

CDM

# Algebra of Regular Languages

Klaus Sutner

Carnegie Mellon University

Fall 2009

# Outline

- 1 Kleene's Theorem
- 2 The Algebra of Languages
- 3 Conversion To Regular Expression
- 4 Conversion To Machine
- 5 Realistic Regular Expressions
- 6 Limitations of Regular Languages

## The Structure of Finite State Machines

- We have a number of good algorithms to manipulate FSMs and to test their properties.
- Alas, these machines have one serious drawback: they don't have any nice internal structure and they are somewhat difficult to describe (other than by a brute force table).
- It would be nice if we could somehow build machines in some systematic way, using only trivial basic components and a handful of reasonably simple operations.
- Is there any hope for this?

## Krohn-Rhodes

There is a basic result from 1962, the Krohn-Rhodes Theorem, arguably one of the most important results in automata theory.

The theorem says, in essence, that every deterministic automaton can be decomposed into simple components.

Alas, the argument uses algebra quite heavily (the simple components are machines corresponding to finite simple groups plus a 2-state flip-flop automaton).

Recently, some actual implementations have been developed, but overall the theorem seems to be mostly of theoretical value.

BTW, this is one of the rare cases where 2 PhDs were granted for one result, Krohn at Harvard and Rhodes at MIT.

## Kleene's theorem

Here is another, much easier result that has many direct practical applications.

### Theorem (Kleene)

*Every regular language over  $\Sigma$  can be constructed from  $\emptyset$  and  $\{a\}$ ,  $a \in \Sigma$ , using only the operations union, concatenation and Kleene star.*

Regular languages are closed under other operations such as intersection and complement, but these are not needed to construct a regular language from the basic ones.

Before we sketch the proof, let us introduce the notation system for regular languages suggested by Kleene's result.

## Regular Expressions

### Definition

A **regular expression** is a term constructed as follows:

- Basic expressions:  $\emptyset$ ,  $\underline{a}$  for all  $a \in \Sigma$
- Operators:  $(E_1 + E_2)$ ,  $(E_1 \cdot E_2)$ ,  $(E^*)$ .

For example,

$$((\underline{a} \cdot (\underline{b}^*)) + \underline{c})$$

is a regular expression. While correct, this is too clumsy for words: as usual in arithmetic, one uses precedence ordering to avoid parentheses and drops the dot for concatenation.

One often allows  $\underline{\epsilon}$  as a primitive denoting the empty word (though this is technically redundant since  $\emptyset^*$  denotes the same language).

## The Corresponding Languages

We can associate a language with each regular expression:

$$\mathcal{L}(\emptyset) = \emptyset$$

$$\mathcal{L}(\underline{a}) = \{a\}$$

$$\mathcal{L}(E_1 + E_2) = \mathcal{L}(E_1) \cup \mathcal{L}(E_2)$$

$$\mathcal{L}(E_1 \cdot E_2) = \mathcal{L}(E_1) \cdot \mathcal{L}(E_2)$$

$$\mathcal{L}(E^*) = \mathcal{L}(E)^*$$

So by Kleene's theorem, for every regular language  $L$  there is a regular expression  $\alpha$  such that  $\mathcal{L}(\alpha) = L$ .

Lastly, one avoids the underlines and writes things like  $ab^* + c$ , it's always clear from context what is meant. One should also be relaxed about identifying  $\{x\}$  and  $x$  whenever convenient.

## Regex Example

### Example

All words containing *bab*:  $(a + b)^* bab(a + b)^*$ .

All words containing 3 *a*'s:  $b^* ab^* ab^* ab^*$

All words not containing *aaa*:  $(\varepsilon + a + aa)(b + ba + baa)^*$

### Exercise

*Construct a regex for all words with an even number of *a*'s and *b*'s.*

### Exercise

*Construct a regex for all words with an odd number of *a*'s and *b*'s.*

## Proof Sketch Kleene

One direction is easy given the results with already have.

To show that  $\mathcal{L}(\alpha)$  is regular for  $\alpha$  all we need is induction on the build-up of  $\alpha$ :

- The claim is trivial for atomic regular expressions.
- For compound regular expressions use closure under union, concatenation and Kleene star.

More on realistic implementations later.

## Proof Sketch Kleene, the Hard Part

Suppose we have an NFA that accepts some regular language  $L$ . Assume  $Q = [n]$ . For  $p, q$  in  $Q$  define

$$L_{p,q} = \mathcal{L}(\langle Q, \Sigma, \delta; \{p\}, \{q\} \rangle)$$

Then  $L = \bigcup_{p \in I, q \in F} L_{p,q}$  and it suffices to construct regular expressions for the  $L_{p,q}$ .

In order to obtain an inductive argument, define a run from state  $p$  to state  $q$  to be  **$k$ -bounded** if all intermediate states are no greater than  $k$ . Note that  $p$  and  $q$  themselves are not required to be bounded by  $k$ .

Now consider the approximation languages:

$$L_{p,q}^k = \{x \in \Sigma^* \mid \text{there is a } k\text{-bounded run } p \xrightarrow{x} q \}.$$

Note that  $L_{p,q}^n = L_{p,q}$ .

## Proof Sketch, contd.

One can build expressions for  $L_{p,q}^k$  by induction on  $k$ .

For  $k = 0$  the expressions are easy:

$$L_{p,q}^0 = \begin{cases} \sum_{\tau(p,a,q)} a & \text{if } q \neq p, \\ \sum_{\tau(p,a,q)} a + \varepsilon & \text{otherwise.} \end{cases}$$

So suppose  $k > 0$ . The key idea is to use the equality

$$L_{p,q}^k = L_{p,q}^{k-1} + L_{p,k}^{k-1} \cdot (L_{k,k}^{k-1})^* \cdot L_{k,q}^{k-1}$$

Done by IH. □

## Code

```
foreach p = 1,..,n do
  foreach q = 1,..,n do
    initialize A[p,q,0];

  foreach k = 1,..,n do
    foreach p = 1,..,n do
      foreach q = 1,..,n do
        A[p,q,k] = A[p,q,k-1] + A[p,k,k-1].A[k,k,k-1]*.A[k,q,k-1];

return sum( A[p,q,n] | p in I, q in F );
```

## Aside 1: Algebra

Note that Kleene's theorem really establishes an algebraic result. Define the **language semiring** over  $\Sigma$  to be

$$\mathcal{L}(\Sigma) = \langle \mathfrak{P}(\Sigma^*), \cup, \cdot, *, \emptyset, \varepsilon \rangle$$

This is a type of algebra with 3 operations and two constants.

Then Kleene's theorem can be interpreted thus: the least subalgebra generated by the singletons  $\{a\}$ ,  $a \in \Sigma$ , consists precisely of the regular languages. More later.

## Aside 2: Déjà Vu, All Over Again

This should look eminently familiar: logically, Floyd-Warshall's all-pairs shortest path algorithm is essentially the same.

The underlying algebra, the min-plus semiring, is different and simpler (loops are irrelevant for shortest paths). The recursion for dynamic programming looks like this:

$$d_{p,q}^k = \min(d_{p,q}^{k-1}, d_{p,k}^{k-1} + d_{k,q}^{k-1}).$$

And, of course, we are calculating with rational numbers here, not with formal expressions. There is no danger of expressions blowing up.

## Aside 3: Simple Application

When arguing about properties of regular languages it is sometimes easier to use regular expressions rather than machines.

For example, consider closure under homomorphisms.

### Definition

A **homomorphism** is a map  $f : \Sigma^* \rightarrow \Gamma^*$  such that

$$f(x_1x_2 \dots x_n) = f(x_1)f(x_2) \dots f(x_n)$$

where  $x_i \in \Sigma$ . In particular  $f(\varepsilon) = \varepsilon$ .

Note that a homomorphism can be represented by a finite table: we only need  $f(a) \in \Gamma^*$  for all  $a \in \Sigma$ .

## Closure under Homomorphisms

### Claim

*Regular languages are closed under homomorphisms:  $f(L)$  is regular whenever  $L$  is.*

Given a homomorphism  $f$  (a finite table) define a regular expression  $\alpha^f$  over  $\Gamma$  for each  $\alpha$  over  $\Sigma$ :

$$\begin{array}{ll} \emptyset \mapsto \emptyset & \alpha + \beta \mapsto \alpha^f + \beta^f \\ \varepsilon \mapsto \varepsilon & \alpha \cdot \beta \mapsto \alpha^f \cdot \beta^f \\ a \mapsto f(a) & \alpha^* \mapsto (\alpha^f)^* \end{array}$$

Then  $\mathcal{L}(\alpha^f) = f(\mathcal{L}(\alpha))$ .

### Exercise

*Give a machine based proof of the claim.*

- Kleene's Theorem

## 2 The Algebra of Languages

- Conversion To Regular Expression
- Conversion To Machine
- Realistic Regular Expressions
- Limitations of Regular Languages

## Algebra of Languages

Given an alphabet  $\Sigma$  we consider the collection of all languages over  $\Sigma$ :

$$\mathcal{L}(\Sigma) = \mathfrak{P}(\Sigma^*) = \{L \mid L \subseteq \Sigma^*\}$$

There are the obvious Boolean operations union, intersection and complement that can be applied to languages over  $\Sigma$ . Hence we have a Boolean algebra

$$\langle \mathcal{L}(\Sigma), \cup, \cap, \bar{\phantom{x}} \rangle$$

That's OK, but not terribly interesting: at no point are we using the fact that the sets in question are sets of words, rather than arbitrary objects.

So the question is: what are interesting operations on  $\mathcal{L}(\Sigma)$  that exploit this fact?

## Concatenation

Recall that we can “multiply” two words by concatenating them.

We can lift this operations to language by applying concatenation pointwise. Given two languages  $L_1, L_2 \subseteq \Sigma^*$  we concatenate them by concatenating all possible combinations of words:

$$L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}.$$

Note that this operation fails to commute.

Also,  $L \cdot \emptyset = \emptyset \cdot L = \emptyset$  and  $L \cdot \{\varepsilon\} = \{\varepsilon\} \cdot L = L$ .

### Example

$$\{a, b\} \{a, b\} = \{aa, ab, ba, bb\}$$

## Kleene Star

Boolean operations and concatenation when applied to finite languages produce only finite and co-finite languages and are thus insufficient to generate interesting languages. We need at least one operation that generates an infinite language from a finite one. Here is one such operation that turns out to be immensely useful.

### Definition

Let  $L$  be a language. The **powers** of  $L$  are the languages obtained by repeated concatenation:

$$L^0 = \{\varepsilon\}$$
$$L^{k+1} = L^k \cdot L$$

The **Kleene star** of  $L$  is the language

$$L^* = L^0 \cup L^1 \cup L^2 \dots \cup L^n \cup \dots$$

Kleene star corresponds roughly to a while-loop or iteration.

## Star Examples

### Example

$\{a, b\}^*$ : all words over  $\{a, b\}$

### Example

$\{a, b\}^* \{a\} \{a, b\}^* \{a\} \{a, b\}^*$ : all words over  $\{a, b\}$  containing at least two  $a$ 's

### Example

$\{\epsilon, a, aa\} \{b, ba, baa\}^*$ : all words over  $\{a, b\}$  not containing a subword  $aaa$

### Example

$\{0, 1\} \{0, 1\}^*$ : all numbers in binary, with leading 0's

$\{1\} \{0, 1\}^* \cup \{0\}$ : all numbers in binary, no leading 0's

## Regular Languages as a Subalgebra

The choice of concatenation and Kleene star may seem rather arbitrary. It is justified by Kleene's theorem that every regular language can be obtained from singletons  $\{a\}$  for  $a \in \Sigma$ , and  $\emptyset$ , by finitely many applications of the operations union, concatenation and Kleene star.

In other words, the collection  $\text{Reg}(\Sigma)$  of all regular languages over alphabet  $\Sigma$  is a subalgebra of the **language semiring**

$$\mathcal{L}(\Sigma) = \langle \mathfrak{P}(\Sigma^*), \cup, \cdot, *, \emptyset, \varepsilon \rangle$$

and is generated by singletons  $\{a\}$ .

## How about Intersection and Complement?

Note that the theorem makes a rather surprising claim: it suffices to consider operations union, concatenation and Kleene star when one tries to construct regular languages from atomic pieces (in this case singletons  $\{a\}$  and the empty set).

But regular languages are closed under intersection and complement. It is by no means clear how

$$L \cap K \quad \text{or} \quad \Sigma^* - L$$

can be so generated, even if we already know how to handle  $K$  and  $L$ .

## How about Division?

Conspicuously absent from our algebra so far is any operation resembling division. If we think of division as the inverse of multiplication (i.e., concatenation) the natural answer is the following.

### Definition

Let  $L \subseteq \Sigma^*$  be a regular language and  $x \in \Sigma^*$ . The **left quotient of  $L$  by  $x$**  is

$$x^{-1}L = \{y \in \Sigma^* \mid xy \in L\}.$$

So we are simply removing a prefix  $x$  from all words in the language that start with this prefix. If there is no such prefix we get an empty quotient.

This is the reason why it is a bit more elegant to talk about quotients in the context of languages rather than words: for words  $x$  and  $y$  the quotient  $x^{-1}y$  would be undefined whenever  $x$  fails to be a prefix of  $y$ .

## Algebra of Quotients

To simplify notation, let

$$\Delta(L) = \begin{cases} \{\varepsilon\} & \text{if } \varepsilon \in L, \\ \emptyset & \text{otherwise.} \end{cases}$$

## Lemma

Let  $a \in \Sigma$ ,  $x, y \in \Sigma^*$  and  $L, K \subseteq \Sigma^*$ . Then the following hold:

- $(xy)^{-1}L = y^{-1}x^{-1}L$ ,
- $x^{-1}(L \cup K) = x^{-1}L \cup x^{-1}K$ ,
- $x^{-1}(L \cap K) = x^{-1}L \cap x^{-1}K$ ,
- $x^{-1}(\Sigma^* - L) = \Sigma^* - x^{-1}L$ ,
- $a^{-1}(LK) = (a^{-1}L)K \cup \Delta(L)a^{-1}K$ ,
- $a^{-1}L^* = (a^{-1}L)L^*$ .

## Comments

Note that  $(xy)^{-1}L = y^{-1}x^{-1}L$  and NOT  $x^{-1}y^{-1}L$ . The problem is that algebraically left quotients are a right action. Oh well.

Quotients coexist peacefully with Boolean operations, we can just push the quotients inside.

But for concatenation and Kleene star things are a bit more involved; the lemma makes no claims about the general case where we divide by a word rather than a single letter.

### Exercise

*Prove the last lemma.*

### Exercise

*Generalize the rules for concatenation and Kleene star to words.*

## All Quotients

Here is a wild idea that will turn out to be extremely useful.

Given a regular language  $L$ , can we calculate all the quotients

$$\{x^{-1}L \mid x \in \Sigma^*\}$$

Note that we may well have  $x^{-1}L = y^{-1}L$  for different  $x$  and  $y$ , so it is not clear how large the set of all quotients will be.

## Quotients Example 1

Using the lemma, we can compute the quotients of  $a^*b$  as follows: we systematically compute all quotients with respect to  $a^{-1}$  and  $b^{-1}$ .

$$a^{-1} a^* b = a^* b$$

$$b^{-1} a^* b = \varepsilon$$

$$a^{-1} \varepsilon = \emptyset$$

$$b^{-1} \varepsilon = \emptyset$$

$$a^{-1} \emptyset = \emptyset$$

$$b^{-1} \emptyset = \emptyset$$

So there are exactly 3 quotients:  $a^*b$ ,  $\varepsilon$  and  $\emptyset$ .

## Quotients Example 2

Let  $L$  be the finite language  $\{a, aab, bbb\}$ .

There are exactly six quotients of  $L$ :

| $x$        | $x^{-1}L$          |
|------------|--------------------|
| $\epsilon$ | $\{a, aab, bbb\}$  |
| $a$        | $\{\epsilon, ab\}$ |
| $b$        | $\{bb\}$           |
| $bb$       | $\{b\}$            |
| $aab$      | $\{\epsilon\}$     |
| $ab$       | $\emptyset$        |

Note that the  $x$  is not uniquely determined, for example  $(abz)^{-1}L = (baz)^{-1}L = \emptyset$  for any  $z$ .

## Quotients Example 3

A larger example,  $L = L_1 = a^*b^* \cup bab$ .

|                 |               |       |
|-----------------|---------------|-------|
| $a^{-1}L_1$     | $a^*b^*$      | $L_2$ |
| $b^{-1}L_1$     | $b^* \cup ab$ | $L_3$ |
| $a^{-1}L_2$     | $L_2$         |       |
| $b^{-1}L_2$     | $b^*$         | $L_4$ |
| $a^{-1}L_3$     | $b$           | $L_5$ |
| $b^{-1}L_3$     | $L_4$         |       |
| $a^{-1}L_4$     | $\emptyset$   | $L_6$ |
| $b^{-1}L_4$     | $L_4$         |       |
| $a^{-1}L_5$     | $L_6$         |       |
| $b^{-1}L_5$     | $\varepsilon$ | $L_7$ |
| $a^{-1}L_{6/7}$ | $L_6$         |       |
| $b^{-1}L_{6/7}$ | $L_6$         |       |

## Exercise

Verify this table.

## Quotients Example 4

An even larger example,  $L = L_1 = a^*ba^* \cup b^*ab^*$ .

|             |                 |       |             |             |       |
|-------------|-----------------|-------|-------------|-------------|-------|
| $a^{-1}L_1$ | $a^*ba^* + b^*$ | $L_2$ | $a^{-1}L_6$ | $b^*$       | $L_9$ |
| $b^{-1}L_1$ | $b^*ab^* + a^*$ | $L_3$ | $b^{-1}L_6$ | $b^*ab^*$   |       |
| $a^{-1}L_2$ | $a^*ba^*$       | $L_4$ | $a^{-1}L_7$ | $b^*$       |       |
| $b^{-1}L_2$ | $a^* + b^*$     | $L_5$ | $b^{-1}L_7$ | $\emptyset$ |       |
| $a^{-1}L_3$ | $a^* + b^*$     | $L_6$ | $a^{-1}L_8$ | $\emptyset$ |       |
| $b^{-1}L_3$ | $b^*ab^*$       | $L_7$ | $b^{-1}L_8$ | $b^*$       |       |
| $a^{-1}L_4$ | $a^*ba^*$       | $L_8$ | $a^{-1}L_9$ | $\emptyset$ |       |
| $b^{-1}L_4$ | $a^*$           |       | $b^{-1}L_9$ | $\emptyset$ |       |
| $a^{-1}L_5$ | $a^*$           |       |             |             |       |
| $b^{-1}L_5$ | $b^*$           |       |             |             |       |

## Quotients Example 5

$$L = \{ a^i b^i \mid i \geq 0 \} = \{ \varepsilon, ab, aabb, aaabbb, \dots \}$$

Here things become a bit complicated: we obtain infinitely many quotients.

$$\begin{aligned} (a^k)^{-1}L &= \{ a^i b^{i+k} \mid i \geq 0 \} \\ (a^k b^l)^{-1}L &= \{ b^{k-l} \} & 1 \leq l \leq k \\ (a^k b^l)^{-1}L &= \emptyset & l > k \end{aligned}$$

This is no coincidence: the language  $L$  is not regular. As we will see shortly, the number of quotients is finite if, and only if, the language is regular.

## Computing Quotients

These calculations can be described less informally by the following algorithm.

```
i = j = 1;
L[1] = L;
while( i <= j ) {
    foreach a in Sigma do
        K = left_quotient( a, L[i] );
        if( K is new ) L[++j] = K;
    i++;
}
```

Note that this is again a closure operation: we generate the smallest collection  $\mathcal{L}$  of languages that contains  $L$  and is closed under quotients:

- $L \in \mathcal{L}$ ,
- $K \in \mathcal{L}$  implies  $a^{-1}K \in \mathcal{L}$ .

## Is this an Algorithm?

Can we really compute the number of quotients of a given regular language?  
Could we build a Turing machine to do this? Or is this just un-implementable pseudo-code?

The logical control structure is easy: just a while-loop. But we need to represent the basic objects and operations:

- represent languages by some datastructure,
- implement the operations  $a^{-1}K$ ,
- implement the equality test  $K = K'$ .

Are all these problems surmountable?

## Real Implementation

Naturally, we represent languages by machines (we don't have much choice at this point).

- Since we are only dealing with regular languages we can use DFAs as representation.
- Quotients are then easy to implement: just move the initial state.
- The equality test comes down to checking Equivalence of DFAs, we already know how to do this.

Note how the choice of datastructure really settles the whole issue: if a regular language is represented by a DFA we know how to compute quotients, and we know how to check equality.

The question arises: how efficient is this approach?

## Running Time Analysis

Suppose the DFA representing  $L$  has  $n$  states. For simplicity we ignore the size of the alphabet.

Clearly, there will be at most  $n$  quotients to compute.

For each one, we have to test equality against  $O(n)$  others.

Doing this the obvious way requires  $O(n^2)$  steps for each equality check, so each new quotient requires  $O(n^3)$  steps.

The whole algorithm is then an unimpressive  $O(n^4)$ .

Can we speed this up?

### Exercise

*Explain the running time of this method in detail. Take into account the size of the alphabet.*

- Kleene's Theorem
- The Algebra of Languages
- ③ Conversion To Regular Expression
  - Conversion To Machine
  - Realistic Regular Expressions
  - Limitations of Regular Languages

## Machine to Regex

There are two basic approaches to converting a finite state machine to a regular expression:

- **Linear systems of equations**

The machine is converted into a system of linear equations over the language semiring. The system can then be solved using Arden's lemma.

- **Kleene's State elimination method**

The proof of Kleene's theorem provides a dynamic programming algorithm.

Converting a finite state machine to a regular expression is a bit of an academic exercise.

The key problem is that for both approaches to yield manageable expressions one needs to simplify the intermediate results. There are heuristics to do that, but in general simplification is computationally hard.

## Kleene's Method

The proof the theorem provides a direct dynamic programming algorithm: construct expression  $\alpha(p, q, k)$  in stages  $k = 0, \dots, n$  for  $1 \leq p, q \leq n$ , assuming the state set is  $[n]$ .

There are two problems this algorithm:

- There are  $n^2(n + 1)$  expressions to construct, which is bad but not fatal.
- The expressions roughly quadruple in size in the step from level  $k$  to level  $k + 1$ . This is a disaster.

In practice, one has to simplify the expressions to make them smaller and choose the elimination order cleverly so that some of the expressions are not needed (top-down versus bottom-up dynamic programming).

## Top Down Elimination

Here is a reasonable method for small machines.

- Add two new states  $b$  (begin) and  $e$  (exit) to the machine. Add  $\varepsilon$ -transitions from  $b$  to all initial states, and from all final states to  $e$ .
- Think of the edge labels as regular expressions. We will successively remove all states other than  $b$  and  $e$ .
- To this end pick some state  $r \in Q$ . For all incoming transitions  $p \xrightarrow{\alpha} r$ , outgoing transitions  $r \xrightarrow{\gamma} q$ , and self-loops  $r \xrightarrow{\beta} r$ , add a new transition

$$p \xrightarrow{\alpha\beta^*\gamma} q$$

In the end, remove  $r$  and all incident transitions.

- Repeat until only  $b$  and  $e$  remain.

## Result

After all states in  $Q$  are eliminated we are left with

$$b \xrightarrow{\alpha} e$$

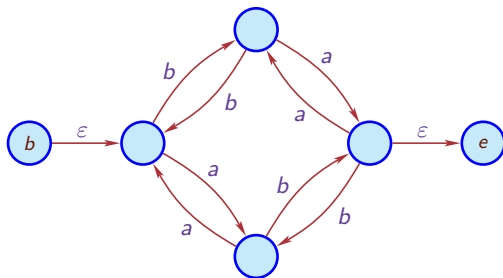
where  $\alpha$  denotes the language of the machine.

The proof is to show by induction that, at any point during the construction, the “automaton” is equivalent to the original one. Here we need to generalize the notion of automaton a little: allow regular expressions as labels rather than just letters.

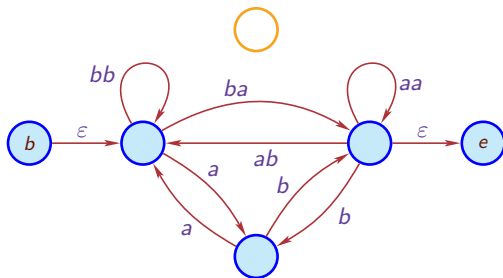
### Exercise

*Figure out the details.*

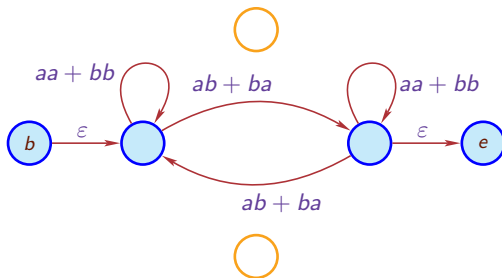
## Odd/Odd, Step 0



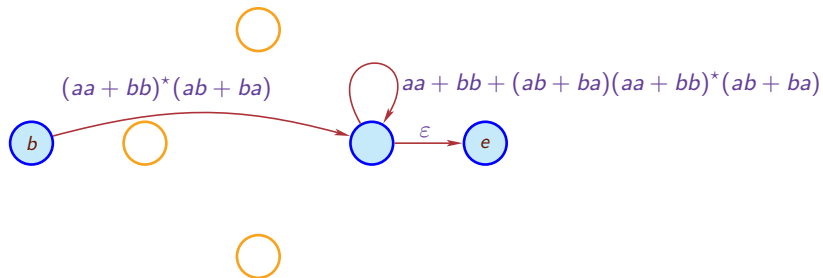
## Odd/Odd, Step 1



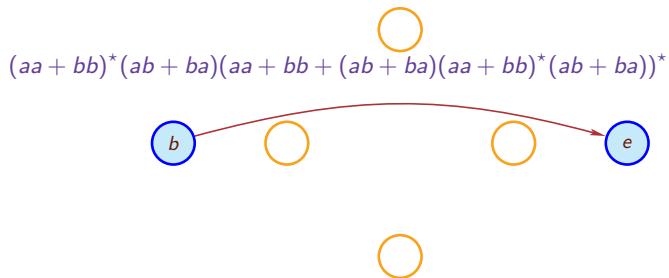
## Odd/Odd, Step 2



## Odd/Odd, Step 3



## Odd/Odd, Step 4



## Final Result

The final regex is

$$\alpha = (aa + bb)^*(ab + ba)(aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*$$

Why should this be correct?

A little argument shows that the last part

$$(aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*$$

denotes all even/even words.

Then  $\alpha$  must be correct, too: all odd/odd words consist of a (possibly empty) prefix  $(aa + bb)^*$ , followed by  $(ab + ba)$ , followed by an even/even word.

## Simplification

To keep the size of the expressions small it is important to apply various simplifications. For example, the following rules seem reasonable:

$$\emptyset \cdot \alpha \mapsto \emptyset$$

$$\varepsilon \cdot \alpha \mapsto \alpha$$

$$\alpha + \alpha \mapsto \alpha$$

$$\varepsilon + \alpha\alpha^* \mapsto \alpha^*$$

Unfortunately, the algebra of regular expressions is complicated and there is no simple set of rules that would produce reasonable expressions.

## Basic Equations

Disregarding Kleene star, the basic rules are not too bad:

$$(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma)$$

$$\alpha + \beta = \beta + \alpha$$

$$\emptyset + \alpha = \alpha$$

$$\alpha + \alpha = \alpha$$

$$(\alpha \cdot \beta) \cdot \gamma = \alpha \cdot (\beta \cdot \gamma)$$

$$\varepsilon \cdot \alpha = \alpha \cdot \varepsilon = \alpha$$

$$\emptyset \cdot \alpha = \alpha \cdot \emptyset = \emptyset$$

$$\alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma$$

$$(\beta + \gamma) \cdot \alpha = \beta \cdot \alpha + \gamma \cdot \alpha$$

Alas ...

## Rules for Kleene

But Kleene star causes huge problems. Here are some (natural?) rules:

$$(\alpha + \beta)^* = (\alpha^* \beta)^* \alpha^*$$

$$(\alpha \beta)^* = \varepsilon + \alpha (\beta \alpha)^* \beta$$

$$(\alpha^*)^* = \alpha^*$$

$$\alpha^* = (\varepsilon + \alpha + \dots + \alpha^{n-1}) \cdot (\alpha^n)^*$$

Note that the last equation is actually an infinite family of equations; it is known that no finite family will do.

Not to mention that for simplification we need rewrite rules, not equations.

## Horror Example

Here is the result running a conversion algorithm with some degree of simplifications on the even/even example (which is easier than odd/odd):

$$\begin{aligned} &\epsilon + b(bb)^*b + (a + b(bb)^*ba)(aa + ab(bb)^*ba)^*(a + ab(bb)^*b) + \\ &\quad (b(bb)^*a + (a + b(bb)^*ba)(aa + ab(bb)^*ba)^*(b + ab(bb)^*a)) \\ &\quad (a(bb)^*a + (b + a(bb)^*ba)(aa + ab(bb)^*ba)^*(b + ab(bb)^*a))^* \\ &\quad (a(bb)^*b + (b + a(bb)^*ba)(aa + ab(bb)^*ba)^*(a + ab(bb)^*b)) \end{aligned}$$

It takes quite a bit of effort just to check that this expression describes the even/even language.

## Arden's lemma

Consider a linear equation of the form

$$X = A \cdot X + B.$$

where  $A, B \subseteq \Sigma^*$  are arbitrary languages, though we will be mostly interested in the case where  $A$  and  $B$  are finite or perhaps regular. We are looking for a solution  $X_0 \subseteq \Sigma^*$ .

As it turns out, linear equations are rather easy to solve.

### Lemma (Arden's Lemma)

*Let  $A$  and  $B$  be languages over  $\Sigma$ . Then the equation  $X = A \cdot X + B$  has a solution  $X_0 = A^*B$ . Moreover, if  $\varepsilon \notin A$ , then this solution is unique. In any case,  $X_0$  is the smallest solution (with respect to set-theoretic inclusion).*

## Proof

To see that  $X_0 = A^*B$  is a solution note that

$$AX_0 + B = AA^*B + B = (A^+ + \varepsilon)B = A^*B = X_0.$$

Now let  $Z$  be any solution, so  $Z = AZ + B$ . Then for all  $k \geq 0$ :

$$Z = A^{k+1}Z + (A^k + \dots + \varepsilon)B.$$

Hence  $X_0 \subseteq Z$ .

Lastly suppose  $\varepsilon \notin A$  and let  $x \in Z$ ; set  $k = |x|$ . Then  $x \notin A^{k+1}Z$ , whence necessarily  $x \in (A^k + \dots + \varepsilon)B \subseteq X_0$  and we are done.  $\square$

In the applications of interest to us  $\varepsilon \notin A$  so that the solution is unique.

## Fixed Point Perspective

The solution of  $A \cdot X + B = X$  can be considered to be the least fixed point of the map

$$f(Z) = A \cdot Z + B$$

We can construct the least fixed point inductively:

$$Z_0 = \emptyset$$

$$Z_{i+1} = A \cdot Z_i + B$$

Then  $\bigcup_i Z_i$  is the least fixed point ( $X_0$  on the last slide).

There is no problem with existence here, we are dealing with a monotonic map in a complete lattice (Knaster-Tarski).

## Regularity

Note that Arden's lemma implies that equation  $X = A \cdot X + B$  has a regular solution  $X_0 = A^*B$  whenever  $A$  and  $B$  are regular. Moreover, if  $A$  does not contain  $\varepsilon$  this is the only solution. If  $A$  and  $B$  are given as rational expressions we obtain a rational expression for the solution.

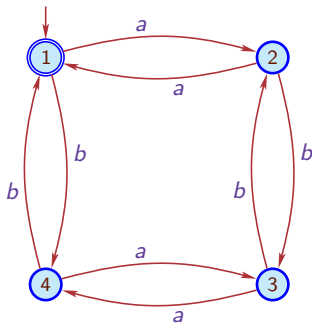
### Example

The equation  $X = bX + a$  has  $b^*a$  as its unique solution.

By contrast, the equation  $X = b^*X + a$  has infinitely many solutions  $X_k = b^*(a + a^2 + \cdots + a^k)$  and  $X_1 = b^*a$  is the least solution. Indeed, there are uncountably many solutions  $b^* \cdot L$  where  $a \in L \subseteq a^*$  is arbitrary.

## Even/Even Example

Let's return to the even/even language. The canonical DFA looks like so:



We convert the DFA into a system of equations.

## Corresponding System

$$X_1 = \varepsilon + aX_2 + bX_4 \quad (1)$$

$$X_2 = aX_1 + bX_3 \quad (2)$$

$$X_3 = aX_4 + bX_2 \quad (3)$$

$$X_4 = aX_3 + bX_1 \quad (4)$$

Substituting (2) and (4) into (1) and (3) we get

$$X_1 = \varepsilon + (aa + bb)X_1 + (ab + ba)X_3 \quad (5)$$

$$X_3 = (aa + bb)X_3 + (ab + ba)X_1 \quad (6)$$

Applying Arden's lemma to (6) we get

$$X_3 = (aa + bb)^*(ab + ba)X_1 \quad (7)$$

## The Solution

Substituting (7) into (5)

$$X_1 = \varepsilon + ((aa + bb) + (ab + ba)(aa + bb)^*(ab + ba))X_1$$

Applying Arden's lemma one more time we get

$$X_1 = (aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*$$

which solution makes intuitive sense.

It is important to note, though, that we tacitly did quite a bit of cleanup along the way, writing the expressions in a simplified form. Remember the horror example, from above.

## Another Example

Needless to say, solving systems of equations does not always produce a neat solution either. Consider the system

$$X = aX + bY$$

$$Y = \varepsilon + aX + ba^*$$

$$Z = aZ + b$$

$Z$  can immediately be replaced by  $a^*b$ . If we solve for  $X$  in the first equation,  $X = a^*bY$ , and substitute in the second and then solve for  $Y$  and resubstitute we get

$$X = a^*b(a^+b)^*(\varepsilon + ba^*)$$

$$Y = (a^+b)^*(\varepsilon + ba^*)$$

So far, so good.

## Different Approach

However, we could also first substitute the second equation into the first and solve for  $X$  (after getting rid of  $Z$ ).

This leads to solutions

$$X' = (a + b(ab)^*aa)^*b(ab)^*(\varepsilon + ba^*)$$

$$Y' = (ab)^*(aa(a + b(ab)^*aa)^*b(ab)^*(\varepsilon + ba^*) + \varepsilon + ba^*)$$

Unless we made a mistake, these expressions must be equivalent to  $X$  and  $Y$ , but this is certainly not obvious from looking at them.

What is sorely missing here is a simplification algorithm that brings a regular expression into a normal form. We can check for equivalence by converting to finite state machines and then testing these for equivalence.

- Kleene's Theorem
- The Algebra of Languages
- Conversion To Regular Expression
- ④ Conversion To Machine
  - Realistic Regular Expressions
  - Limitations of Regular Languages

## Regex To Machine

We will ignore the issue of parsing a regular expression and simply worry about how to construct the finite state machine.

It is not hard to guess that we will build the machine by induction on the structure of the regular expression, but the question is what kind of machine we should be build.

DFAs are certainly a bad choice since we have to deal with concatenation and Kleene star. It is a fair guess that some kind of NFAE is the right target architecture.

As it turns out, the construction becomes quite a bit easier if insist on very special NFAEs. Of course, the right choice requires a bit of experimentation.

## Begin/Exit Automata

### Definition

A **begin/exit automaton (BEA)** is an NFAE that has exactly one initial state  $b$  (the begin), exactly one final state  $e$  (the exit). Furthermore, no transitions end at  $b$  and no transitions start at  $e$ .

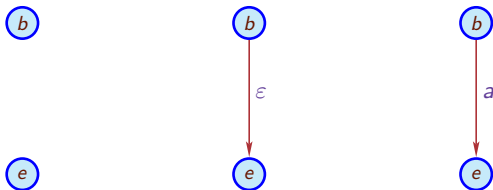
If we assume the state set  $[n]$  a BEA is essentially just a list of transitions: by renumbering we can make sure that  $b = 1$  and  $e = n$ .

Hence the data structure representing a BEA is particularly simple.

There are other ways to go about the conversion, but BEAs are probably the most intuitive choice.

## The Basic machines

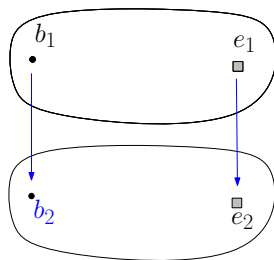
For expressions  $\emptyset$ ,  $\epsilon$  and  $a$  the corresponding BEAs are as follows.



In practice, the BEA for  $\emptyset$  is never used.

## Operations: Sum

A BEA for the sum of two BEAs.

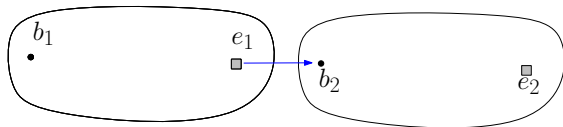


All new transitions are  $\epsilon$ -transitions.

Number of states:  $n_1 + n_2$ , number of transitions:  $t_1 + t_2 + 2$ .

## Operations: Product

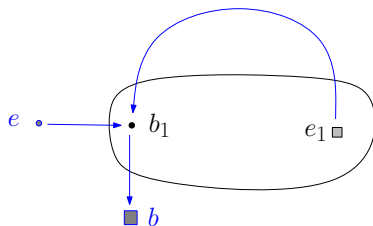
A BEA for the product of two BEAs.



Number of states:  $n_1 + n_2$ , number of transitions:  $t_1 + t_2 + 1$ .

## Operations: Star

A BEA for the Kleene star of a BEA.

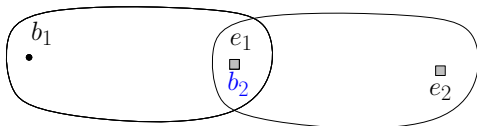


Number of states:  $n_1 + 2$ , number of transitions:  $t_1 + 3$ .

## Improvements

So far we have hardly used the conditions on begins and exits. They can be exploited to streamline the sum and product operations.

For example, a BEA for the product can be built like so:



The number of states is  $n_1 + n_2 - 1$ , the number of transitions is  $t_1 + t_2$ .

### Exercise

*Figure out how to optimize sums of BEAs.*

## Correctness

Even without optimization we have a linear time conversion algorithm.

### Theorem

*A regular expression can be converted in linear time into an equivalent begin/exit automaton.*

*Proof.* Write  $\alpha$ ,  $\sigma$ ,  $\pi$  and  $\kappa$  for the number of atomic symbols, sums, products and Kleene stars in the regular expression, respectively. Then the number of states/transitions in the corresponding BEA is bounded by

$$2\alpha + 2\kappa \qquad \alpha + 2\sigma + \pi + 3\kappa$$

□

### Exercise

*Determine the size of a BEA with optimization.*

## BEA Example

Expression  $(\epsilon + a + aa)(b + ba + baa)^*$  produces an 8-state BEA with 13 transitions (with brutal optimization):

|          |          |            |          |            |          |          |          |            |          |          |          |            |
|----------|----------|------------|----------|------------|----------|----------|----------|------------|----------|----------|----------|------------|
| 1        | 1        | 1          | 2        | 3          | 4        | 4        | 4        | 4          | 5        | 6        | 7        | 8          |
| <i>a</i> | <i>a</i> | $\epsilon$ | <i>a</i> | $\epsilon$ | <i>b</i> | <i>b</i> | <i>b</i> | $\epsilon$ | <i>a</i> | <i>a</i> | <i>a</i> | $\epsilon$ |
| 2        | 3        | 3          | 3        | 4          | 5        | 6        | 8        | 9          | 8        | 7        | 8        | 4          |

$\epsilon$ -elimination produces an NFA with initial states  $\{1, 3, 4, 9\}$  and final states  $\{9\}$ .

|          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 1        | 1        | 1        | 1        | 2        | 2        | 2        | 4        | 4        | 4        | 4        | 4        | 5        | 5        | 5        | 6        | 7        | 7        | 7        |
| <i>a</i> | <i>a</i> | <i>a</i> | <i>a</i> | <i>a</i> | <i>a</i> | <i>a</i> | <i>a</i> | <i>b</i> | <i>b</i> | <i>b</i> | <i>b</i> | <i>b</i> | <i>a</i> | <i>a</i> | <i>a</i> | <i>a</i> | <i>a</i> | <i>a</i> |
| 2        | 3        | 4        | 9        | 3        | 4        | 9        | 4        | 5        | 6        | 8        | 9        | 4        | 8        | 9        | 7        | 4        | 8        | 9        |

## Deterministic Machines

Determinizing the NFA yields a DFA on 7 states (initial state 1, all states other than the sink 6 are final).

|          |  |   |   |   |   |   |   |   |
|----------|--|---|---|---|---|---|---|---|
|          |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| <i>a</i> |  | 2 | 4 | 5 | 6 | 7 | 6 | 6 |
| <i>b</i> |  | 3 | 3 | 3 | 3 | 3 | 6 | 3 |

The corresponding minimal DFA has 4 states (initial state 1 and final states {1, 2, 3}).

|          |  |   |   |   |   |
|----------|--|---|---|---|---|
|          |  | 1 | 2 | 3 | 4 |
| <i>a</i> |  | 2 | 3 | 4 | 4 |
| <i>b</i> |  | 1 | 1 | 1 | 4 |

## Blow-Up

The regular expression

$$(a + b)^* a(a + b)(a + b)(a + b)(a + b)(a + b)$$

for  $L(a, -6)$  produces a BEA with 10 states

|            |     |     |            |            |     |     |     |     |     |     |     |     |     |     |     |
|------------|-----|-----|------------|------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1          | 2   | 2   | 2          | 3          | 4   | 5   | 5   | 6   | 6   | 7   | 7   | 8   | 8   | 9   | 9   |
| $\epsilon$ | $a$ | $b$ | $\epsilon$ | $\epsilon$ | $a$ | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ |
| 2          | 3   | 3   | 4          | 2          | 5   | 6   | 6   | 7   | 7   | 8   | 8   | 9   | 9   | 10  | 10  |

... but the DFA obtained from conversion has 65 states; the minimal DFA has 64 states.

- Kleene's Theorem
- The Algebra of Languages
- Conversion To Regular Expression
- Conversion To Machine
- ⑤ Realistic Regular Expressions
  - Limitations of Regular Languages

## Extended Regex

Our definition of regular expressions directly mimics Kleene's theorem. Sometimes it is convenient to enhance regular expressions a bit. There are two types of enhancements:

- More compact ways to describe regular languages.
- Describing non-regular languages.

Type 1 comes down to notational convenience, but may have radical effects on the running time of the conversion algorithm. For example, we know that regular languages are closed under intersection and complement. If we were to add corresponding regular expressions for these operations we would still describe regular languages. However, the conversion process based on BEAs now fails: the only way we can perform, say, complementation is by converting to a DFA first. This conversion may carry an exponential cost.

Type 2 requires a redesign of the acceptance testing algorithm: finite state machines are no longer sufficient, though the modifications may turn out to be fairly easy to do from an algorithmic point of view.

## Intersection

As an example, consider the addition of a new operation symbol  $\cap$  for intersection to regular expressions.

What has to change in the conversion algorithm? We need to add a product automata construction.

From the implementation perspective this is not too bad, but it breaks polynomial bounds on the size of the machine constructed from a regex. The following theorem shows that there is little hope for a simple remedy.

### Theorem

*Suppose  $M_1, \dots, M_n$  is a list of DFAs (over the same alphabet). It is PSPACE-hard to check whether there is a string that is accepted by all the  $M_i$ .*

## Complement

Similar problems arise when a complementation operation  $\neg$  is added. In the Real World<sup>TM</sup> complementation can often be kludged by piping.

```
fgrep foo file.txt | fgrep -v foobag
```

But to do this in general we have to construct a machine for the complement of a regular language – and that requires to build a DFA first, at a potentially exponential cost.

Also note that complement together with union automatically produces intersection, so the hardness result from above applies.

## Iteration

A very handy feature in most regular expression matchers generalizes concatenation and Kleene star.

| notation                | number of matches |
|-------------------------|-------------------|
| *                       | $\geq 0$          |
| +                       | $\geq 1$          |
| ?                       | $= 0, 1$          |
| { <i>n</i> }            | $= n$             |
| { <i>n</i> ,}           | $\geq n$          |
| { <i>n</i> , <i>m</i> } | $\geq n, \leq m$  |

So one can write things like

```
egrep -e '0\.[0-9]{5}'
```

to find all decimal numbers starting with 0 with exactly 5 digits after the decimal point.

- Kleene's Theorem
- The Algebra of Languages
- Conversion To Regular Expression
- Conversion To Machine
- Realistic Regular Expressions
- ⑥ Limitations of Regular Languages

## Limitations of Regular Languages

A brute-force cardinality argument that most languages fail to be regular. More importantly, relative simple and practically relevant languages such as

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

turn out to be non-regular.

It would be nice to have a reasonably simple test that can identify non-regular languages as such.

## Using Quotients

One approach is to count quotients.

### Proposition

*A language fails to be regular if, and only if, it has infinitely many left quotients.*

### Example

$L = \{ a^i b^i \mid i \geq 0 \}$  fails to be regular.

For consider the quotients  $K_i = (a^i)^{-1} L$ . The shortest word in  $K_i$  is  $b^i$ , so they are all distinct.

Alas, making sure that there are indeed infinitely many quotients can be difficult, we really need a less challenging test is needed.

## Pumping lemma

### Lemma (Pumping Lemma)

*For every regular language  $L$  there is a constant  $n$  such that for all words  $x \in L$  with  $|x| \geq n$  we have  $x = uvw$  where  $v \neq \varepsilon$ ,  $|uv| \leq n$  and  $uv^t w \in L$  for all  $t \geq 0$ .*

*Proof.*

Consider the minimal DFA  $M$  for  $L$  and let  $n$  be the number of states of  $M$ . Then any word in  $L$  of length at least  $n$  must trace a loop in the diagram of  $M$ . The claim follows.

□

The Pumping lemma is useless to establish regularity but often the weapon of choice to refute it.

## PL Example 1

$L = \{a^i b^i \mid i \geq 0\}$  fails to be regular.

Assume otherwise.

Let  $n$  be as in the PL and consider  $x = a^n b^n \in L$ .

Then  $x = uvw$  and  $v = a^i$  for some  $i > 0$ .

But then  $uv^t w \notin L$  for all  $t \neq 1$ , contradiction.

It follows that the language  $P$  of balanced parentheses is also non-regular.

For otherwise  $P \cap a^* b^* = L$  would also be regular, which assertion we already know to be false.

## PL Example 2

$L = \{ zz \mid z \in \Sigma^* \}$  fails to be regular.

Assume otherwise.

Let  $n$  be as in the PL and consider  $x = ab^n ab^n \in L$ .

Then  $x = uvw$  and  $|uv| \leq n$ . If  $v = a$  we get a contradiction with  $t = 0$ . If  $v = b^i$  for some  $i > 0$  we also get a contradiction with  $t = 0$ .

The problem here really is that a DFA cannot remember an arbitrarily long prefix  $z$  which is needed to check the remainder of the input.

## PL Example 3

$L = \{ zz^r \mid z \in \Sigma^* \}$  fails to be regular.

Assume otherwise.

Let  $n$  be as in the PL and consider  $x = (ab)^n(ba)^n \in L$ .

Then  $x = uvw$  and  $|uv| \leq n$ .

A straightforward but tedious argument shows that  $uv^t w$  cannot be a palindrome for any  $t \neq 1$ .

As in the last example, a DFA cannot remember an arbitrarily long prefix  $z$  which is needed to check the remainder of the input.

## Complexity of Languages

Incidentally, between

$$L_1 = \{ zz \mid z \in \Sigma^* \}$$

and

$$L_2 = \{ zz^r \mid z \in \Sigma^* \}$$

the second one (even length palindromes) is much simpler:

To recognize these words one only needs to attach a stack to a FSM (context free language).

For  $L_1$ , a simple stack is not sufficient (context sensitive language).

## Non-Regular Extensions

Many pattern matchers allow the user to specify repetitions of previously matched parts of a string.

The standard `egrep` for example allows

```
egrep -e '([a-z]*)\1'
```

which will match words of the form `ww`. It is not hard to see that these words form a non-regular language.

The `\1` is a so-called back-reference and matches whatever string has already matched the expression in parens.

## Back-References

In general,  $\backslash n$  refers to the string that has already matched the  $m$ th paren pair, which has to occur before the back-reference.

Also note that one actually has to understand the matching mechanism in greater detail (greedy versus lazy).

Usually the longest match possible is chosen.

For example, the expression

```
(((a|b)*c\2)*
```

matches all words in

$$\{ xcx \mid x \in \{a, b\}^* \}^*$$

## Primality

Here is a beautiful way to check primality, albeit in unary.

The file `prime.txt` contains strings of `a`'s up to length 25, one on each line.

```
> egrep -ve '^(aaa*)\1(\1)*$' prime.txt
a
aa
aaa
aaaaa
aaaaaaa
aaaaaaaaaaa
aaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

There is a little bug at `a`, but that's not hard to fix.

## Warning

Careful, though, the fancy stuff in regular expression matchers sometimes does not work right.

The expression `(.?)` means: match at most one single character and remember it. So the following is supposed to match all palindromes of even length up to 8:

```
egrep -e '(.?) (.?) (.?) (.?)\4\3\2\1'
```

It does, but it also crashes on odd length strings (this is GNU `grep-2.5.2`).