Regular Expressions - a Graphical User Interface

Stefan Kahrs
University of Kent at Canterbury
Computing Laboratory
email: smk@ukc.ac.uk

September 13, 1999

Abstract

Regular expressions are a standard tool for language descriptions, in use for scanner generators and various search engines. Therefore, they and their connection to finite automata is an ubiquitous part of any computer science syllabus.

This paper is about a graphical user interface that demonstrates the translation of regular expressions into nondeterministic finite automata. We discuss the design choices of this tool and relate it to other translation techniques.

1 Introduction

Regular expressions are so widely used that they barely need an introduction. In CS teaching, they are traditionally associated with courses on compiler construction largely because formal language theory has become a bit too unfashionable on its own. Similarly, textbooks on the subject always include a section on regular expressions and regular languages, and explain how they can be translated into a corresponding finite automaton.

The translations of regular expressions are typically described in a graphical way, one translation rule for each connective. One could even view these rules as formal graph grammar rules, e.g. in the style of [HK87], which shows that they follow a very strict pattern.

Generally, regular expressions are just about the simplest part of a Compiler course. There should not be any need to lose even the weakest students on that subject, or to spend more than a tiny amount of time explaining them in lectures. Especially, the graphical form of the translation rules should help.

However, it is not quite as simple as that in practice. The rules in the literature deal with the general case, they make worst-case assumptions of what could happen elsewhere, and consequently they create unnecessarily large numbers of states and even more unnecessarily large numbers of ϵ-transitions. As a result, this is unnecessarily confusing, even more so since situations that require all

this overhead do not arise very often. Apart from the confusion, a small NFA makes a subsequent translation into a DFA less daunting.

In examinations on this subject, when we ask students to translate a regular expression into a finite automaton, we can observe two kinds of perfect answers:

- students who have slavishly memorised the rules and include every silly ϵ-transition the translation demands
- students who have understood the concept of NFAs and draw the NFAs by hand and intuition

Amongst the imperfect answers we often find variations of the actual rules, except that students often either forget the odd ϵ-transition or draw a few too many.

In short, it would be nice to perform a more sophisticated translation that saves states and transitions and looks less mysterious when applied to concrete examples because all generated ϵ-transitions have a purpose. On the other hand, we normally would not want to dwell on these sophistications in lectures (or even textbooks). I implemented a graphical user interface that applied such slightly more sophisticated transformation rules which keeps the set of states and transitions small. The implementation is a Java applet (made Java 1.1 compliant to increase browser compatibility) located at

`http://www.cs.ukc.ac.uk/people/staff/smk/regexp/gui.html`

The user types in a regular expression which is then transformed step by step into the corresponding NFA; each transformation step is invoked by a button press. The idea is for students to play with this tool in their own time to get an intuitive understanding about regular expressions and NFAs.

An important motivation of the applet was to avoid the creation of NFAs with too many unnecessary states and unnecessary ϵ-transitions. This is different from the `jflap` tool [GR99] which supports (since version 3.1, August 1999) a translation of regular expressions into NFAs as well, but which sticks to a much simpler translation scheme.

2 Strategies of translation

Although the flavour of the translation rules is always the same, there is a considerable variation in the literature about which rules to use. There is also the question of the general strategy of the rules. The two main strategies are:

- top-down; this strategy starts with an edge labelled with a regular expression and then applies the rules step by step until no further refinements are possible
- bottom-up; here, first the atomic components of the regular expression are translated into NFAs — the rules say how these NFAs are then combined.

The most commonly used form in the literature is bottom-up [ASU86, TS85, Mar91, Par92], but top-down has some supporters as well [WM95, PP92].

For hand-drawing purposes, the bottom-up strategy is best, for two reasons: first, it does not require the erasure of earlier drawn bits; second, the graphs grow outwards which means that one usually avoids overcrowding problems in the middle of the drawing area.

For a graphical user interface the choice between these two strategies should be the other way around. The reason for this is that a bottom-up strategy works (at least conceptually) with regular expressions which have finite automata *as their components*, while top-down simply requires regular expressions as labels of edges. The former is a considerable challenge[1] when we finally face the question: how do we draw it? The latter is not.

Still, the difference between both strategies is largely superficial, as it is usually easy to derive a top-down version from bottom-up rules and *vice versa*.

There are other possible variations one can employ:

- Appel uses in [App98] a different (also bottom-up) translation scheme that does not translate into NFAs, but into something one could call "NFA with a half-loose initial edge". One can create an NFA from the final result of his translation by connecting the loose end of the half-loose edge to a fresh initial state. The advantage of his scheme over the one in [ASU86] is that it needs fewer states — the sub-automata can often be linked with the loose edges directly, instead of having intermediate states and mediating ϵ-transitions.

- Bernstein recently invented an allegedly new translation strategy for regular expressions [Ber99] which also keeps the number of states relatively small (roughly equal to the size of connectives). This strategy appears to be completely different as it uses regular expressions as *states*. The great appeal of this strategy is that the language of a regexp-state is exactly the language of the NFA were this state the sole accepting state.

For graph-drawing purposes, Bernstein's translation has the big disadvantage not to be fully compositional. With this, I mean that a refinement step could be forced to draw a new edge between two states which are located in completely different parts of the graph. However, it is possible to view his rules as edge-replacement rules; the non-compositionality can be solved for the price of (possibly) generating a few more states, without affecting his generic claim about the number of generated states.

Appel's rules, when translated into the world of edge-replacements, are surprisingly similar to Bernstein's, the only difference being a slightly more "expensive" rule for the choice operator.

[1] Notwithstanding, JFLAP 3.1 uses this strategy — with less than perfect results.

3 The Rules

For our purposes we stick to the presentation of rules as edge-replacement rules, i.e. we follow a top-down approach.

Sets of transformation rules usually work with some kind of invariant that is maintained throughout, typically for correctness purposes and to contain state-growth. One invariant almost[2] every translation system uses is to restrict itself to a *single* accepting state and we will do the same. So, in the following the term "NFA" will implicitly carry this restriction.

The other used invariants are safety criteria for the initial and accepting states. An NFA is called *in-safe* if its initial state has no incoming transitions, and dually it is called *out-safe* if its accepting state has no outgoing transitions. It is *safe* if it is both in-safe and out-safe. Notice that we can easily make an NFA in-safe by adding a new initial state that is prepended with an ϵ-transition to the old one. An analogous argument applies for out-safety.

Many of the translation schemes work by maintaining safety as an invariant, i.e. their rules always produce a safe NFA, and the fact that they do can be exploited for the connectives. This applies for example to the transformation rules in [WM95, ASU86, TS85].

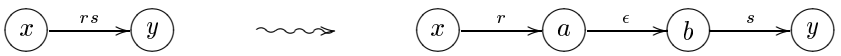
One can fine-tune the translation process a little by working with a weaker invariant. Appel [App98], Bernstein [Ber99], Martin [Mar91], and JFLAP [GR99] all use schemes that only guarantee in-safety.

3.1 Sequencing

Consider for example the very simple transformation rule below for treating sequential composition of regular expressions.

What could be simpler and more obvious than this rule? However, even this simple rule involves assumptions about safety: It is *correct* if the replacement of "r" is out-safe or the replacement of "s" is in-safe. It is in-safe if "r" is, and out-safe is "s" is. These are fairly weak requirements and therefore most translation schemes use this rule. In particular, any translation scheme that has otherwise either in-safety or out-safety (or both) as an invariant can use this rule, because it is not only correct but maintains the invariant.

There is a safer rule though, used e.g. in [PP92, Lou97, HU79]:



This still inherits the in-safety from "r" and the out-safety from "s", but it does not require either safety of "r" or "s" for its own correctness. Bottom-up translation schemes that permit multiple accepting states (e.g. [GR99, Mar91])

[2]The only exceptions I know are [Mar91, GR99].

also use this rule to avoid copying regular expressions — they would otherwise need one copy of s for every accepting state of r. Tremblay & Sorenson go super-safe[3] by further adding *epsilon*-transitions at beginning and end.

If we want to contain state-growth we clearly would like to apply the first rule wherever possible. If we employed a translation scheme that preserved safety (or just in-safety) then we could use that rule in every case.

The applet does not maintain safety, not even in the weaker forms of in-safety or out-safety. What it does instead is to label edges with safety requirements. In a slightly simplified form this goes as follows:

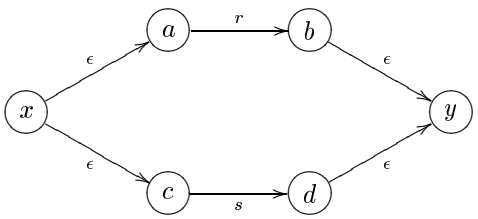
The safety requirement that "s" is in-safe is expressed here by double-lined connection. This also means, of course, that the arrow we started with could have had its own safety requirement. We have to deal with that, e.g. by providing an additional rule:

There is a dual inheritance of an out-safety requirement (so far we do not have any but they show up later) which goes to the second edge.

However, this is not 100% ideal. For our initial rule we had the choice imposing either in-safety for "s" or out-safety for "r". If we are unlucky in our choice then we create one state and one ϵ-transition too many, i.e. if the safety criterion we chose to impose is violated while the one we did not simply holds. The implementation addresses that by using the safe rule whenever r and s are likely to clash, i.e. when r ends with an iteration and s begins with one.

3.2 Choice

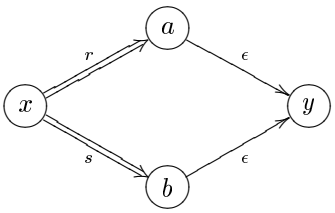
For the replacement of a choice edge $x \xrightarrow{r|s} y$ there are quite a few different possibilities available. The Dragon book plays it very safe by translating into:

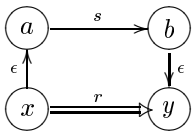


This is *very* safe as it is correct and safe without any safety requirements for "r" and "s". This is the most commonly used rule in the literature [ASU86, TS85, PP92, Lou97, Par92, Mor98, HU79].

[3] Quite frankly, this is mad. Unsurprisingly, they optimise quite a few ϵ's away when they demonstrate the translation at an example.

The price paid for this amount of safety are four extra states and four extra ϵ-transitions. Bernstein and Sippu & Soisalon-Soininen translate two of these into safety requests (which in both cases are invariants of their respective translation schemes):

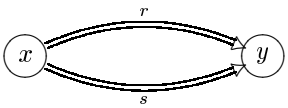




The rule from [SSS88] (on the right) imposes both in-safety and out-safety. The out-safety request is indicated graphically by the closed arrow-head.

Bernstein's rule (on the left) propagates and demands in-safety. Appel, Martin and JFLAP use slight variations of this rule: Appel [App98] makes it in-safe by prepending an ϵ-transition at the beginning; Martin and JFLAP [Mar91] save the out-going ϵ-transitions and simply permit many accepting states.

In fact, we can eliminate *all* extra states and ϵ-transitions for the price of safety requests. This is particularly useful if the translation scheme propagates full safety anyway.

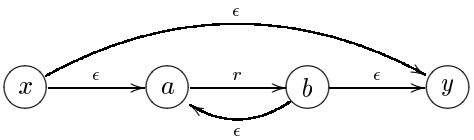


Wilhelm & Maurer's translation scheme creates safe NFAs, and so they can (and do) use this rule. My applet uses this rule as well; since it does not always generate safe NFAs it creates instead the corresponding safety requests, i.e. full safety for both r and s.

Most bottom-up schemes do not use this rule even if they create safe NFAs, for the simple pragmatic reason that a bottom-up strategy faces the awkward scenario (for hand-drawing purposes) that the initial and accepting states of two automata need to be identified.

3.3 Iteration

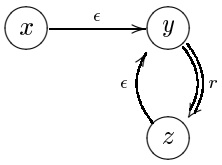
This is perhaps the trickiest[4] of all the cases, translating $x \xrightarrow{r^*} y$. Most books go here for the very safe version:



[4] An early version of Appel's book even used an unsound translation.

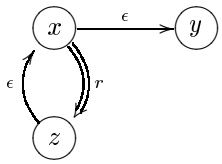
This is correct and safe without any safety requirements for "r". Seppu & Soisalon-Soininen vary this rule slightly by having an ϵ-transition from a to y instead of x to y — this is only sound if r is in-safe.

Bernstein has a niftier translation scheme (also used by Appel and Martin). All his states *are* regular expressions and we can reach one of these states s from the initial state with a word w only if w is in the language[5] of s. Therefore, y must be equal to xr^*, and all we need is an additional state xr^*r with the rather obvious transitions between those states derived from the universal properties of r^*. Long story, short transformation rule:

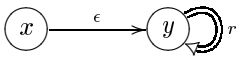


Here, the fresh state z stands for the language xr^*r. As a general rule (outside the context of Bernstein's translation scheme) this is correct if "r" is in-safe, and it is itself in-safe (and not out-safe). JFLAP uses a slightly weird variation of the rule in which not y is the accepting state but x and z are.

The above is a simple rule for the propagation of in-safety; there is a dual one which produces an out-safe NFA:

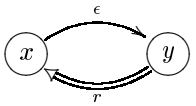


One can do slightly better than these rules by turning one of the ϵ transitions into a safety requirement. Here is such a rule which also generates in-safety:



I did not chose rules of that kind for the graphical user interface, because these little self-loops are rather cumbersome for drawing purposes. In which direction should they go, what should be the radius of the loop, how do we carry on if r is further refined? If these choices are made badly then the layout is quickly ruined.

For simplicity (fewer case distinctions and simpler graph layout) I did not use Bernstein's rules either, instead I went for the following:



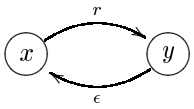
[5] In our setting the word could contain itself regular expressions in which case the condition is that the language of w is a sub-language of s.

$$
\begin{array}{rcll}
(r^*)^* & = & r^* & \\
\epsilon^* & = & \emptyset^* & \\
(p|q)^* & = & (r|q)^* & \text{if } p^* = r^* \\
(pq)^* & = & (p|q)^* & \text{if } p \text{ and } q \text{ are nullable} \\
p|\emptyset & = & p & \\
p^*|q & = & p^+|q & \text{if } q \text{ is nullable} \\
p^*|q & = & p^+|q|\epsilon & \\
p|\epsilon & = & p & \text{if } p \text{ is nullable}
\end{array}
$$

Table 1: Optimisations

This rule is correct with the indicated (full) safety requirements for "r". If there were no safety requirements for the edge we started with this rule is preferable to Bernstein's anyway as it creates fewer states. Otherwise, Bernstein's rule is slightly better as it generates one safety requirement fewer.

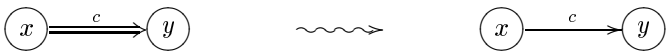
Interestingly, an apparently similar rule for r^+ can drop its safety requirements entirely:



Based on this, it is clear that the ultra-safe rule for r^*, mentioned at the beginning of this section, can be seen as resulting from the equality $r^* = (r^+|\epsilon)$.

3.4 Atomic Regular Expressions

The atomic regular expressions, i.e. characters, ϵ, and $\emptyset$ always produce safe connections. The only refinement rule is the removal of any $\emptyset$-labelled edge. Apart from that, safety constraints are removed.



Translation schemes which are not based on edge-replacement [Mar91, App98] can do slightly better with ϵ-regexps as they correspond to single-state NFAs.

3.5 Optimisations

The applet also employs (optionally) a number of optimisations and simplifications, especially in connection with the iteration operator. The most significant ones are listed in table 1.

These equations can be seen as conditional term rewrite rules, going from left to right, as this is the direction in which we save states and transitions (or even weaken safety requests only). This is not necessarily the case concerning the rule $p^*|q = p^+|q|\epsilon$ which is therefore only applied for certain p. Example: the normal forms of $(a^*(b|c)^*)^*$ and $(p^*h)^*|b^*$ are $(a|b|c)^*$ and $(p^*h)^+|b^+|\epsilon$, respectively.

The user of the applet can switch the optimisations on and off, even in the middle of the transformation process. The reason for giving that flexibility is that on the one hand the combined effect of the optimisations can be rather puzzling for the uninitiated, on the other their complete absence would too easily create overcrowding problems in the drawing area.

3.6 Counting States

Since most sets of translation rules work in a rather context-free manner, it is fairly easy to compute the number of states they generate. This number can normally be expressed as $a \cdot m + b \cdot n + c \cdot p + 2$, where m, n, and p are the number of choice operators, iteration operators, and sequencing operators in the original regular expression, and a, b and c are constants that depend on the translation scheme. The reason for the "+2" is the initial scenario of the top-down translation which has already one initial and one accepting state.

Here is a table giving these constants for some of these schemes: The table is

Source	choice	iter	seq	char	plus	example
[TS85, Mor98]	4	2	4	0	2	28
[HU79, PP92, Lou97]	4	2	2	0	2	22
[ASU86, Par92]	4	2	1	0	2	19
[Mar91, GR99]	1	1	0	2	0	17
[SSS88]	2	2	1	0	2	15
[App98]	2	1	0	1	0–1	13
[Ber99]	2	1	1	0	2	12
[WM95]	0	2	1	0	2	11
this	0	0–2	1	0	2	9

Table 2: Computing the state numbers

to be read as follows: if the original regular expression has m (binary) choices, n iterations, p (binary) sequences and q single-character regular expressions then, say, the translation scheme in [ASU86] produces an NFA with $4m + 2n + p + 2$ states. The column on the right shows this number for the sample regular expression $(b|a^*x)y|z^*m^*$.

The reason the table has 5 central columns (rather than 3) is that not every scheme can be seen as an edge-replacement system, some are intrinsically bottom-up. This applies in particular to [Mar91] and [GR99] for which the number of states of the generated NFA is $m + n + 2q$, where q is the number of single-character regular expressions. Also, Appel's NFA-mutants are just a wee bit different.

The ranges 0–1 and 0–2 in two of the entries indicate "it depends on the context". For the applet, this number depends on the inherited safety requirement. Moreover, the entries for the applet (and also Bernstein's translation) do not take the optimisations into account, e.g. a sequence operator inside an iteration may be optimised away by the applet, and Bernstein's translation implicitly exploits laws such as $(rs|rt) = r(s|t)$.

4 Implementation Details

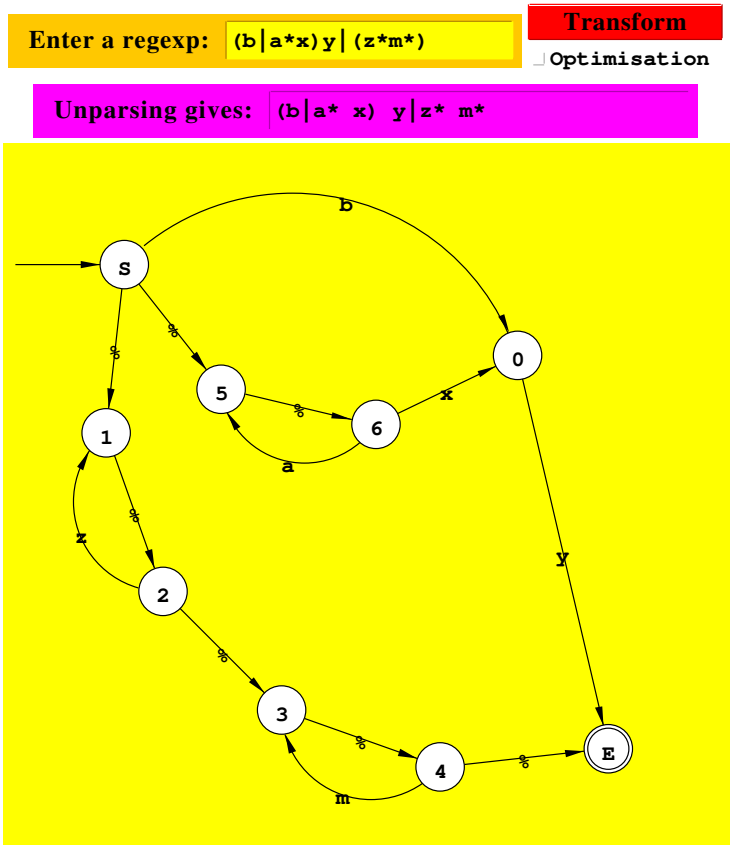


Figure 1: Program appearance

The picture shows the general appearance of the applet after a few steps. One regular expression (here: $(b|a^*x)y|z^*m^*$) has been typed in and completely transformed in several steps.

regexp	→	sequence
regexp	→	regexp | sequence
sequence	→	iteration
sequence	→	sequence iteration
iteration	→	primary
iteration	→	iteration *
iteration	→	iteration +
primary	→	%
primary	→	@
primary	→	(regexp)
primary	→	character

Table 3: Input Grammar

4.1 The User Interface

The applet has four components: three panels, and a canvas. The first (orange) panel asks from the reader to input a regular expression. The text of the input field of this panel is passed to a syntax analyser (a hand-written LL(1) parser). The second (magenta) panel then displays either an unparsed version of the input (which could have fewer parentheses) in its textfield, or an error message if the input was syntactically incorrect. The third panel contains the **transform** button and the **optimise** check-box.

Table 3 describes the recognised syntax for regular expressions. The terminal "character" stands for any alpha-numerical character. The % represents ϵ, the @ represents $\emptyset$. Spaces and tabs are allowed in the input but ignored.

The canvas is the drawing area of the applet. Its size can be varied by modifying the applet's dimensions in the accompanying HTML file.

Initially the canvas displays two states, S and E, the initial and accepting states of the NFA, plus an unmarked edge going from S to E. When a regular expression has been successfully parsed it becomes the label of this edge. By clicking the "transform" button an edge-replacement rule is invoked and the graph changes accordingly. The redexes are chosen in a breadth-first manner.

A safety requirement of both in-safety and out-safety is indicated by displaying the arrow in blue instead of black. Slightly different shades of blue are used to indicate in-safety or out-safety only. These weaker requests are only generated when the optimisation is switched off to cover for the case that the optimisation is switched on during the translation process.

4.2 Optimisation

If the optimisation check-box is switched on, the regular expressions are first "optimised" before the transformation process can proceed. This optimisation replaces regular expressions by normal forms w.r.t. the rewrite rules mentioned earlier, plus a few others such as $\emptyset r = \emptyset = r\emptyset$, etc.

If the optimisation is switched off then some edges labelled with choice-regexps have specific (context-dependent) safety criteria imposed on them. There is a subtle reason why this is necessary here and not in the presence of optimisation rules: the edge-replacement rule for choice will always produce a safe NFA, therefore any safety imposed on a choice-edge appears to be redundant.

However, the user is allowed to switch optimisations on in the middle of the transformation process, in which case all current regexp-labels are optimised. In this situation a choice-regexp can be replaced by a regular expression of a different kind and in this situation the imposed safety criteria do matter.

If, on the other hand, such edges with choice-regexp labels are created when optimisation is switched on then the above situation cannot arise: these edges are already labelled with normal forms and choice normal forms are intrinsically safe.

4.3 Graph Layout

The graph layout is moderately safely computed — it is possible for nodes and edges to be placed at very awkward places, causing overcrowding, but there is a check that ensures that the canvas area is never left.

New nodes are only ever created by splitting a (possibly curved) edge into segments, e.g. if we refine an edge labelled "abc" then we need two intermediate nodes, the centres of which are placed at equal distances along the original edge.

An edge labelled with e.g. "abc" or "$a|b|c$" is refined in a single step, giving equal room for all its components. It simply looks nicer to give each of a, b, and c a third of the available space each instead of giving a half, and b and c subsequently a quarter each. It also makes overcrowding less likely.

All curved lines are drawn using Java's `drawArc` method, more specifically they are all segments of a circle[6]. Java 1.1 provides little support for dealing with circle segments, in particular affine transformations are lacking (supported in Java 1.2 but not here). They were re-implemented and used quite extensively, e.g. for equipping edges with arrow-heads.

Initially there were some problems with printing the picture, i.e. wrong circle segments were drawn and extra lines appeared on the print-out. The latter was a known Java-bug (number 4,076,291), the former a (then) unknown one.

The check that the drawing stays on the canvas works as follows. The only risk to leave the canvas is from drawing curved arrows for alternatives and iterations, because all newly generated nodes lie on previously generated lines[7]. The check for new arcs first computes all intersections of the circle with the canvas borders and then tests whether any of these points is on the arc. Since we want to be able to draw nodes along such lines (which have some space

[6] The `drawArc` method supports the drawing of ellipse-segments, but unfortunately these ellipses cannot be turned, i.e. the line through the two focal points of the ellipse is always parallel with either x-axis or y-axis.

[7] This is one of the major reasons for choosing the earlier described translation scheme for r^* and r^+.

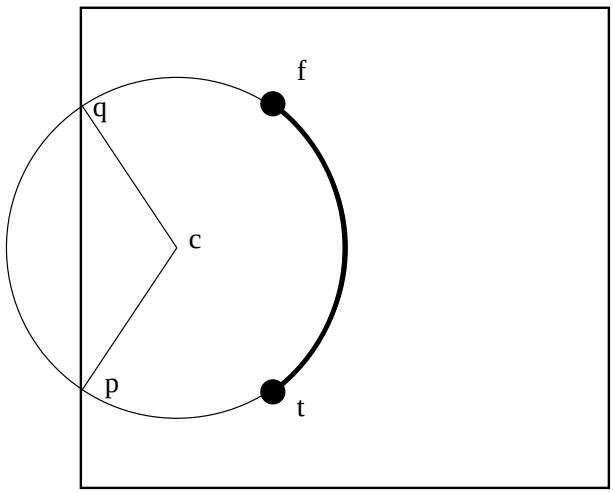


Figure 2: Canvas Check

demands of their own) all canvas borders are reduced by the (fixed) radius of nodes.

Computing the intersections of a circle with the bounding area of the canvas is fairly simple, especially as the border lines are parallel with either x-axis or y-axis. For example, the circle $(x-c_x)^2+(y-c_y)^2 = r^2$ (with centre $C = (c_x, c_y)$ and radius r) intersects the vertical line $x = a$ if and only if $r^2 - (a - c_x)^2$ is positive — in which the case the two intersections are $(a, c_y \pm \sqrt{r^2 - (a - c_x)^2})$. To check that such a point P is on the arc, we compute the angle between x-axis and the line CP.

An example can be seen in figure 2: we want to draw the (bold) arc from f to t. The circle this arc belongs to has two intersections with the canvas area (the rectangle), points p and q. Since neither the line cp nor the line cq is in the arc's segment of the circle it is clear that this drawing stays on the canvas area and is therefore permitted.

5 Summary

The implemented applet is a nice little toy for students to get to grips with regular expressions and finite automata. Because it makes a lot of use of context information it can operate with more sophisticated translation schemes than the ones in the literature. In particular, it produces fewer states and fewer transitions.

This has two advantages: first, the students see a bit of relation algebra in action, and second, all these mystifying useless ϵ-transitions which the known translation schemes so happily generate are omitted by the applet. If there is

an ϵ-transition then there is a good reason why it is there.

There is still room for improvement though, e.g. it is possible to reduce the number of states even further (especially by supporting transitions from a state to itself), and one could implement further optimisations, such as $(rs|rt) = r(s|t)$. Also, the placement of new edges and nodes on the drawing surface could be made "intelligent", i.e. aware of the current state of the drawing, to avoid nodes being placed on top each of other or edges crossing nodes.

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers — Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Ber99] Daniel J. Bernstein. A universal pattern-matching automaton. University of Illinois at Chicago, Department of Mathematics, Statistics and Computer Science, 1999.

[GR99] Eric Gramond and Susan H. Rodger. Using JFLAP to interact with theorems in automata theory. In *30th SIGCSE Technical Symposium on Computer Science Education*, pages 336–340. ACM, 1999.

[HK87] Annegret Habel and Hans-Jörg Kreowski. Characteristics of graph languages generated by edge replacements. *Theoretical Computer Science*, 51:81–115, 1987.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Lou97] Kenneth C. Louden. *Compiler Construction.* PWS Publishing Company, 1997.

[Mar91] John C. Martin. *Introduction to Languages and the Theory of Computation.* McGraw-Hill, 1991.

[Mor98] Bernard M. Moret. *The Theory of Computation.* Addison-Wesley, 1998.

[Par92] Thomas W. Parsons. *Introduction to Compiler Construction.* Computer Science Press, 1992.

[PP92] Thomas Pittman and James Peters. *The Art of Compiler Design.* Prentice Hall, 1992.

[SSS88] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory*, volume 1. Springer, 1988.

[TS85] Jean-Paul Tremblay and Paul G. Sorenson. *The Theory and Practice of Compiler Writing.* McGraw-Hill, 1985.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design.* Addison-Wesley, 1995.