot do this directly, so instead we construct a template sequence of applications in terms of $m$ and $n$. The template here is

$$\begin{aligned} S &\overset{m}{\Longrightarrow} \mathtt{a}^m S && \text{(by } \mathrm{r}_2\text{)} \\ \mathtt{a}^m S &\overset{n}{\Longrightarrow} \mathtt{a}^m S\mathtt{b}^n && \text{(by } \mathrm{r}_3\text{)} \\ \mathtt{a}^m S\mathtt{b}^n &\Longrightarrow \mathtt{a}^m\mathtt{a}^n && \text{(by } \mathrm{r}_1\text{)} \end{aligned}$$

By applying rule $\mathrm{r}_2$ $m$ times we can generate $m$ a's at the beginning of the string; by applying rule $\mathrm{r}_3$ $n$ times we can generate $n$ b's at the end. Finally, we finish with rule $\mathrm{r}_1$ in order to eliminate the $S$ which remains in the middle and close the string. (We could, of course, reverse the order of the applications of $\mathrm{r}_2$ and $\mathrm{r}_3$ without changing the result.)

To show $L(G) \subseteq L$ we must show that every terminal string derived from $S$ by a finite number of rule applications is in $L$. Because the "length" of a derivation is the number of applications used in it, we can use a proof by induction on the number of applications. Developing our inductive hypothesis, however, will be somewhat tricky, and is often dependent on the details of the grammar in question. We want a property which can be (easily!) proved about derivations of length 1 (i.e., applications to $S$), and which we can show holds for derivations of length $n$, if it is assumed to hold about derivations of length $n+1$. In the case of $L$, the only relevant (or possible) property is that all a's come before all b's in the string. The language places no restrictions on relationship on the number of a's or b's; in more complex grammars, the terminals and non-terminals may have interrelationships that the inductive step must be shown to preserve.

PROOF. For any derivation $S \overset{*}{\Longrightarrow} s$ in G, $s$ has the property that all a's appear before all b's. By induction on the length of the derivation.

- Base cases:

$$\begin{aligned} S &\Longrightarrow \varepsilon && \text{(by } \mathrm{r}_1\text{, trivial)} \\ S &\Longrightarrow \mathtt{a}S && \text{(by } \mathrm{r}_2\text{, a at the head)} \\ S &\Longrightarrow S\mathtt{b} && \text{(by } \mathrm{r}_3\text{, b at the tail)} \end{aligned}$$

- Inductive cases: Assuming that $\ldots \overset{n}{\Longrightarrow} s$ has a's before all b's, we show that $s \Longrightarrow s'$ preserves this property, by showing that every possible application preserves it:

$$\begin{aligned} S &\Longrightarrow \varepsilon && (\mathrm{r}_1, \text{trivial}) \\ S &\Longrightarrow \mathtt{a}S && (\mathrm{r}_2, \text{adds an } \mathtt{a} \text{ at the head}) \\ S &\Longrightarrow S\mathtt{b} && (\mathrm{r}_3, \text{adds a } \mathtt{b} \text{ at the tail}) \end{aligned}$$

Because every possible application either adds no new terminals ($\mathrm{r}_1$) or adds a terminal in the appropriate order, we conclude that the property is preserved by the $(n+1)$-th application. QED.

## *Regular Languages*

Here we are interested in investigating the properties of a set of languages known as the *regular languages*. We denote this set $\mathsf{REG}$. The set $\mathsf{REG}$ is defined inductively from the operations we've already seen:

$$\text{Empty}\frac{}{\varnothing \in \mathsf{REG}} \qquad \text{Symbol}\frac{a \in \Sigma}{\{a\} \in \mathsf{REG}}$$

$$\text{Concat}\frac{R_1 \in \mathsf{REG} \quad R_2 \in \mathsf{REG}}{R_1 \cdot R_2 \in \mathsf{REG}} \qquad \text{Union}\frac{R_1 \in \mathsf{REG} \quad R_2 \in \mathsf{REG}}{R_1 \cup R_2 \in \mathsf{REG}}$$

$$\text{Star}\frac{R \in \mathsf{REG}}{R^* \in \mathsf{REG}}$$

We can also define the set of regular langauges as those generated by *regular grammars*: A grammar is regular if every rule can be put into one of the forms

$$\begin{aligned} A &\rightarrow \mathtt{a} \\ A &\rightarrow \mathtt{a}B \\ A &\rightarrow \varepsilon \end{aligned}$$

Note that the language generated by a grammar may still be regular even if it does not have this form, so long as it can be *transformed* into the above form. For example, the grammar

$$\begin{aligned} A &\rightarrow A\mathtt{a} \\ A &\rightarrow \varepsilon \end{aligned}$$

which defines the language $\{a^n \mid n \geq 0\}$ is not regular, but it is equivalent to the grammar

$$\begin{aligned} A &\rightarrow \mathtt{a}A \\ A &\rightarrow \varepsilon \end{aligned}$$

which is.

Intuitively, you can think of regular languages as languages which have the ability to perform *repetition*, but not the ability to "count" or "keep track" of how many times something has occured. Thus, for example, the language of arithmetic expressions with parentheses is *not* regular, because matching parentheses requires "keeping track" of how deeply we are nested. Regular languages have no "memory".[1]

[1] We will see later that this is *literally* the case: deterministic finite state machines, the computational implementation of regular languages, do not have any memory!

## *Properties of regular languages*

Regular languages can have several interesting properties which will be of use to use later.

Definition 0.1 A language $L$ is *empty* if $L = \varnothing$.

Definition 0.2 A language $L$ is *nullable* if $\varepsilon \in L$.

Definition 0.3 A language $L$ is *null* if $L = \{\varepsilon\}$.

Definition 0.4 A language $L$ is *infinite* if $L$ does not contain finitely many strings. A language that is not infinite is *finite*.

## *Closure properties of regular languages*

Since we have built-up regular languages union, concatenation, and Kleene star operations, it should be obvious that REG is *closed* under these same operations: if we take (e.g.) the union of two regular languages, the result will still be regular. Is this true of the other set operations? I.e., is REG closed under intersection, complement, difference, and symmetric difference? What if we define the reversal of a regular language $L^R$ to be

$$L^R = \{s^R \mid s \in L\}$$

Is the reversal of a regular language also regular?

In fact, the answer to all these is Yes. The proofs for the set operations are all interrelated (because $A - B = A \cap \neg B$, $\neg A = \Sigma - A$, $A \cap B = \neg(\neg A \cup \neg B)$, etc.) The proofs proceed on the structure of a RE; since we have already established that $L(\text{RE}) = \text{REG}$, this is sufficient. We will demonstrate the process for the reversal; proofs of the other closure properties are left as an exercise for the reader.

Which might make an appearance on the midterm or final...

Lemma 0.5 ?? *For $s \in \Sigma^*$, if $s = s_1 s_2$ then $s^R = s_2^R s_1^R$.*

PROOF. *For $r \in \text{RE}$, $r^R \in \text{RE}$. Proof by induction on the structure of $r$.*

- *If $r = 0$ then $L(r) = \varnothing$ and $\varnothing^R = \varnothing$.*
- *If $r = a, a \in \Sigma$ then $L(r) = \{a\}$ and $\{a\}^R = \{a\}$.*
- *If $r = r_1 \cup r_2$ then by the induction hypothesis, $r_1^R \in \text{RE}, r_2^R \in \text{RE}$, and $r^R = r_1^R \cup r_2^R$ so it follows that $r^R \in \text{RE}$.*

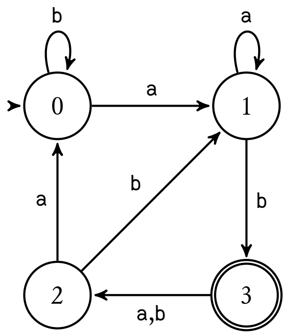
- *If $r = r_1 r_2$ then let $r' = r_2^R r_1^R$. By IH, $r_1^R \in$ RE and $r_2^R \in$ RE, and thus $r' \in$ RE.*
- *If $r = r_1^*$ then $r' = (r_1^R)^*$. By IH, $r_1^R$ is regular, and thus $r' \in RE$.*

$\dashv$

## *The Pumping Lemma for Regular Language*

Showing that a language *is* regular is relatively easy: just construct one of the valid representations for a REG for it (literal finite set, RE, DFA, NDA(-$\varepsilon$), or regular grammar). Showing that a language *is not* regular is somewhat more tricky, as we would have to show that no such representation can be constructed. In this section we will derive a general result about *all* regular languages and representations, which will give us a method for categorizing languages which cannot be regular.

Consider a DFA $M$:



A *path* through a machine is an ordered sequence of nodes. For example, one path in $M$ from state 0 to 3 is $\mathbf{p} = \langle 0, 1, 3, 2, 1, 3 \rangle$. Note that the *length* of a path $|\mathbf{p}|$ is defined to be the number of nodes −1. I.e., for the path given we have $|\mathbf{p}| = 5$.

The length of a path gives the number of *transitions* in the path, not the number of nodes visisted, and thus is one less than the number of nodes.

A *cycle* is a path $\langle n_0, n_1, \dots n_i \rangle$ such that $n_0 = n_i$. Note that there is a subpath of $\mathbf{p}$ which is a cycle: $\langle 1, 3, 2, 1 \rangle$. In fact, for a machine with $k$ states, we can easily see than any path $\mathbf{p}$ with $|\mathbf{p}| \geq k$ *must* have a cycle, because such a path will visit $k + 1$ nodes, and by the pigeonhole principle, must visit at least one node more than once.

Note, also, that in a DFA we have no way to distinguish one trip around a cycle from another, so if we are generating strings and we find a cycle, we know that *any number* of trips around the cycle (including 0) will produce a valid string. This forms the key part of the pumping lemma: for a "sufficiently long" string, we are guaranteed to find a cycle; if we can show that some of the strings resulting from repeating the cycle ("pumping") are *not* in the language, then the language is not regular, because there is no way to build a DFA that can "count" the number of trips around a cycle and thus distinguish between them.

Put more concretely, assuming our machine has $k$ states, we are looking for a string of length $\geq k$. For the above machine, we could take the string `ababb`. Then, because this string must have visited $\geq k + 1$ states, we know that it contains a cycle. (In this case, the cycle is a(bab)b.) And, since $L$ is assumed to be regular, we know that we should be able to "pump" the cycle, repeating it: $\mathtt{a(bab)}^i\mathtt{b}, i \geq 0$ and all the resulting strings will still be in $L$.

If the string is *longer* than $k$, then there is some prefix of length $k$ that contains a cycle. If it is not obvious why this must be the case, consider that a prefix of length $k$ is still a string of length $k$, and thus must have visited $k + 1$ nodes, and thus, some node more than once. So we do not need to look a strings longer than $k$, although it will often be useful to do so.

A cycle cannot have length 0, it must include at least one transition (although it could be a cycle such as $\langle 1, 1 \rangle$ which visits a single node twice). Hence, the portion of the string which is generated by the cycle must have length at least 1. This leads us to the full definition of the pumping lemma:

Lemma 0.6 *The Pumping Lemma for regular languages*

*Let $L$ be a regular language, generated by a* DFA *with $k$ states. Then for every $s \in L$ with $|s| \geq k$, we have*

$$\begin{aligned} s &= uvw \quad \text{for some substrings } u, v, w \\ |uv| &\leq k \\ |v| &\geq 1 \\ uv^iw &\in L \quad \text{for any } i \geq 0 \end{aligned}$$

(Note that this definition allows one or both of $u, w$ to be $\varepsilon$.)

To *apply* the pumping lemma, we assume that the language in question is regular, and can be generated by a machine with $k$ states (leaving $k$ unknown). We then construct a string $s$ in the language, usually with the structure of the string depending on $k$. Finally, we show that there is *no* decomposition of $s$ into $u, v, w$ such that $uv^iw \in L$ for all $i \geq 0$.

Example 0.2 *Prove that the language*

$$L = \{\mathtt{a}^i\mathtt{b}^i \mid i \geq 0\}$$

*is not regular.*

Assume that $L$ is regular, and is generated by some DFA with $k$ states. Let

$$s = \mathtt{a}^k\mathtt{b}^k \quad (\in L)$$

Then, for any decomposition of $s = uvw$ according to the conditions above, we have $uv = \mathtt{a}^i$ with $i \geq 1$. (Because there are $k$ a's at the beginning of the string, and we have $|uv| \leq k$ we know that the $uv$ portion of the string consists of some number of a's, and no b's, and because $|v| \geq 1$ we know that $uv$ consists of at least one a.) But then we have $v = a^j$ with $1 \leq j \leq k$. If we pump $v$, then the prefix $uv$ will consist of $k'$ a's with $k' > k$ and thus the resulting string is *not* in $L$. ⊣

## Regular Expressions

### Names for regular languages

Defining a regular language by giving the operations used to build it is cumbersome: $(\{a\} \cdot \{b\}) \cup \{b\}^*$. We would like a way to "name" regular languages that succinctly and clearly represents the operations used to build them. This role is served by *regular expressions*.[2]

[2] Note that regular expressions as we define them are a more limited form of the regular expressions commonly seen in normal programming languages. REs have been extended over the years with named captures, backreferences, etc. to the extent that they hardly resemble their theoretical foundations.

We define the set of regular expressions $\mathsf{RE}$ inductively as follows:

$$\text{Empty}\,\frac{}{0 \in \mathsf{RE}} \qquad \text{Symbol}\,\frac{\mathsf{a} \in \Sigma}{\mathsf{a} \in \mathsf{RE}}$$

$$\text{Union}\,\frac{r_1 \in \mathsf{RE} \quad r_2 \in \mathsf{RE}}{r_1 + r_2 \in \mathsf{RE}} \qquad \text{Concat}\,\frac{r_1 \in \mathsf{RE} \quad r_2 \in \mathsf{RE}}{r_1 r_2 \in \mathsf{RE}}$$

$$\text{Star}\,\frac{r \in \mathsf{RE}}{r^* \in \mathsf{RE}}$$

It is important to note that a regular expression *is not* a regular language. A regular expression *names* a regular language, but we still have to define how a regular expression is interpreted. We define the function $L(r)$ to be the interpretation of $r$ as a regular language. (The names of the rules above should clue you in to their function when translated to regular languages.)

$$\begin{aligned} L(0) &= \varnothing \\ L(\mathsf{a}) &= \{\mathsf{a}\} \\ L(r_1 + r_2) &= L(r_1) \cup L(r_2) \\ L(r_1 r_2) &= L(r_1) \cdot L(r_2) \\ L(r^*) &= L(r)^* \end{aligned}$$

### Properties of regular expressions

We can determine all of the properties listed in section ?? by examining the structure of the regular expression that generates it. This is obviously better than trying to enumerate the strings in the generated language in order to test the property. As an example, we will describe a recursive procedure for determining whether a language $L(r)$ is empty. Note that just because $L(r) = \varnothing$ it is not necessarily the case that $r = 0$! For example, $L((\mathsf{a}0) + 0) = \varnothing$.

- If $r = \varnothing$ then $r$ is empty.
- If $r = a, a \in \Sigma$ then $r$ is not empty.
- If $r = r_1 \cup r_2$ then if either $r_1$ or $r_2$ is non-empty, then $r$ is non-empty.
- If $r = r_1 r_2$ then if either $r_1$ or $r_2$ is empty, $r$ is empty.
- If $r = r'^*$ then false (because the Kleene star always contains at least $\varepsilon$).

## *Matching regular expressions directly*

Here we are concerned with the problem of writing a function which takes a regular expression and a string and returns `True` if the RE *matches* the string. That is, the function determines whether $s \in L(r)$ for string $s$ and regular expression $r$.

Formally, we match a regular expression against a string by structural recursion on the RE. The only slightly tricky cases are for $\cdot$ and $^*$ as we shall see.

Definition of $s \in L(r)$ by structural recursion on $r$:

$$\text{Zero}\,\frac{\qquad}{s \notin \varnothing} \qquad \text{Symbol}\,\frac{a \in \Sigma}{a \in a}$$

$$\text{Union}\,\frac{s \in r_1 \vee s \in r_2}{s \in r_1 + r_2} \qquad \text{Cat}\,\frac{s = s_1 s_2 \quad s_1 \in r_1 \quad s_2 \in r_2}{s \in r_1 \cdot r_2}$$

$$\text{Star-}\varepsilon\,\frac{\qquad}{\varepsilon \in r^*} \qquad \text{Star}\,\frac{s = s_1 s_2 \quad s_1 \neq \varepsilon \quad s_1 \in r \quad s_2 \in r^*}{s \in r^*}$$

In both the Cat and Star cases, we have the premise $s = s_1 s_2$. $s_1$ and $s_2$ must range over all possible *splits* of $s$. That is, if $s =$ `ab` then it is necessary to test the premises over all of $s_1 = \varepsilon, s_2 =$ `ab`; $s_1 =$ `a`, $s_2 =$ `b`; $s_1 =$ `ab`, $s_2 = \varepsilon$. In the case of Star, we remove the possibility that $s_1 = \varepsilon$ as the Star-$\varepsilon$ rule handles this case.

The necessity to test Cat and Star over all possible splits leads to this method being very inefficient. For RE with many nested stars or concatenations, the same splits being repeatedly computed and tested.

## *An incorrect method for matching regular expressions*

One commonly-expounded method for matching RE is to match the RE against the longest possible *prefix* of the string; if the "prefix" that matches is in fact the entire string, then the match is successful. This method is incorrect, due to the seemingly-innocuous phrase "longest possible". Recall that an RE such as `a`$^*$ when part of a larger RE does not match as many `a`'s as *possible*, but rather as many as *necessary*. The difference is clear if we consider a pair of equivalent REs such as

$$\texttt{a}^*\texttt{a} \qquad \text{and} \qquad \texttt{aa}^*$$

These should both match all strings consisting of 1 or more `a`'s. However, using the "greedy" algorithm described in this section, the second will match but the first will fail; the initial `a`$^*$ will consume *all* the `a`'s in the string, leaving nothing for the final `a` to match.

Nonetheless, this method is commonly seen presented as "the" method for matching regular expressions, and it is simpler and faster than many of the more sophisticated methods we shall see, so we present it here. But we stress that what it matches are not REs, but rather some restricted subset thereof.

We say that

$$\frac{s\varepsilon \in r}{s \in L(r)}$$

Bear in mind in the rules presented below that while the prefix $s_1$ and the RE $r$ are the "inputs", $s_2$, the remainder of the string to be matched, is an "output".

$$\text{Empty}\,\frac{}{s \notin \varnothing} \qquad \text{Symbol}\,\frac{a \in \Sigma}{as_2 \in a}$$

$$\text{Union}\,\frac{s_1s_2 \in r_1 \lor s_1s_2 \in r_2}{s_1s_2 \in r_1 + r_2} \qquad \text{Cat}\,\frac{s_1s_2 \in r_1 \quad s_2s_3 \in r_2}{s_1s_3 \in r_1 \cdot r_2}$$

$$\text{Star-}\varepsilon\,\frac{}{\varepsilon s_2 \in r^*} \qquad \text{Star}\,\frac{s_1s_2 \in r \quad s_2s_3 \in r^*}{s_1s_3 \in r^*}$$

## *A continuation-based approach to matching regular expressions*

Although the pure prefix matching method presented in the previous section is incorrect, we can see the germ of a good idea in it. The problem is that we do not retain enough information about what the remainder of the RE needs to match; when matching the Kleene star in `a*a` we have no knowledge of the following `a`.

We can encode the knowledge of the rest of the RE in a *continuation*. A continuation is simply a function that tells us "what to do next". In a continuation-passing style, a normal function does not return a value directly, but rather passes it to its continuation, which it receives as an argument, and then returns the result of the continuation.

We will use the notation $s; k \in r$ to mean that $r$ matches $s$ with continuation $k$. We say that

$$\frac{s; (\lambda s'.s' = \varepsilon) \in r}{s \in r}$$

(As shown, our continuations are $\lambda$ abstractions over propositions. The notation $k\ s$ means that $k$ is true of string $s$.)

$$\text{Empty}\,\frac{}{s; k \notin \varnothing} \qquad \text{Symbol-}\varepsilon\,\frac{a \in \Sigma}{\varepsilon; k \notin a} \qquad \text{Symbol}\,\frac{a \in \Sigma \quad k\ s}{as; k \in a}$$

$$\text{Union}\,\frac{s; k \in r_1 \lor s; k \in r_2}{s; k \in r_1 + r_2} \qquad \text{Cat}\,\frac{s; (\lambda s'.s'; k \in r_2) \in r_1}{s; k \in r_1 \cdot r_2}$$

$$\text{Star-}\varepsilon\,\frac{k\ s}{s; k \in r^*} \qquad \text{Star}\,\frac{s; (\lambda s'.s \neq s' \land s'; k \in r^*) \in r}{s; k \in r^*}$$

## *Matching by enumeration*

Haskell gives us the ability to manipulate infinite data structures. Can we implement a RE matching algorithm that works simply by searching through an infinite list? As it turns out, the answer is Yes, although this will turn out to be one of the most inefficient methods of RE matching we will examine.

In order to search through an infinite list, we must know when to stop looking. The obvious case is when we find the string in question, but, of course, the string may not exist in $L(r)$. In that case, we would ideally like to avoid continuing our search on to the "end" of an infinite list! In order to give ourselves an exit, we will work with *length-ordered* lists. The items in our lists will be sorted by length, so that all strings of length $n$ appear before any strings of length $n+1$. Thus, we continue our search for the string $s$ until we see a string which is longer than $s$. At this point, no strings of length $|s|$ will occur later in the list, and we can safely reject the string.

We will use the Haskell notation for list construction: the empty list is [] while the list with head $e$ and tail $t$ is $e : t$. We define membership in a length-ordered list as

$$\frac{}{e \notin []} \qquad \frac{|e| < |e'|}{e \notin (e' : s)} \qquad \frac{}{e \in (e : t)} \qquad \frac{e \neq e' \quad e \in t}{e \in (e' : t)}$$

We then define the function LOL($r$) to be the *length-ordered language* of $r$ and say that

$$\frac{s \in \mathrm{LOL}(r)}{s \in L(r)}$$

where LOL($r$) is defined by the following rules:

$$\text{Empty}\frac{}{\mathrm{LOL}(\varnothing) = []} \qquad \text{Symbol}\frac{a \in \Sigma}{\mathrm{LOL}(a) = [a]}$$

To construct the union of two length ordered lists, we *merge* them, using string length as our comparison key:

$$\frac{}{[] \cup l_2 = l_2} \qquad \frac{}{l_1 \cup [] = l_1}$$

$$\text{LT}\frac{|a| < |b|}{(a : t_1) \cup (b : t_2) = a : (t_1 \cup (b : t_2))}$$

$$\text{EQ}\frac{|a| = |b|}{(a : t_1) \cup (b : t_2) = a : b : (t_1 \cup t_2)}$$

$$\text{GT}\frac{|a| > |b|}{(a : t_1) \cup (b : t_2) = b : ((a : t_1) \cup t_2)}$$

To do a length-ordered concatenation, we decompose the result of $(a : t_1)\cdot(b : t_2)$ into $(a \cdot b) : (([a] \cdot t_2) \cup (t_1 \cdot (b : t_2)))$:

$$\frac{}{[] \cdot l_2 = []} \qquad \frac{}{l_1 \cdot [] = []}$$

$$\frac{}{(a : t_1) \cdot (b : t_2) = (a \cdot b) : (([a] \cdot t_2) \cup (t_1 \cdot (b : t_2)))}$$

Constructing a length-ordered Kleene start follows from the recursive definition: $L^* = \{\varepsilon\} \cup L \cdot L^*$.

$$\frac{}{(\varepsilon : t)^* = t^*} \qquad \frac{a \neq \varepsilon}{(a : t) = \varepsilon : ((a : t) \cdot (a : t)^*)}$$

Note that the $\varepsilon$ case here is *not* a base case. In fact, the Kleene star rule has no base case! However, due to the fact that $L^*$ is self-recursive and relies on $\cdot$ for its implementation, if $\varepsilon \in L(r)$ then a naive implementation would generate an infinite list of $\varepsilon$. While this does not contradict our requirement that the lists be ordered by length, it is not particularly useful, either. Singling out $\varepsilon$ for special treatment allows the remainder of the list to be processed.

## *Matching regular expressions using derivatives*

Here we build a more straightforward method for matching regular expressions, that relies on a clever trick: taking the *derivative* of a regular expression!

Although you are probably familiar with the derivative from calculus, as the derivative of $\mathbb{R}$-valued functions, we can in fact define the derivative of other structures. The only hard-and-fast requirement is that the derivative operator $D$ respect the *product rule*:

$$D_v(X \cdot Y) = X \cdot D_v(Y) + D_v(X) \cdot Y$$

This applies even when the operators + and · are not "traditional" addition and multiplication. (In our case, they are union and concatenation).

We define the derivative of a regular expression $r$ with respect to a symbol $c$ to be the result of "pre-truncating" $c$ from every string in $L(r)$. That is

$$D_c(r) = \{t \mid ct \in L(r)\}$$

(Note that if a string $u \in L(r)$ does *not* start with $c$ then $u$ will not be present in $D_c(r)$ at all. This fact will be important.)

That is the *definition* of the derivative of a regular expression; it remains to *find* it, given a particular regular expression. In order to help with this process, we define the *nullability* function, which returns $\{\varepsilon\}$ if a regular expression can accept the empty string, and $\varnothing$ if it cannot:

$$\delta(r) = \begin{cases} \{\varepsilon\} & \text{if } \varepsilon \in L(r) \\ \varnothing & \text{if } \varepsilon \notin L(r) \end{cases}$$

$\delta(r)$ can be defined quite easily from the recursive definition of nullability giving above. (And recall that $\{\varepsilon\} = \varnothing^*$.)

Given $\delta(r)$, we define the derivative operator $D_c(r)$ by structural recursion on $r$:

$$\begin{aligned}
D_c(\varnothing) &= \varnothing && \text{(case } \varnothing\text{)} \\
D_c(\{\varepsilon\}) &= \varnothing && \text{(case 0-length string)} \\
D_c(c) &= \{\varepsilon\} && \text{(symbol case } c\text{)} \\
D_c(c') &= \varnothing && \text{((}c \neq c'\text{))} \\
D_c(R^*) &= D_c(R)R^* && \text{(Kleene star case)} \\
D_c(R_1R_2) &= D_c(R_1)R_2 + \delta(R_1)R_1D_c(R_2) && \text{(concatenation case)} \\
D_c(R_1 + R_2) &= D_c(R_1) + D_c(R_2) && \text{(union case)}
\end{aligned}$$

Given the derivative, how can we use it to match against a string $s$? Suppose $s = c$, a single symbol. In this case, we take the derivative with respect to $c$: if $c$ is accepted then the derivative will be $\{\varepsilon\}$, if it is rejected it will be $\varnothing$. But suppose our string is *two* symbols long, $c_1 c_2$. If we take the derivative with respect to $c_1$ then the result will either be $\varnothing$ (if *no* strings starting with $c_1$ are accepted) or some non-empty regular expression $r'$. We can then take the derivative of $r'$ with respect to $c_2$

(This method, due to Brzozowski, is more than 50 years old at this point, but was only recently re-discovered and brought into wider knowledge.)

## *Deterministic Finite Automata*

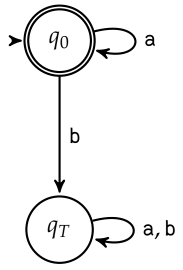
*Deterministic finite automata* (FSM) are the computational manifestations of regular languages. Although we will prove this rigarously later, to begin with we present the mathematical definition of a FSM, followed by several examples, some familiar, some less so.

*Finite state machines:* A finite state machine is a 4-tuple $M = (Q, q_0, F, \delta)$ with the following components:

Some definitions make a FSM into a 5-tuple, adding the alphabet $\Sigma$ to the definition; we leave the alphabet as implicit.

| | |
|---|---|
| $Q$ | (The set of states) |
| $q_0 \in Q$ | (An identified state, the *start* state) |
| $F \subseteq Q$ | (The set of *final* states) |
| $\delta : Q \times \Sigma \to \mathscr{P}(Q)$ | (The *transition function*) |

Since this kind of definition is rather cumbersome to visualize, we will usually prefer to present machines as labeled directed graphs:



This machine accepts the langauge $\mathtt{a}^*$. Its formal definition is

$$\begin{aligned} Q &= \{q_0, q_T\} \\ q_0 &= q_0 \\ F &= \{q_0\} \\ \\ \delta(q_0, \mathtt{a}) &= q_0 \\ \delta(q_0, \mathtt{b}) &= q_T \\ \delta(q_T, \mathtt{a}) &= q_T \\ \delta(q_T, \mathtt{b}) &= q_T \end{aligned}$$

Note that we illustrate the start state with an arrow, and final states with double-circles. Arcs between states are labeled with the symbol(s) that will trigger the corresponding transitions.

We generally refer to a state such as $q_T$ as a *trap state*. Its sole purpose is to consume the remainder of the string after an invalid symbol has been seen. In this case, the only valid symbols are a's, so if we see a b we know there is nothing else that can happen in the remainder of the string that will cause it to be accepted. Thus, the trap state is not final, and the trap state has transitions $\forall a \in \Sigma : \delta(q_T, a) = q_T$. But note that the trap state is not "special" or designated in anyway. It is entirely possible for a machine to have *multiple* states which act as traps, although in machines we construct by hand, this will not occur.

For a *deterministic* finite state machine, it will always be the case that $\forall q \in Q, a \in \Sigma : |\delta(q, a)| = 1$, i.e., the delta function will map a pair $(q, a)$ to a *single* state. For non-deterministic machines we may have $|\delta(q, a)| \geq 1$, and some definitions allow machines with $|\delta(q, a)| = 0$ with the assumption that for any $q, a$ such that $|\delta(q, a)| = 0$, $\delta(q, a) = q_T$.

## *The language of a DFA*

For any deterministic machine, we informally define *acceptance of a string* as whether, after "feeding" the string into the transition function, starting at $q_0$, is the resulting state a final state? For example, the string aaa is accepted by the above machine because

$$\begin{array}{l} \delta(q_0, \mathtt{a}) = q_0 \\ \qquad \delta(q_0, \mathtt{a}) = q_0 \\ \qquad\qquad \delta(q_0, \mathtt{a}) = q_0 \\ \qquad\qquad\qquad q_0 \in F \end{array}$$

We define the language of a DFA, $L(M)$ to be the set of strings that it accepts. We can give two formal definitions of acceptance:

*The $\delta^*$ function:* We can extend $\delta$ to operate not just on individual characters, but entire strings:

$$\delta^*\text{-}\varepsilon\,\frac{}{\delta^*(q,\varepsilon) = q} \qquad\qquad \delta^*\text{-}as\,\frac{\delta^*(q,a) = q' \qquad \delta^*(q',s) = q''}{\delta^*(q,as) = q''}$$

We then define acceptance as

$$s \in L(M) \quad \text{iff} \quad \delta^*(q_0, s) \in F$$

This definition is recursive in the length of the string. The next definition is recursive in the length of the *path* traced through a machine during processing of a string (of course, because each transition is triggered by a character from the string, the length of the path will be equal to the length of the string).

*The modified machine language $L_q(M)$:* Here we define an alternate version of $L(M)$, $L_q(M)$, the language of *M started in state q*. Clearly, $L_{q_0}(M) = L(M)$, so if we can define $L_q(M)$ we will have an easy definition for $L(M)$. We define $L_q(M)$ as

$$L_q\text{-}F\,\frac{q \in F}{\varepsilon \in L_q(M)} \qquad\qquad L_q\text{-}q\,\frac{\delta(q,a) = q' \qquad s \in L_{q'}(M)}{as \in L_q(M)}$$

and again, we define acceptance as

$$s \in L(M) \quad \text{iff} \quad s \in L_{q_0}(M)$$

*Proof that these methods are equivalent:* Our first attempt would likely be to prove this by induction on the length of $s$, however, a naive approach will fail. The problem is that if we state our theorem as

$$\forall s \in \Sigma^* : \delta^*(q_0, s) \in F \quad \text{iff} \quad s \in L_{q_0}(M)$$

then we will get stuck as soon as we apply a single transition. We will no longer be in state $q_0$, and thus will be unable to apply the IH. Instead, we quantify over both all strings *and all states*:

$$\forall s \in \Sigma^* : \forall q \in Q : \delta^*(q, s) \in F \quad \text{iff} \quad s \in L_q(M)$$

PROOF. The proof is then straightforward by induction on $s$: Case $\varepsilon$: $s = \varepsilon$. Then by rules $\delta^*$-$\varepsilon$ and $L_q$-$F$ we have $\delta^*(q,\varepsilon) \in F$ iff $\varepsilon \in L_q(M)$.

Case CONS: $s = as'$ for some $s'$.

$$\begin{aligned}
\delta^*(q, as') \in F \quad &\text{iff} \quad \delta(q,a) = q' \wedge \delta^*(q', s') \in F && \text{(By rule } \delta^*\text{-}as\text{)} \\
&\text{iff} \quad \delta(q,a) = q' \wedge s' \in L_{q'}(M) && \text{(By IH)} \\
&\text{iff} \quad as' \in L_q(M) && \text{(By rule } L_q\text{-}q\text{)}
\end{aligned}$$

The proof is completed by letting $q = q_0$. ⊣

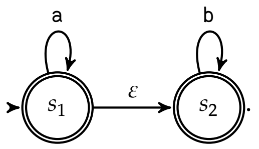
## *Nondeterministic Finite Automata*

### *ε-transitions*

We've seen how DFAs can be used to recognize and generate strings, and now we want to consider the question of whether, and if so, how DFAs can be *combined* to form larger machines. We'll see later on that normal DFAs can in fact be combined to form the union, concatenation, etc. of their languages (and, indeed, this result will be a key part of our proof that DFAs recognize accept exactly the regular languages) but performing these operations on standard DFAs is somewhat tricky and involved. In this section, we'll introduce an extension to DFAs called NDA-$\varepsilon$, DFAs with a touch of *non-determinism*

"Non-determinism" is a computer science term meaning "magic".

A NDA-$\varepsilon$ is a DFA in which transitions can be labeled with an element of $\Sigma \cup \{\varepsilon\}$. That is, we now allow transitions that "recognize" the empty string! For example, here is a NDA-$\varepsilon$ that recognizes the language `a`$^*$`b`$^*$:



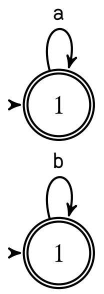
This looks crazy; how does the machine know when to follow the $\varepsilon$ arc? In fact, we redefine acceptance for NDA-$\varepsilon$ machines as follows

Definition 0.7 An NDA-$\varepsilon$ accepts a string if there is *any* execution of the machine in which the machine terminates in a final state with the entire string consumed.

That is, a NDA-$\varepsilon$ *magically* knows when to take an $\varepsilon$-transition, so that if there is any possible way of ending in a final state, it will do so. Later, we'll see how this can be implemented on normal, non-magical computers, and indeed how every NDA-$\varepsilon$ can be transformed into an equivalent DFA. Right now, whether $\varepsilon$-transitions are *useful*; we'll worry about how to actually make them work later.

### *Combining DFAs with ε-transitions*

Here are the DFAs that accept `a`$^*$ and `b`$^*$:

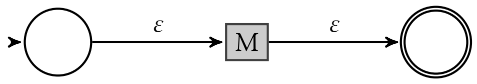


If we look at the above machine, it would appear that all we have to do to construct the concatenation of two machines is put an $\varepsilon$-transition between them. In fact, that's almost the case. For what follows, we require our machines to have a certain structure:

- The machine's start state must have a *single* transition from it, and this must be an $\varepsilon$-transition.

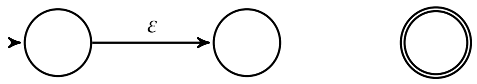
- The machine must have a single final state, and this state must have a single $\varepsilon$-transition into it.

Any NFA can be trivially transformed into one that meets these criteria. If the start state is not suitable then we simply add a new start state, with a single $\varepsilon$-transition to the original start state. Likewise, if the final state(s) are not suitable, we simply add a new final state, and add $\varepsilon$-transitions from every existing final state (which will no longer be final) to the new final state. Thus, every machine is of the form:



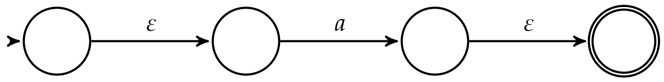
Given this assumption, we can now proceed to construct machines for each of the "constructors" of regular sets:

*Empty set:* The machine that accepts $\varnothing$ in $\varepsilon$-form is simply

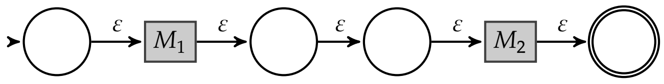


With no way to reach the final state, no string is ever accepted.

*Single symbol a:* To build a machine which only accepts $\{a\}$ we use four states, including the two "wrapper" states: one before we have recognized $a$, and one after:

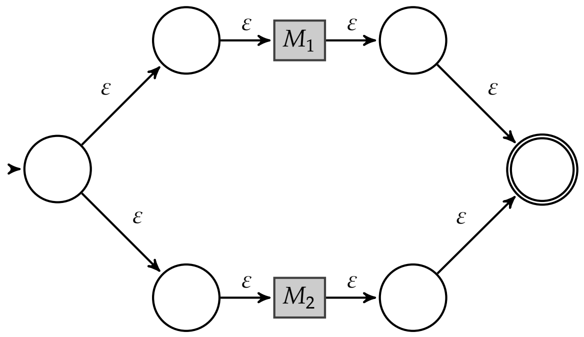


*Concatenation (of machines $M_1$ and $M_2$):* To construct the concatenation of two machines, we simply string them together with an $\varepsilon$ transition in the middle. That is,

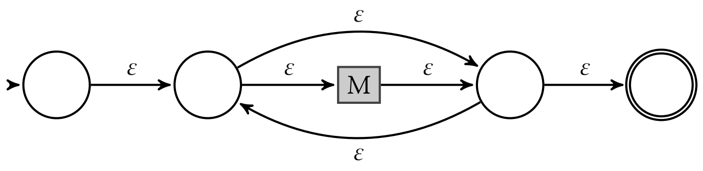


(Note that we remove the final state on machine $M_1$.)

*Union (of machines $M_1$ and $M_2$):* To build the union, we add a new start state which *branches* to both machines with $\varepsilon$ transitions. Likewise, the (formerly) final states of both machines now join to a single new final state:



*Kleene Star:* To form the Kleene start of an existing machine, we "nest" it inside a new pair of initial and final states (because we are going to add arcs to the original states) and then add $\varepsilon$ transitions that allow us to repeat the body of the machine as much as necessary:



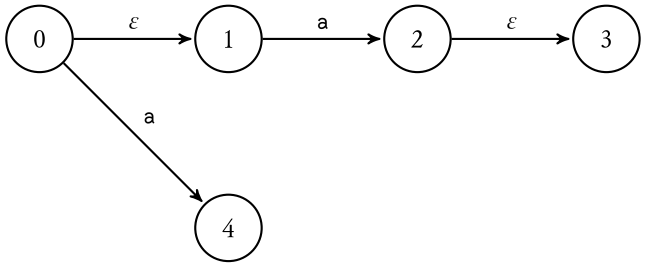
## *Execution of a* NDA-$\varepsilon$

We now come to the problem of actually "running" a NDA-$\varepsilon$. Here we consider executing a NDA-$\varepsilon$ on a string directly; later we will consider the problem of converting a NDA-$\varepsilon$ to a DFA.

From a given state $q$, in a NDA-$\varepsilon$, for any $a \in \Sigma$, we now have two possible ways of transitioning to another state:

- Directly from $q$, with $\delta(q,a) = q'$
- Indirectly, by following some chain of $\varepsilon$-transitions from $q$ to some $q_i$, and then following $\delta(q_i,a) = q_j$ and finally following yet another chain of $\varepsilon$-transitions to reach a state $q'$.

The following machine illustrates both kinds of transitions:



From node 1, by reading an `a`, we can reach nodes 3 and 4.

In order to capture the notion of "all states reachable from $q$ by $\varepsilon$-transitions" we define the $\varepsilon$ *closure function*. The $\varepsilon$ closure of a state is all the states that are reachable from it *without* consuming any input. (Thus, a state is always in its own $\varepsilon$ closure.) $\varepsilon$-closure is defined inductively as

$$\text{Base case}\ \frac{}{q \in \varepsilon\text{-closure}(q)} \qquad \text{Inductive case}\ \frac{q' \in \varepsilon\text{-closure}(q) \qquad q'' \in \delta(q', \varepsilon)}{q'' \in \varepsilon\text{-closure}(q)}$$

That is, $q$ is always in its own $\varepsilon$ closure, and if $q'$ is in the $\varepsilon$ closure of $q$, and there is an $\varepsilon$ transition from $q'$ to $q''$, then $q''$ is in the $\varepsilon$ closure of $q$ as well.

We can use $\varepsilon$-closure to define the extended transition function $\delta_\varepsilon(q, a)$ which gives us the *set* of states that can be reached from $q$ by reading $a$:

$$\delta_\varepsilon(q, a) = \bigcup_{q' \in \varepsilon\text{-closure}(q)} \varepsilon\text{-closure}(\delta(q', a))$$

We first find the $\varepsilon$ closure of $q$. Then, for every state in the closure, we perform a normal transition on $a$, giving us a new set of states. From each of these states, we take a further $\varepsilon$ transition; the union of all these final $\varepsilon$ transitions is the full set of states reachable from $q$ by reading an $a$.

When simulating an $\varepsilon$ machine (or, more generally, a NDA), our register will not be a pair of (state, string) but rather a pair of a *set of states* and a string. We can define $\delta^*_\varepsilon$ based on $\delta_\varepsilon$; the difference is that we must now "run" the transition function on *every* set in the current set of states.

$$\frac{}{\delta^*_\varepsilon(X, \varepsilon) = \{X\}} \qquad \frac{X' = \{q' \mid q \in X, q' \in \delta_\varepsilon(q, a)\} \qquad \delta^*_\varepsilon(X', s) = X''}{\delta^*_\varepsilon(X, as) = X''}$$

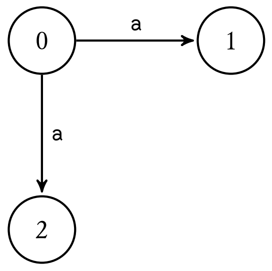
We can then use $\delta^*_\varepsilon$ to simulate a $\varepsilon$ machine just as we would use $\delta^*$ to simulate a DFA. Acceptance is defined as the set of states containing *any* final states at the end of the string; that is,

$$s \in L_\varepsilon(M) \quad \text{iff} \quad \delta^*_\varepsilon(\{q_0\}, s) \cap F \neq \varnothing$$

(But note that we can reject a string early if, at any point in the simulation, $X = \varnothing$.)

## *Non-Deterministic Finite Automata*

NDA-$\varepsilon$ machines are in fact just an instance of *non-deterministic* finite automata. An NDA is an DFA in which multiple arcs with the same label are allowed out of a state:



Given the preceding discussion of $\varepsilon$-transitions, it should now be obvious how this works: we redefine acceptance so that a string is accepted if there is *any* sequence of states that leads to a final state, and when executing a NDA we simply try *all* the possible transitions.

### *Execution of a* NDA

Almost everything said above with respect to the execution of a NDA-$\varepsilon$ applies here as well, the only difference is that our transition function now outputs sets of states:

$$\delta(0, a) = \{1, 2\}$$

(for the above machine).

(details to follow)

## *Regular Language Representations*

We've seen several "representations" of languages: regular expressions, regular grammars, and the languages defined by DFAs and NDAs. Here we will consider conversions between all of these representations. In particular, by showing that every NDA can be converted to an equivalent DFA, we will prove that non-determinism does *not* offer any additional computational power, and by showing that there is a bijection between DFAs and REs we will show that DFAs (and hence, NDAs) accept exactly the regular languages. (In fact, we have been dealing with regular languages all along, just in various forms.)

### *Regular Grammar to* NDA

We will assume, without loss of generality, that every rule in the grammar is of the forms

$$A \to \mathtt{a}B \quad \text{or} \quad A \to \varepsilon$$

If this is not the case, note that a rule $A \to \mathtt{a}$ can be rewritten into a pair of rules

$$\begin{aligned} A &\to \mathtt{a}T \\ T &\to \varepsilon \end{aligned}$$

Then the conversion to NDA is straightforward: every nonterminal is a state, the start symbol is the start state, every rule of the form $A \to \mathtt{a}B$ indicates a transition from state $A$ to $B$ accepting $\mathtt{a}$, and every rule of the form $A \to \varepsilon$ indicates that $A$ is a final state. More formally:

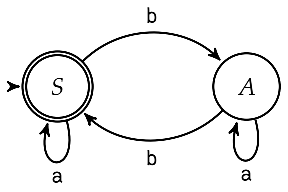
$$\begin{aligned} Q &= V \\ q_0 &= S \\ F &= \{A \mid (A \to \varepsilon) \in R\} \end{aligned}$$

$$\delta(q, a) = q' \quad \text{when} \quad (q \to aq') \in R$$

As an example, the grammar

$$S \rightarrow \mathsf{a}S \mid \mathsf{b}A \mid \varepsilon$$
$$A \rightarrow \mathsf{a}A \mid \mathsf{b}S$$

becomes the two-state machine



Note that this procedure is entirely reversible: we can go from a NDA to a regular grammar with equal ease.

## *Regular Expression to* NDA-$\varepsilon$

We've seen most of this conversion above. Here we will just present the results.
(details to follow)

### NDA-$\varepsilon$ *to* NDA

Here we are interested in removing $\varepsilon$ transitions from a machine. The result will be a NDA. This is the first step in the process of converting a NDA $-\ \varepsilon$ to a DFA.

We will rely on the function defined in section . The intuitive idea is that we will replace a chain of arcs

$$q_0 \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} q_i \xrightarrow{a} q_j \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} q_n$$

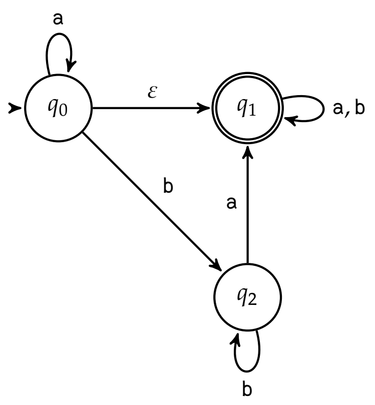
with a single arc

$$q_0 \xrightarrow{a} q_n$$

In order to do this, we need to find $\varepsilon$-closure($q_0$) and then, from every node in $\varepsilon$-closure($q_0$) that has an arc labeled $a$, find the connected nodes, and then again take the $\varepsilon$-closure of all of them. Formally, we define the $\varepsilon$-transition function $\delta_\varepsilon$ as

$$\delta_\varepsilon(q,a) = \bigcup_{q' \in \varepsilon\text{-closure}(q)} \varepsilon\text{-closure}(\delta(q',a))$$

For example, consider the machine



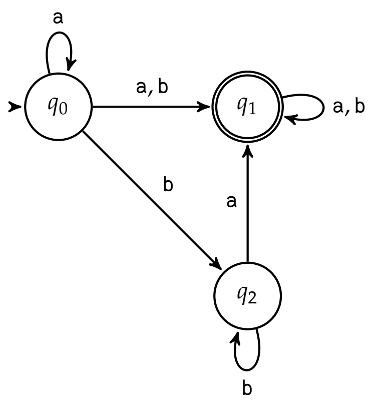
The $\varepsilon$-closure of each of the states is

$$\begin{aligned}
\varepsilon\text{-closure}(q_0) &= \{q_0, q_1\}\\
\varepsilon\text{-closure}(q_1) &= \{q_1\}\\
\varepsilon\text{-closure}(q_2) &= \{q_2\}
\end{aligned}$$

To construct $\delta_\varepsilon(q, a)$ we simply look at $q$ and find all the other nodes that could be reached by following some (possible empty) sequence of $\varepsilon$-transitions, follow by a literal $a$-transition, followed by another (possibly empty) sequence of $\varepsilon$-transitions:

$$\begin{aligned}
\delta_\varepsilon(q_0, \mathsf{a}) &= \{q_0, q_1\}\\
\delta_\varepsilon(q_0, \mathsf{b}) &= \{q_1, q_2\}\\
\delta_\varepsilon(q_1, \mathsf{a}) &= \{q_1\}\\
\delta_\varepsilon(q_1, \mathsf{b}) &= \{q_1\}\\
\delta_\varepsilon(q_2, \mathsf{a}) &= \{q_1\}\\
\delta_\varepsilon(q_2, \mathsf{b}) &= \{q_2\}
\end{aligned}$$

Which gives us the $\varepsilon$-free machine



## NDA *to* DFA

Now we consider the problem of removing nondeterminism completely. We will continue with the $\varepsilon$-free machine from the previous section. (The method described in this section assumes that the input machine is $\varepsilon$-free.)

Here we will think of the execution of a non-deterministic machine as running in multiple “threads”, one for each state the machine might be in at a given point in the input string. For example, in the machine above, from state $q_0$ and reading an $a$, we can be in state $q_0$ or $q_1$, so we will assume that we are in *both*. The states in our deterministic machine will be *sets of states* from the original machine.

$$\begin{aligned} Q_D &= \mathscr{P}(Q) \\ q_0 &= \{q_0\} \\ F_D &= \{X \mid X \in Q_D, X \cap F \neq \varnothing\} \\ \delta_D(X,a) &= \{\delta(q,a) \mid q \in X\} \end{aligned}$$

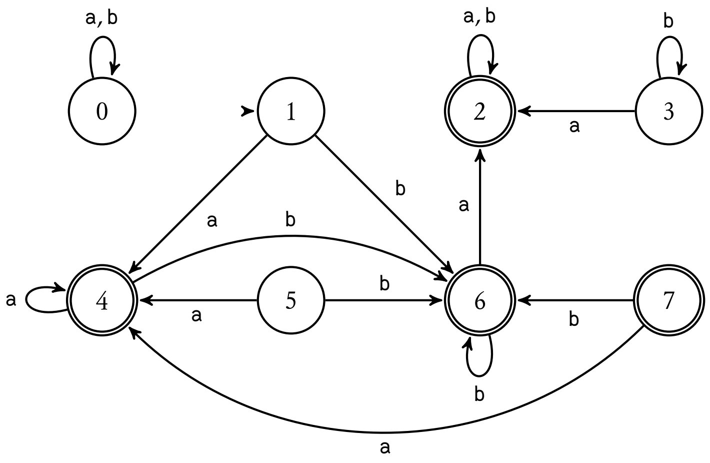
The start state of the deterministic machine is simply the set of just the start state of the original machine. The final states of the deterministic machine are any sets which contain a final state of the original machine. The transition function simply runs the original transition function on all states in the set simultaneously.

For the example machine above (with the states numbered for easy of reference) we have

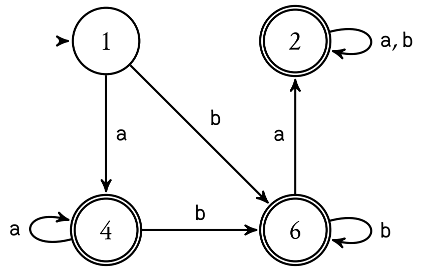
$$\begin{aligned} Q_D &= \{\underbrace{\varnothing}_{0}, \underbrace{\{q_0\}}_{1}, \underbrace{\{q_1\}}_{2}, \underbrace{\{q_2\}}_{3}, \underbrace{\{q_0,q_1\}}_{4}, \underbrace{\{q_0,q_2\}}_{5}, \underbrace{\{q_1,q_2\}}_{6}, \underbrace{\{q_0,q_1,q_2\}}_{7}\} \\ q_0 &= \{1\} \\ F_D &= \{2,4,6,7\} \end{aligned}$$

$$\begin{array}{ll} \delta_D(0,\mathsf{a}) = 0 & \delta_D(0,\mathsf{b}) = 0 \\ \delta_D(1,\mathsf{a}) = 1 & \delta_D(1,\mathsf{b}) = 6 \\ \delta_D(2,\mathsf{a}) = 2 & \delta_D(2,\mathsf{b}) = 2 \\ \delta_D(3,\mathsf{a}) = 2 & \delta_D(3,\mathsf{b}) = 3 \\ \delta_D(4,\mathsf{a}) = 4 & \delta_D(4,\mathsf{b}) = 6 \\ \delta_D(5,\mathsf{a}) = 4 & \delta_D(5,\mathsf{b}) = 6 \\ \delta_D(6,\mathsf{a}) = 2 & \delta_D(6,\mathsf{b}) = 6 \\ \delta_D(7,\mathsf{a}) = 4 & \delta_D(7,\mathsf{b}) = 6 \end{array}$$

Giving us the machine:



Note that states 0, 3, 5 and 7 are unreachable. After their removal we have

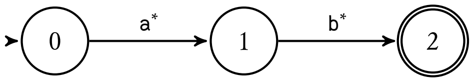


Thus completing the transformation from NDA-$\varepsilon$ to DFA.

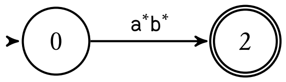
## NDA/DFA *to Regular Expression (direct)*

We will examine a method for converting a FSM (DFA or NDA) directly to a RE. Incidentally, this method, together with any of the methods for constructing a DFA from a REG, completes the proof that deterministic finite automata accept exactly the regular languages.

Our method works by converting the graph of a FSM to an *expression graph*. An expression graph is, like a DFA/NDA, a directed graph with a distinguished initial node and a set of final nodes. Here, however, the arcs between nodes are labeled with regular expressions. For example:



This expression graph accepts the language $a^*b^*$. Since edges can be labeled with arbitrary regular expressions, we can simplify this graph by *deleting* state 1, replacing it with a single arc from 0 to 1, labeled with the concatenation of $a^*$ and $b^*$:



Our general algorithm will thus be to choose an arbitrary non-initial, non-final state to delete, examine its inbound and outbound arcs, and combine them into arcs *directly* connecting the now-adjacent nodes (i.e., bypassing the deleted node). Once the deleted node has been fully disconnected, it can be removed from the graph. Eventually, we will have a graph in one of two forms

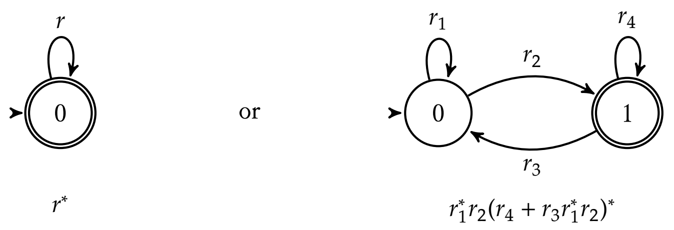


Figure 5: Results of the node deletion algorithm

The first represents the regular expression $r^*$ while the second represents $r_1^* r_2 (r_3 + r_4 r_1^* r_2)^*$. (Although this might appear complex, in practice the regular expression identities in figure 4 can often be used to significantly simplify the resulting expression.)

The full algorithm is presented in algorithm 1.

**Algorithm 1** Node deletion algorithm

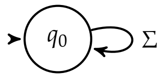
```
 1: For each $f \in F$ make a copy of $M$: $M_f$, with a single final state $f$
 2: for all $M_f$ do
 3:     for all non-initial, non-final nodes $q$ do
 4:         Choose nodes $q_{-1}$ and $q_{+1}$ with $q_{-1} \neq q$ and $q_{+1} \neq q$ such that $\delta(q_{-1}, -) = q$ and $\delta(q, -) = q_{+1}$.
 5:         for all pairs of arcs $q_{-1} \xrightarrow{r_1} q$ and $q \xrightarrow{r_2} q_{+1}$ do
 6:             if $\delta(q, r) = q$ for some $r$ then
 7:                 Add arc $q_{-1} \xrightarrow{r_1 r^* r_2} q_{+1}$
 8:             else
 9:                 Add arc $q_{-1} \xrightarrow{r_1 r_2} q_{+1}$
10:             end if
11:             Remove all arcs $q_{-1} \xrightarrow{r_{1,2,\ldots}} q_{+1}$ and replace with a single arc labeled $q_{-1} \xrightarrow{r_1 + r_2 + \ldots} q_{+1}$
12:         end for
13:         Delete node $q$ from $M_f$, along with all arcs into/out of it.
14:     end for
15:     The RE corresponding to $M_f$, $R(M_f)$ is is given by the machines in figure 5.
16: end for
17: The final RE is $R(M_{f_1}) + R(M_{f_2}) + \ldots$
```

## *Regular Expression to* DFA *(direct)*

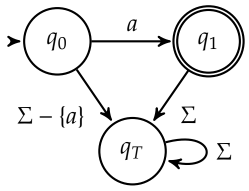
We mentioned above that it is possible, but cumbersome, to directly construct the union and concatenation of a pair of DFAs, and the Kleene star of a single DFA. Since this construction gives us a *directly* translation from RE to DFA it serves to emphasize that all regular languages can be accepted by a DFA. But note that the machines constructed by this method tend to be quite large, so some kind of simplification is needed.

In order to transform a REG into a DFA, we will map each of the allow constructors for REG — $\varnothing, a, \cup, \cdot, *$ — into a corresponding machine. The constructions will be increasingly complex.

*$M_\varnothing$, the machine that accepts $\varnothing$:* This machine will have one state, which is not final. It can be thought of as a machine which is nothing but a trap state:



*$M_a$, the machine that accepts a single symbol $a$:* This machine will have three states: the start state must not be final (because $\varepsilon$ is not accepted), there must be a final state after $a$ has been accepted, and there must be a trap state to reject all other strings:



*$M_\cup$, the Union machine:* To construct a machine that accepts the *union* of the languages of two other machines (i.e., $M$ such that $L(M) = L(M_1) \cup L(M_2)$) requires some cleverness. We are going to build a machine that essentially runs *both* its constituent machines together, in lockstep. If, at the end of the string, either machine is in a final state, then the string is accepted.

(For clarity in the following discussion, elements of $M_1$ will be colored in blue, while elements of $M_2$ will be red.)

In order to run both machines simultaneously, we will represents states in $M$ as *pairs* of states from $M_1$ and $M_2$:

$$Q = \{(q, q) \mid q \in Q_1, q \in Q_2\}$$

The start state will be the pair of both machines' start states:

$$q_0 = (q_0, q_0)$$

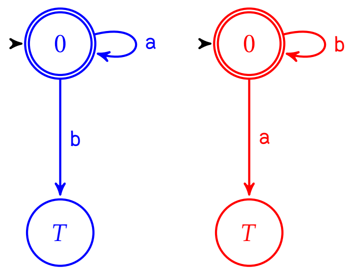
while the final states will consist of those pairs in which *either* component is final:

$$F = \{(q, q) \mid q \in Q_1, q \in Q_2, q \in F_1 \vee q \in F_2\}$$

The transition function will, for any input character $a$, simply run $\delta_1$ and $\delta_2$ on the respective components of the current state:

$$\delta((q, q), a) = (\delta_1(q, a), \delta_2(q, a))$$

As an example, consider constructing the union of the two machines



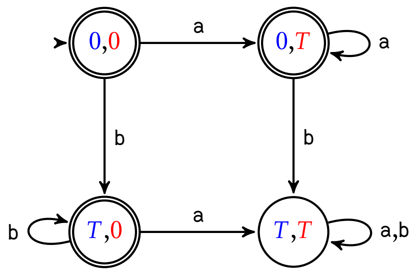
Each our these machines has 2 states, so the union of them will have $2 \times 2 =$ 4 states:

$$Q = \{(0,0),(0,T),(T,0),(T,T)\}$$

Of these, all but one – $(T,T)$ – is final. The transition function is given by

$$\delta((0,0),a) = (0,T)$$
$$\delta((0,0),b) = (T,0)$$
$$\delta((0,T),a) = (0,T)$$
$$\delta((0,T),b) = (T,T)$$
$$\delta((T,0),a) = (T,T)$$
$$\delta((T,0),b) = (T,0)$$
$$\delta((T,T),a) = (T,T)$$
$$\delta((T,T),b) = (T,T)$$

The resulting machine is



As an aside, note that we can use a similar construct to build an *Intersection* machine: we simply change the criteria for final states so that $(q,q)$ is final iff $q \in F \wedge q \in F$. I.e.,

$$F = \{(q,q) \mid (q,q) \in Q, q \in F \wedge q \in F\}$$

This serves as a proof that REG is closed under intersection. A similar trick can be used to build machines that accept the symmetric or set difference of two machines' languages, thus proving that REG is closed under those operations as well.

An alternate proof relies on the proof that REG is closed under *complementation* via the identity $L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$.

*M·, the Cat machine:* In order to construct a single machine which accepts the concatenation of two machines' languages, we have to somehow run one machine until it finishes on some prefix of the string, and then start the second machine at that point in the string and run it to its completion. The catch is that the first machine may finish (reach a final state) at many different positions in the string, and we have no way of knowing in advance which prefix will allow the second machine to complete successfully.

We've already seen from the union machine that it's possible to effectively run two machines in parallel. What we need here is the ability to run $n$ "instances" of $M_2$ in parallel with $M_1$, where $n$ is not fixed (e.g., at 1, as in the union machine) but can vary at runtime. We will do this by representing the collection of all running instances of $M_2$ as a *set* of states from $Q_2$. Because two instances of $M_2$ are indistinguishable if they are in the same state, we do not need to keep track of multiple instances per state. Thus, the upper bound on $n$ is $|\mathscr{P}(Q_2)|$. Of course, this can still be quite large; the number of states in our combined machine will be $|Q_1|(2^{|Q_2|})$.

The states of our machine will be the states of $M_1$, paired with subsets of the states of $M_2$:

$$\{(q_1, X_2) \mid q_1 \in Q_1, X_2 \in \mathscr{P}(Q_2)\}$$

However this alone is not quite correct. We need to "fork" a new instance of $M_2$ whenever $M_1$ is in a final state. Thus, if $q_1 \in F_1$, we must ensure that $q_0$ of $M_2$ is an element of $X_2$. To do this, we define the notion of "correcting" a state:

$$(q, X)^C = \begin{cases} (q, X \cup \{q_0\}) & \text{if } q \in F_1 \\ (q, X) & \text{otherwise} \end{cases}$$

The final states will consist of those states in which *any* instance of $M_2$ is in a final state.

With the correction operation specified, we can now give the specification of the Cat machine:

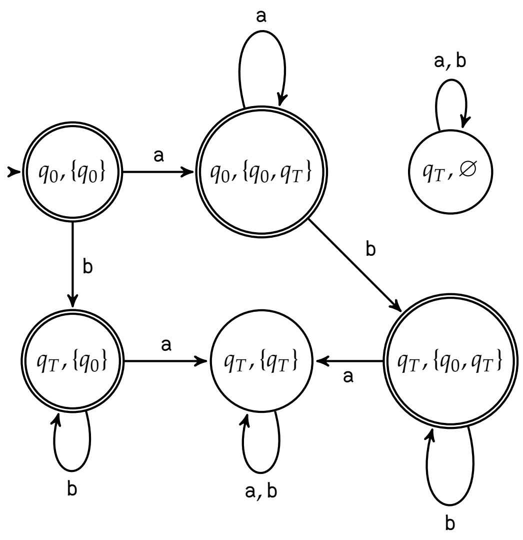
$$\begin{aligned} Q &= \{(q_1, X_2)^C \mid q_1 \in Q_1, X_2 \in \mathscr{P}(Q_2)\} \\ q_0 &= (q_0, \varnothing)^C \\ F &= \{(q, X) \mid (q, X) \in Q, X \cap F_2 \neq \varnothing\} \\ \delta((q, X), a) &= (\delta_1(q, a), \{\delta_2(x, a) \mid x \in X\})^C \end{aligned}$$

As an example, we will build the machine that accepts `a*b*` out of the component machines given above. The resulting machine will have $2(2^2) = 8$ states, but correction will remove 2 of these. Of the resulting 6 states, 4 will be final:

$$\begin{aligned} Q &= \{(q_0, \{q_0\}), (q_0, \{q_0, q_T\}), (q_T, \varnothing), (q_T, \{q_0\}), (q_T, \{q_T\}), (q_T, \{q_0, q_T\})\} \\ q_0 &= (q_0, \varnothing)^C = (q_0, \{q_0\}) \\ F &= \{(q_0, \{q_0\}), (q_0, \{q_0, q_T\}), (q_T, \{q_0\}), (q_T, \{q_0, q_T\})\} \end{aligned}$$

$$\delta((q_0,\{q_0\}),\mathsf{a}) = (q_0,\{q_0,q_T\})$$
$$\delta((q_0,\{q_0\}),\mathsf{b}) = (q_T,\{q_0\})$$
$$\delta((q_0,\{q_0,q_T\}),\mathsf{a}) = (q_0,\{q_0,q_T\})$$
$$\delta((q_0,\{q_0,q_T\}),\mathsf{a}) = (q_T,\{q_0,q_T\})$$
$$\delta((q_T,\varnothing),\mathsf{a}) = (q_T,\varnothing)$$
$$\delta((q_T,\varnothing),\mathsf{b}) = (q_T,\varnothing)$$
$$\delta((q_T,\{q_0\}),\mathsf{a}) = (q_T,\{q_T\})$$
$$\delta((q_T,\{q_0\}),\mathsf{b}) = (q_T,\{q_0\})$$
$$\delta((q_T,\{q_T\}),\mathsf{a}) = (q_T,\{q_T\})$$
$$\delta((q_T,\{q_T\}),\mathsf{b}) = (q_T,\{q_T\})$$
$$\delta((q_T,\{q_0,q_T\}),\mathsf{a}) = (q_T,\{q_T\})$$
$$\delta((q_T,\{q_0,q_T\}),\mathsf{b}) = (q_T,\{q_0,q_T\})$$

Giving us the machine



(Note that node $(q_T,\varnothing)$ is unreachable and thus can be removed without affecting the behavior of the machine.)

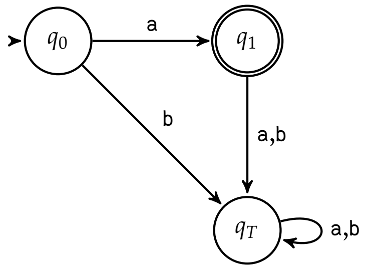
*The Star machine:* The technique for constructing $M^*$ is very similar to that for constructing $M_1 \cdot M_2$. As before, we will run multiple instances of $M$ in parallel, spawning a new instance whenever any existing instance is in a final state. Thus, if, at the end of the string, *any* instance is in a final state, this indicates that there was some repetition which was able to consume the entire string. The one caveat is that, because $\varepsilon \in L(M)^*$ our start state must be a final state. Since this may not be the case in the original machine, we will repurpose the state represented by $\varnothing$ to be the new start state. Its transitions will be identical to those of the original $q_0$.

As before, we need to "correct" our sets of states: whenever any state is final, we need to ensure that the start state is present in the set. To that purpose, we define

$$X^C = \begin{cases} X \cup \{q_0\} & \text{if } X \cap F \neq \varnothing \\ X & \text{otherwise} \end{cases}$$

$$\begin{aligned} Q_* &= \{X^C \mid X \in \mathscr{P}(Q)\} \\ q_0 &= \varnothing \\ F_* &= \{\varnothing\} \cup \{X \mid X \in Q_*, X \cap F \neq \varnothing\} \\ \delta_*(\varnothing, a) &= \{\delta(q_0, a)\}^C \\ \delta_*(X, a) &= \{q' \mid q \in X, \delta(q, a) = q'\}^C \end{aligned}$$

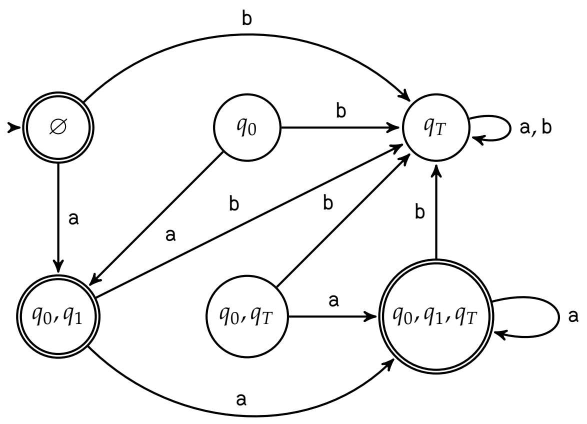
As an example, we will construct the machine recognizing a* from the machine recognizing a:



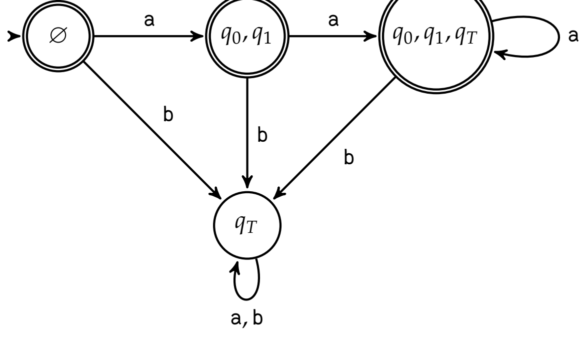
The resulting machine will have $2^3 = 8$ states, but after correction only 6 will remain, of which 4 are final.

$$\begin{aligned} Q &= \{\varnothing, \{q_0\}, \{q_T\}, \{q_0, q_1\}, \{q_0, q_T\}, \{q_0, q_1, q_T\}\} \\ q_0 &= \varnothing \\ F &= \{\varnothing, \{q_0, q_1\}, \{q_0, q_1, q_T\}\} \end{aligned}$$

$$\begin{aligned}
\delta(\varnothing, \mathsf{a}) &= \{q_0, q_1\}\\
\delta(\varnothing, \mathsf{b}) &= \{q_T\}\\
\delta(\{q_0\}, \mathsf{a}) &= \{q_0, q_1\}\\
\delta(\{q_0\}, \mathsf{b}) &= \{q_T\}\\
\delta(\{q_T\}, \mathsf{a}) &= \{q_T\}\\
\delta(\{q_T\}, \mathsf{b}) &= \{q_T\}\\
\delta(\{q_0, q_1\}, \mathsf{a}) &= \{q_0, q_1, q_T\}\\
\delta(\{q_0, q_1\}, \mathsf{b}) &= \{q_T\}\\
\delta(\{q_0, q_T\}, \mathsf{a}) &= \{q_0, q_1, q_T\}\\
\delta(\{q_0, q_T\}, \mathsf{b}) &= \{q_T\}\\
\delta(\{q_0, q_1, q_T\}, \mathsf{a}) &= \{q_0, q_1, q_T\}\\
\delta(\{q_0, q_1, q_T\}, \mathsf{b}) &= \{q_T\}
\end{aligned}$$



Note that nodes $\{q_0\}$ and $\{q_0, q_T\}$ are unreachable and thus can be removed. After their removal we have



*The Complement machine:* Thought it is not necessary for the REG → FSM construction, we mention here that it is easy to construct a machine which accepts the *complement* of the language of another machine. I.e., $L(\neg M)$ =

$\Sigma^* - L(M)$. This is accomplished by simply swapping all final and non-final states. I.e.,

$$F_{\neg} = Q - F$$

The proof that $L(\neg M) = \neg L(M)$ is left as an exercise for the reader, however, we will mention, once proved, this shows that REG is closed under complementation.