Speciale

Line Bie Pedersen 300976

Abstract

Contents

List of Examples

1 Introduction

Regular expressions is an important tool for matching strings of text. In many text editors and programming languages they provide a concise and flexible way of searching and manipulating text. They find their use in areas like data mining, spam detection, deep packet inspection and in the analysis of protein sequences.

1.1 Definitions and conventions

We have kept to the conventions when it comes to our definitions, so there should be no surprises in this section.

1.1.1 Notation

The empty string is denoted as ε. Let Σ be the alphabet, or set of symbols, used to write a string or a regular expression.

Automatons are represented as graphs, where states are nodes and transitions are edges. The start state has an arrow starting nowhere pointing to it. The accepting state is marked with double circles. Edges has an attached string, indicating on which input symbol this particular transition is allowed.

explain where bitvalues can be read, parenthesis, and an example perhaps when code is done

Regular expressions will be written in sans serif font: $\mathsf{a|b}$ and strings will be written slanted: *The cake is a lie*.

2 Regular expressions and finite automatons

A regular expression consists of two types of characters: Meta characters and literal characters. The meta characters have special meaning and are interpreted by a regular expression engine. The basic meta characters are parenthesis, the alternation operator and the Kleene star. Parenthesis provides grouping, alternation allows the choice between different text strings and the Kleene star repeats. The literal characters have no special meaning and match literally.

A formal definition of regular expressions:

Definition 1 (Regular expression) A regular expression over an alphabet Σ can be defined as follows:

- An empty string, ε, and any character from the alphabet Σ
- If r_1 and r_2 is regular expressions, then the concatenation $r_1 r_2$ is also a regular expression

- If r_1 and r_2 is regular expressions, then the alternation $r_1|r_2$ is also a regular expression
- If r is a regular expression, then so is the repetition $r*$

Any expression is a regular expression if it follows from a finite number of applications of the above rules. □

The precedence of the operators are: repetition, concatenation and alternation, from highest to lowest. Concatenation and alternation are both left-associative.

Example 1 (Regular expression) Here we have a somewhat complicated example of a regular expression that demonstrates the basic operators. Consider the sentence:

> This book was written using 100% recycled words.[1]

Other writings such as papers and novels also use words. If we want to catch sentences referring to these writings as well, we can use the regular expression: `(book|paper|novel)`.

To match the number 100 in the sentence, we could use the regular expression `100`. In most cases however we will not know beforehand how many words are recycled, so we may want to use the regular expression `(0|1|2|3|4|5|6|7|8|9)*`, which will match any natural number.

With this in mind we can write a regular expression to match our sentence:

```
This (book|paper|novel) was written using
(0|1|2|3|4|5|6|7|8|9)*% recycled words.
```
□

We have extended the regular expressions from definition 1 on the preceding page with the following: extra repetition operators, character classes, a wild card and some non-capturing parenthesis.

- The repetition operator $+$ causes the regular expression r to be matched one or more times. This can also be written as $rr*$
- The repetition operator $?$ causes the regular expression r to be matched zero or one times. This can also be written as $\varepsilon|r$
- A character class is delimited by [] and matches exactly one character in the input string. Special characters loose their meaning inside a character class; $*$, $+$, $?$, (,) and so on are treated as literals.

  Characters can be listed individually, e.g. `[abc]`, or they can be listed as ranges with the range operator: $-$, e.g. `[a-z]`. These can

[1] Terry Pratchett, Wyrd Sisters

be rewritten in terms of our original regular expression: `a|b|c` and `a|b|c...x|y|z` respectively.

To match characters not within the range, the complement operator is used. `^` used as the first character in a character class, elsewhere it will simply match literally, indicates that only characters not listed in the character class should match. E.g. `[^^]` will match anything but a `^`

- The wild card `.` will match any character, including a newline.
- For the non-capturing parenthesis we have the choice of notation. Here we will list a some of the options, where r is some regular expression:
  - The industry standard, to which Perl, Python, RE2 and most others adhere is: $(?:r)$. This is however plain ugly and confusing
  - Perl 6 **[?]** suggests use of square parenthesis instead: $[r]$. These are however already in use by the character classes.
  - A more intuitive notation could be using single parenthesis for non-capturing, (r), and double parenthesis for capturing, $((r))$
  - Since we are not using $\{r\}$ as special notation, this could be a good use, it surely would be the simplest to implement. This is however used in a repetition notation in industry standard

  Since there is a standard, we will adhere to it as well, and use $(?:r)$ for non-capturing parenthesis.

Example 2 As we saw in example 1, we can match a natural number with the regular expression `(0|1|2|3|4|5|6|7|8|9)*`. Using the expansions to regular expressions above, we can rewrite this as:

`[0-9]*` This literally means the same thing

`[0-9]+` We can use a different repetition operator and require there be at least one number

`[1-9][0-9]*` This matches any natural number as well, but it will not match any preceding zeros. This is a refinement, in that it will match fewer text strings than the first expression. This is however not always an advantage □

3 From regular expression to NFA

Every regular expressions can be converted to a NFA. In this section we will be looking at different methods for converting regular expressions to NFAs.

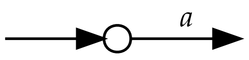


Figure 1: Fragment accepting a single character *a*

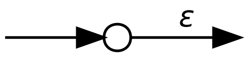


Figure 2: Fragment accepting the empty string

3.1 Thompson

The method described in this section first appeared in Ken Thompsons article from 1968 [**?**]. The NFA will be build in steps from smaller NFA fragments. A NFA fragment has an initial state, but no accepting state, instead it has one or more dangling edges leading nowhere (yet).

The base fragment corresponds to the regular expression consisting only of a single character *a*. The NFA fragment is shown in figure 1. One state with a single edge, marked with the character *a* is added. The new state is the initial state for this fragment and the edge is left dangling.

The second base fragment corresponds to the empty regular expression. The NFA fragment is shown in figure 2. One state with a single edge marked as a ε-edge is added. The new state is the initial state for this fragment and the edge is left dangling. This fragment is used for the empty regular expression and for alternations with one or more options left empty.

The first compound fragment is alternation, see figure 3 on the next page. The two subfragments R and S are automatons with initial states and some dangling edges. What else they are composed of, is of no consequence. One new state is added, this is made the initial state for this fragment. The initial state has two ε-edges leaving, connecting to the initial states of R and S. The dangling edges for the new fragment is the sum of the dangling edges leaving R and S.

Concatenation of two regular expressions R and S is achieved as shown in figure 4 on the following page. The dangling edges of R is connected to the initial state of S. The initial state for the new fragment is the initial state of R and the dangling edges of S is still left dangling.

Zero or more times repetition is shown in figure 5 on the next page. One new, initial, state is added. It has two ε-edges leaving, one is connected to the initial state of R and one is left dangling. The dangling edges of R is connected to the new initial state.

Finally an accepting state is patched into the NFA. All edges left dangling is connected to the accepting state.

In order for this method to be successful, the regular expression has to

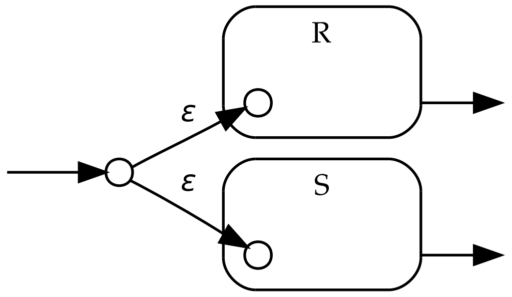


Figure 3: Alternation R|S

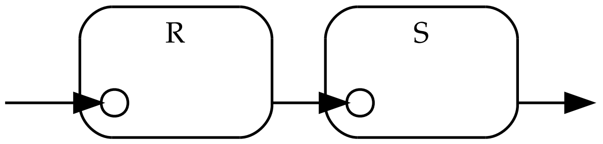


Figure 4: Concatenation RS

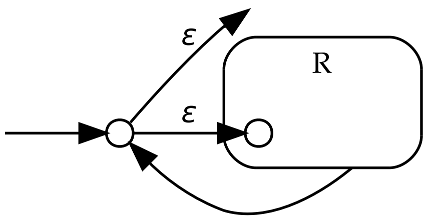


Figure 5: Repetition R*

be in a form where the meta characters and the literals are presented in the right order. Regular expressions with for example | can not simply be read from left to right and be converted correctly. The problem with the alternation operator is that it is an infix operator, so we only have the left hand side and not the right hand side when we read the | and can therefore not complete the fragment. Converting the regular expression to reverse polish notation, with an explicit concatenation operator, or making a parse tree will solve these problems. For this project neither is chosen.

A third solution to this problem is maintaining a stack where fragments and operators are pushed and popped. This is the method that is implemented. We tried determining the quality of the decision by comparing run times with Russ Cox's example code [?]. This did not go well due to several reasons. The main reason is that the example code does not do well on large examples[2] and large examples is needed to do a reasonable comparison.

Properties NFAs created with Thompsons method has these properties:

- At most two edges is leaving a state
- There are no edges leaving the accepting state
- There are no edges leading into the starting state
-

Example 3 (Converting a regular expression to a NFA) In this example we will be converting the regular expression `a*b|a*c` to a NFA using Thompsons method.

Fill out this example

4 Matching

The NFAs generated as described in section 3 on page 4 can be used to match a regular expression with a string, i.e. to decide if a string belongs to the language of the regular expression. Once the NFA is generated this is a straightforward task. At the beginning only the start state belongs to the set of active states. The string is read from left to right, taking each character in turn. When a character is read from the input string, all legal transitions from the states in the active set is followed. A transition is legal if it is a ε transition or if the mark on the transition matches the character read from the input string. The new set of active states is the set of end states for the transitions followed. If the accepting state is included in the active set when the string is read, the string matches the regular expression.

[2]There are constants in the source code and a naive list append function

Example 4 We will continue with example 3 on the previous page. We will again look at the regular expression a*b|a*c

fill out this example

4.1 Protocol specification

In this section we will define a protocol that can communicate information between our processes. The information consists of the mixed bit-values generated by the NFA simulator and the filters. The protocol should enable us to recreate paths taken through an NFA. To this purpose we need the protocol to support the following actions:

| Each time a character from the string is read, we put a |

: Whenever we change channels we put a :. There may be more than one or perhaps even no bits output on a channel for any given character from the string

= Copying of a channel. One channel is split into two, the paths taken through the NFA will be identical up to the point of splitting. The newly created channel is put in front of the rest of the channels

0,1, $\backslash a$ The actual bit values. The character classes needs extra care, we need to know on which character we passed them, so we put the value we pass on escaped in the output

b A channel is abandoned

t A channel has a match

Example 1 (Protocol) In figure 6 on the following page we have an automaton for regular expression `a*b|a*c`. This is matched against the string `aab`.

Initial step: Initially the start state of the automaton is added to the current list. All ε-edges are followed and the following is output:

1. Node 1 is a split-node, a = is output, and we follow the ε-edge to node 2 and output a `0`.
   Output so far: **:**`=0`.
2. Node 2 is also a split node, a = is output, and we follow the ε-edge to node 3 and output a `0`.
   Output so far: **:**`=0=0`.
3. Node 3 is not a split node, so we go back and follow the ε-edge from node 2 to 4 and output a `1`.
   Output so far: **:**`=0=01`.

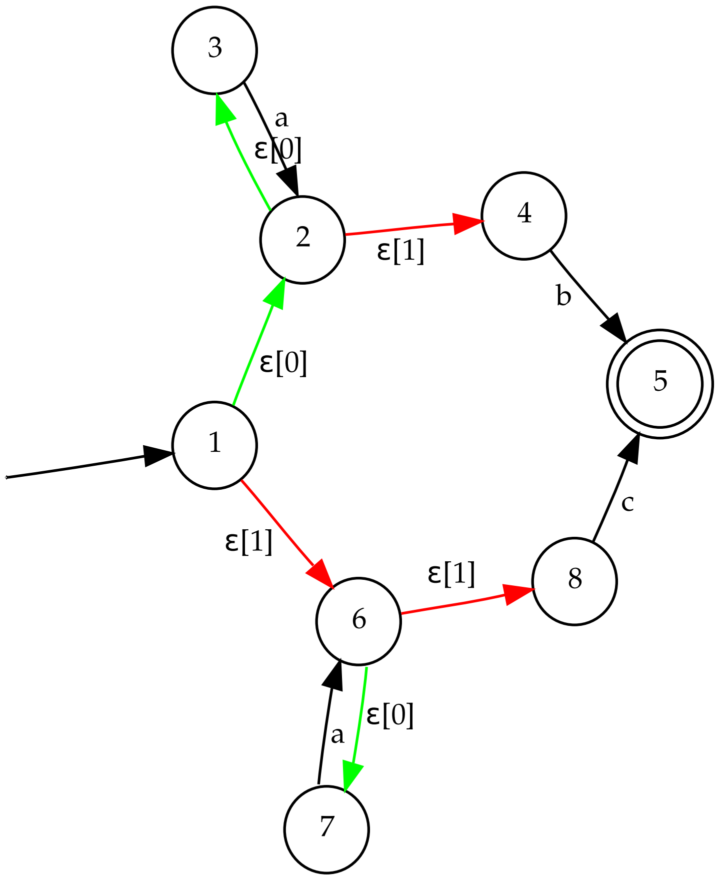


Figure 6: Automaton with bitvalues for regular expression `a*b|a*c`

4. Node 4 is also not a split node, so we go back follow the ε-edge from node 1 to node 6 and output a `1`

   Output so far: `:=0=011`.

5. Node 6 is a split node, a `=` is output, and we follow the ε-edge from node 6 to node 7 and output a `0`.

   Output so far: `:=0=011=0`.

6. Node 7 is not a split node, so we go back and follow the ε-edge from node 6 to 8 and output a `1`.

   Output so far: `:=0=011=0`. ∗

5 Filters

Here we will describe the developed filters.

5.1 A simple match

Input Any mixed bit-values or bit-values

Output A single value indicating match or no match.

This is a simple filter. The input is scanned for a `t` control character, if present we output a `t` otherwise we output a `b`. In the case of empty input, we will output an error message, this is because the empty input is most likely due to an error in the previous programs. To save time on processing, we will assume the input format is correct.

a Example 5 The regular expression a* matches the string *aaa*:

```
$ ./main "a*" "aaa" | ./ismatch
t
```

a* does *not* match *bbb*:

```
$ ./main "a*" "bbb" | ./ismatch
b
```

Since we do not check the correctness of the input, the sentence:

the cake is a lie □

which is clearly not in the correct input format with regards to the protocol defined section 4.1 on page 8, will also produce a positive answer from the filter:

```
$ echo "the cake is a lie" | ./ismatch
t
```

5.2 Materializing the result

Input Mixed bit-values with both matches and fails

Output Mixed bit-values with only matches

This filter outputs the bit-values associated with matches. There may be more than one match.

talk some more on the value of also having a filter that pics just one solution

Example 6

Give example

5.3 Grouping

Input Mixed bit-values

Output Mixed bit-values for rewritten regular expression

This filter facilitates reporting the content of groups. The filter outputs the mixed bit-values associated with the groupings. By this we mean that all mixed bit-values generated while inside a group should be sent to output and all mixed bit-values generated outside a group should be thrown away. By throwing away the unnecessary bit-values we hope to make the mixed bit-values sequence shorter. This will be an advantage when the time comes to apply the materialization filter, which is not streaming, described in section 5.2.

Example 2 (Simple groupings filter) Here we have a few simple examples of what the groupings filter should do.

For regular expression (a|b) matched with *a* the mixed bit-values are `=0:1|t:b`. Since the whole regular expression is contained in a capturing parenthesis, nothing should be thrown away. Output should contain `=0:1|t:b`.

For regular expression (?:a|b)(c|d) matched with *ac* the mixed bit-values are `=0:1|=0:1:b|t:b`. This time the first part of the regular expression is contained only in a non-capturing parenthesis and the associated bit-values should be thrown away. We want to keep only the bit-values from the second alternation. Output should contain `=:|=0:1:b|t:b`.

In this example we have only dealt with simple examples. Regular expressions containing parenthesis under alternation and repetition, e.g. (a)|b and (a)*, require extra care and will be discussed later. $*$

The output of the groupings filter can be used to navigate the NFA for the regular expression altered in a similar manner: Everything not in a capturing parenthesis is thrown away. From example 2 on the previous page we have the regular expression (?:a|b)(c|d), if we throw away everything not in a capturing parenthesis we have left (c|d). Stated in a more formal manner, we can define our first naive rewriting function G':

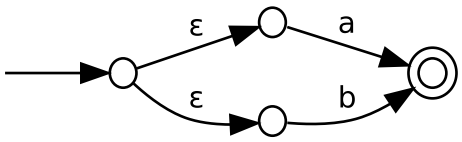
$$\begin{aligned}
G'[[\varepsilon]] &= \varepsilon \\
G'[[a]] &= \varepsilon \\
G'[[[...]]] &= \varepsilon \\
G'[[r'r_2]] &= G'[[r']]G'[[r_2]] \\
G'[[r'|r_2]] &= G'[[r']]G'[[r_2]] \\
G'[[r*]] &= G'[[r]] \\
G'[[r+]] &= G'[[r]] \\
G'[[r?]] &= G'[[r]] \\
G'[[(?:r)]] &= G'[[r]] \qquad (1) \\
G'[[(r)]] &= (r)
\end{aligned}$$

As is seen, G' basically throws away anything not in a capturing parenthesis. There are however a few problems with this definition, as hinted earlier. Our first problem is regular expression with a capturing parenthesis under alternation. When the capturing parenthesis is under the alternation and we throw away the alternation, we lose a vital choice: There is no longer a way to signal whether or not a group participates in a match.

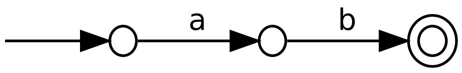
Example 3 (Capturing under alternation) In matching the regular expression (a)|(b), see figure 7(a) on the following page for the NFA, with the string *a* we obtain these mixed bit-values:

```
=0:1|t:b
```

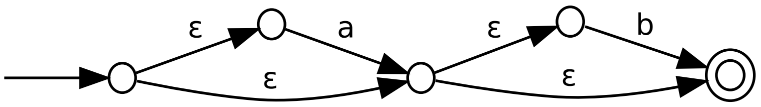
What these mixed bit-values are saying is that we have 2 channels, one that go through a and succeeds and one that go through b and fails. The succeeding channel never goes through b, the contents of that group is not defined.



(a) (a)|(b)



(b) (a)(b)



(c) (?:|(a))(?:|(b))

Figure 7: Capturing under alternation

Rewriting the regular expression (a)|(b) according to G' we have:

$$\begin{aligned} G'[[\text{(a)|(b)}]] &= G'[[\text{(a)}]]G'[[\text{(b)}]] \\ &= \text{(a)(b)} \end{aligned}$$

In this regular expression there is only one way: The one going through both the groups. See figure 7(b) for the NFA of the expression. This is bad news for our rewriting function and our filter, since we need some way of skipping groups: Each channel goes through only one group. $*$

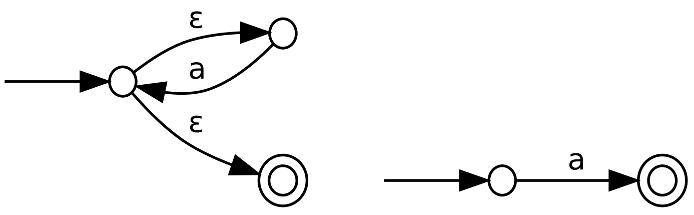
In example 3 we saw an example of how undefined groups are not handled. To solve this problem we need some way of signaling if a group participates in a match or not. We define a new rewriting function G'' it is identical to G' except for equation 1 which is changed to:

$$G''[[(r)]] = (? : |(r))$$

This change will enable us to choose which groups participates in a match. This comes at a cost: Extra bits will have to be added to the mixed bit-values output and extra alternations to the rewritten regular expression.

Example 3 (Continued) With the changed equation 1 we can continue our example from before. Again we rewrite regular expression (a)|(b), this time according to G'':

$$\begin{aligned} G''[[\text{(a)|(b)}]] &= G''[[\text{(a)}]]G''[[\text{(b)}]] \\ &= \text{(?:|(a))(?:|(b))} \end{aligned}$$



(a) (a)* (b) (a)

Figure 8: Capturing under repetition

See figure 7(c) for the NFA. As is clear from the rewritten regular expression and the NFA, there is now a way around the groups. Taking this into account, the output for the groupings filter should be:

```
=1:01|0t:b
```

What these mixed bit-values are saying is that we have two channels, one picks the route through a, around b and succeeds and the other picks the route around a, through b and fails. ∗

As needed, we now have a way of signaling if a particular group is in a match: Insert a 1 in the mixed bit-values and the group participates or insert a 0 and it does not.

Example 4 (Capturing under repetition) In this example we will be matching regular expression (a)*, see figure 8(a) for the NFA, with the string *aaa*. This match generates the following mixed bit-values:

```
=0:1|=0:1:b|=0:1:b|=0b:1t:b
```

First we have two channels, one that goes into the star and one that passes it by. While we can read an *a* from string we can go another round in the star, channels going to the accepting state will fail. After doing the transition on the last *a* from string, we have again 2 channels:

Rewriting (a)* according to G'' gives us:

$$\begin{aligned} G''[[(\mathsf{a})^*]] &= G''[[(\mathsf{a})]] \\ &= (\mathsf{a}) \end{aligned}$$

See figure 8(b) for the NFA. ∗

Definition 1 (The groupings filter rewriting function). For regular expressions r, r_1, r_2, defined over alphabet Σ, and a, any character from Σ, let G be defined by:

$$\begin{aligned}
G[[\varepsilon]] &= \varepsilon \\
G[[a]] &= \varepsilon \\
G[[[...]]] &= \varepsilon \\
G[[r_1 r_2]] &= G[[r_1]]G[[r_2]] \\
G[[r_1|r_2]] &= G[[r_1]]G[[r_2]] \\
G[[r*]] &= G[[r]] \\
G[[r+]] &= G[[r]] \\
G[[r?]] &= G[[r]] \\
G[[(?:r)]] &= G[[r]] \\
G[[(r)]] &= (?:|(E))
\end{aligned}$$

♣

Example 5 Here follows a few examples of how the groupings filter rewriting function, G, works.

$$\begin{aligned}
G[[(\mathsf{a})|\mathsf{b}]] &= G[[(\mathsf{a})]]G[[\mathsf{b}]] \\
&= (?:(\mathsf{a}))\varepsilon \\
&= (?:(\mathsf{a}))
\end{aligned}$$

$$G[[((\mathsf{cup})\mathsf{cake})]] = (?:|((\mathsf{cup})\mathsf{cake}))$$

$*$

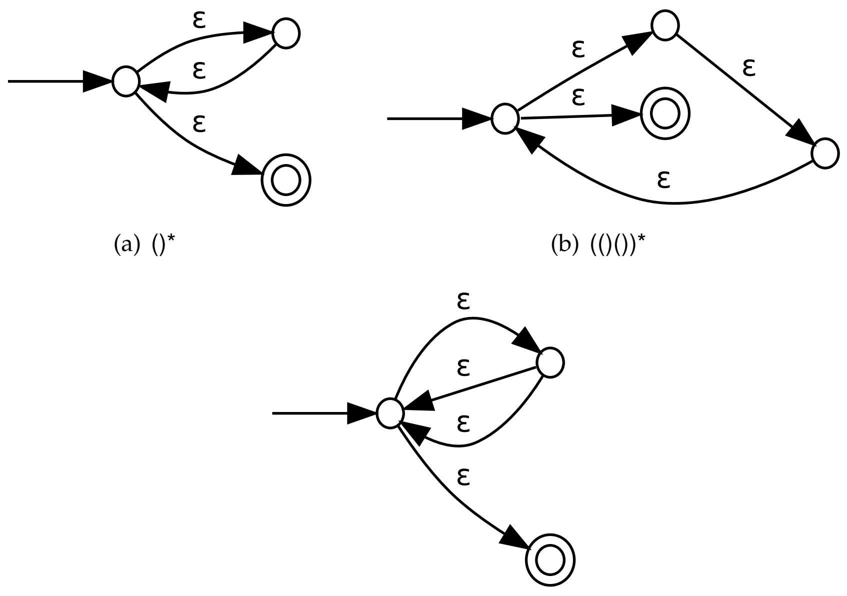
6 Implementing a regular expression engine

The task of implementing a regular expression engine can be undertaken in steps. The first steps is converting the regular expression to a NFA. The next step is to simulate the NFA. The last step in our implementation is to build filters.

6.1 The simulator

6.1.1 ε-cycle detection

When matching regular expressions like ()*, see figure 9(a) on the next page for the NFA, with the empty string the simulator will go into an infinite loop generating the following mixed bit-values:



(a) ()*

(b) (()())*

(c) (|)*

Figure 9: ε-cycles

```
=0=0=0=0=0=0=0=0=0=0=0=0=0=0=0=0=0=0=0=0 . . .
```

This is not desirable behavior and we need to stop the simulation before it goes into an infinite loop.

From [?] we have the depth-first search (DFS) algorithm. This algorithm can be modified to detect cycles. In short it works by initially marking all vertexes white. When a vertex is encountered it is marked gray and when all its descendants are visited, it is marked black. If a gray vertex is encountered, then we have a cycle and do not need to explore further on this path. The algorithm terminates when all vertexes are black. The algorithm will terminate, as we color one vertex each step and we always color the vertexes darker.

To see why this algorithm detects cycles, suppose we have a cycle containing vertex a. Then a is reachable from at least one if its descendants. When we reach a from this descendant, it will still be colored gray, since we are not done exploring a's descendants. Thus the cycle is detected.

Our problem is slightly different: We need to detect if we are in a cycle of ε-transitions. The DFS algorithm solution is still applicable, with slight modifications, as we do a depth first search when we explore the ε-edges. There will be no white states. Instead we will have a counter that is incremented every time a character is read from the input string. Every time a state is encountered it is stamped with the counter. We can only trust the color of the state if the counter and the stamp are identical. The gray and the black states work in much the same way.

In figure 9 on the preceding page we have some of the NFAs we encounter. We have an example of a long cycle in figure 9(b), more parenthesis adds more ε-transitions. We also have an example of how more channels can be created in the loop in figure 9(c) by adding alternations.

6.2 Data structure for the NFA

How to keep track of how many times i have matched a star.

6.3 Compiling regular expressions to NFAs

The NFA will be represented as a linked collection of `State` structures:

```
struct State
{
  // The type of the node: split, accepting, literal or range
  int type;
  // If type is set to literal, this contains the value of the literal
  int c;
  // For freeing the nfa
  enum Boolean is_seen;
  // The parenthesis count
  int parencount;
  // If type is set to range, this will show if the range is negated
  enum Boolean is_negated;
  // If the type is set to range, this contains the pointer to the
  // Range structure
  struct Range *range;
  // If the type is set to split, literal or range, this contains a
  // pointer to a following state
  struct State *out;
  // If the type is set to split, this contains a pointer to a following state
  struct State *out1;
  // Flag to ensure we only add each state once to the next list in
  // the simulation
  int lastlist;

  // For debugging
  Agnode_t *n;
  int id;
};
```

Perl uses bitmaps for deciding membership of characterclasses. There is also lists of ranges and balanced binary trees of ranges, the latter which RE2 uses.

7 Conclusion