Compilers

Introduction:

In this document, we will be exploring a general high level understanding of compilers. As you are likely aware, there are various compilers for various languages. This is why we will be reviewing the core philosophy and principles implemented by the majority of compilers rather than an in depth low-level analysis of one particular compiler. Note that compilers and interpreters (while using many of the same techniques) are different enough that they are considered to be two different things.

Differences Between Interpreters and Compilers:

The aim of compilers and interpreters is the same: translating some input language, A, to some output language, B. The output language is always a lower level language ie. a more simplistic language. For example, C compiles to assembly, and Java compiles to Bytecode, which is a meta language that can be understood by the JVM. The primary difference between the two is that the interpreter is ran at runtime, interpreting and excecuting each line of the program one step at a time, whereas the compiler must compile the entire source code of the input language and store the output into files that take up actual hard drive space. If you’ve programmed in C you will be aware of this, since compiling your source files produces object files (.o) which are essentially machine code that can then be linked with external libraries to create an excecutable. Compiling code takes time, but the resultant program often runs much faster since the code can be optimized during compile-time. Interpreting a language does not take any initial time before the program is executed, however, the programs often run slower during runtime since the interpreter is interpreting the code into CPU instructions in real time, as well as excecuting those instructions all at once. One last difference between compilers and interpreters is that compilers often need to go through a linking phase to link external libraries; however, languages like python do not need to go through such a process since those libraries can be interpreted at runtime along with the source code.

Terminology:

There are a few terms that need to be defined when discussing compilers:

- **Programming Language:** A notational system for describing computations in human-readable and machine-readable form.

- **Language Processors:** Compilers are a subset of language processors. Other subsets of language processors include interpreters, text editors, text-to-speech, grammar checkers, etc.

- **Interpreters:** We’ve already discussed interpreters, but it should be noted that they are a subset of compilers (which are themselves a subset of language processors).

Language Paradigms:

Perhaps you’ve heard of one or more “language paradigms”. These are abstract concepts which are used to express a particular “kind” of language. One such paradigm, with which you are likely familiar, is OOP. Other kinds of paradigms include: procedural, functional, logical, parallel, scripting, markup, and specification. Here is a summary of each:

- **OOP:** Object-Oriented Programming languages are built primarily upon the concept of “objects”, namely, an objects attributes (how we can describe an object), which are represented as instance variables, and their behaviours (what sorts of actions the object can perform), which are represented as methods. They often benefit in bussiness-oriented environments due to the upholding of polymorphism as a central doctrine.

- **Procedural/Imperative:** Procedural languages (sometimes also called Imperative languages) are a bit of a higher level paradigm, since many languages that fit within other paradigms might also be considered as being procedural. Essentially, procedural languages specify “what to do” and “how to do it” in a step by step fashion (a procedure). Some procedural languages include: C, COBOL, FORTRAN, ALGOL, BASIC, and Pascal. Easier than understanding what qualifies as a procedural langauge is to understand which languages *aren’t* procedural. Languages that aren’t procedural include SQL, Prologue, and Lisp.

- **Functional:** Functional programming languages are related to mathematics and are almost always dynamically-typed. Since they do not support type safety (state), they rarely crash or have bugs. They also have other advantages such as efficient parallel programming, nested functions, and lazy evaluation. Such languages include R, Python, Erlang, Haskell, Clojure, Lisp, etc.

- **Logical:** Logical languages are based on formal logic. Rules are written in the form of logical clauses. The famous syllogism in philosophy goes as follows: Premise 1) all men are mortal; premise 2) socrates is a man; premise 3) therefore socrates is mortal. This modal logic follows the deductive predicate: P or Q; not P; therefore Q. This same logic is employed in logical programming languages as follows: man(socrates); mortal(X) :- man(X); mortal(socrates).

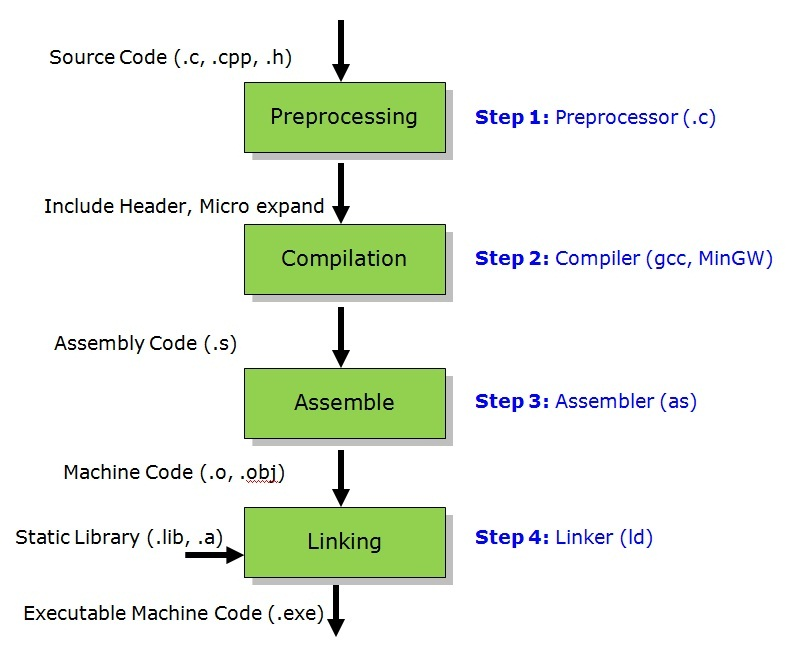
- **Parallel/Concurrent:** Parallel/Concurrent programming languages utilize threads to excecute various parts of a program in parallel ie. At the same time. Since parallel languages rely on this paradigm, programmers often need to take careful consideration as to which libraries they implement.

- **Scripting:** Scripting languages are often interpreted languages that run procedurally. While not always, they often rely on a system’s core utilities/binaries such as the GNU core utils, which are typically written in another language such as C. Bash, Batch, PowerShell, Visual Basic, and BASIC, are examples of scripting languages rely on pre-written system commands. However, languages like Python, R, Perl, Ruby, PHP, and JavaScript are all examples of standalone scripting languages.

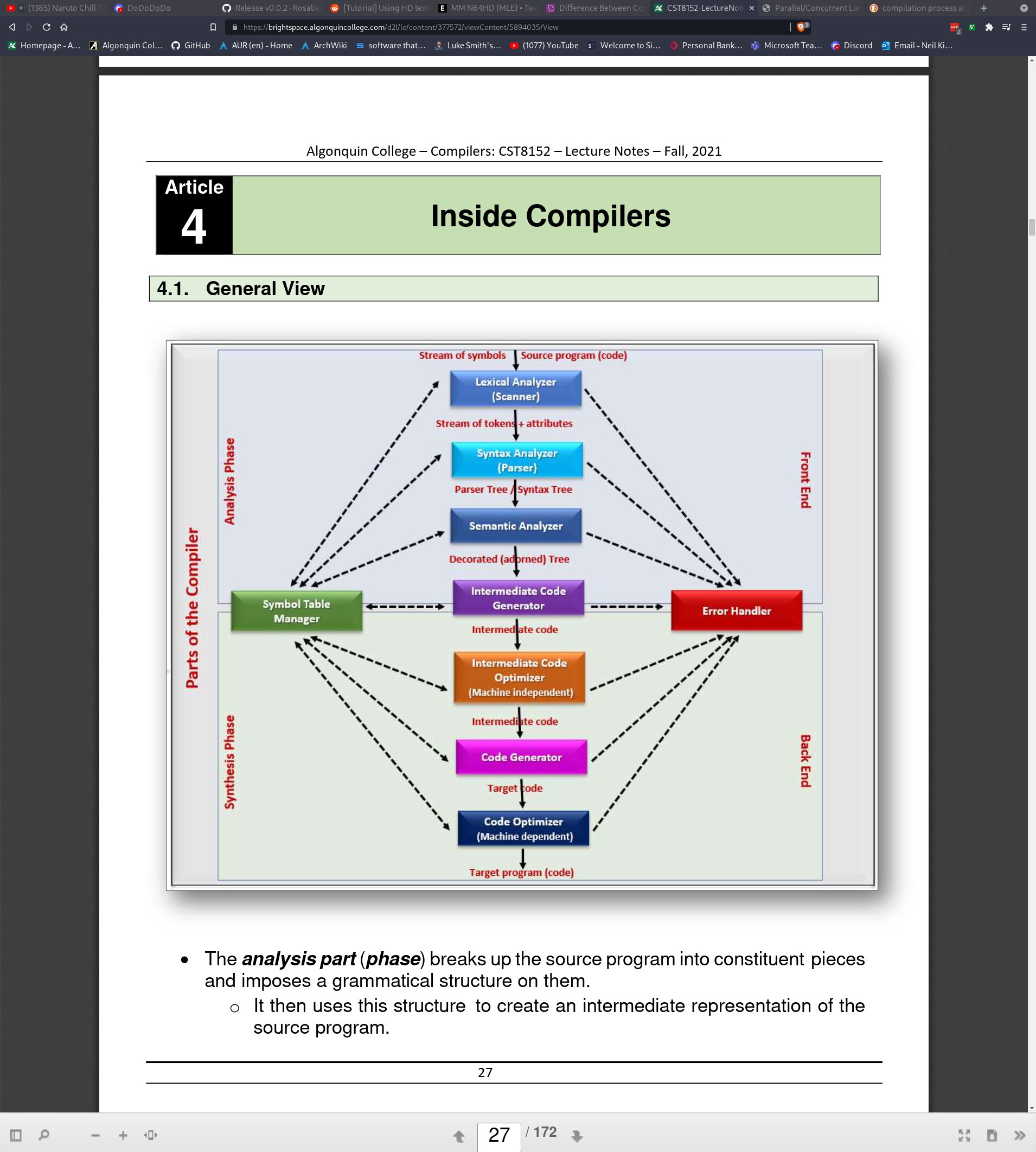
- **Markup:** Markup languages include things like SGML, HTML and XML. They typically use some sort of parent-child relationship between tags called elements. For instance in HTML, you might have a paragraph element wrapped inside a bold element. The bold element would be the parent of the paragraph element in this case. Markup languages contain no logical statements. For this reason, they are not compiled in the traditional sense, although we can “compile” markup languages into visual formats such as webpages or PDFs.

- **Specification:** Similar to functional languages, specification languages are often based on mathematical principles such as algebraic or model-theoretic structures that include sets of data with functions to operate on such sets. Honestly, the details of these go a bit over my head, and you probably wont have to deal with them in your life, so we’ll skip over it.

An Overview of the C Toolchain:  
 To kick things off, we’ll be looking at a basic overview of the C toolchain. The C toolchain, in case you weren’t familiar, is a collection of applications (which include the C compiler) that are ran in sequence with the aim of compiling code.

As I mentioned, the C compiler is only one stage of 5 in the C toolchain. As you can see in the diagram, we start with our source code (.c, .cpp, .h files), then run the preprocessor which will process any preprocessor directives as well as strip comments, then we compile our code with the C compiler into assembly code (this is the part that we’ll be focusing on), then the assembler will assemble the assembly code into object code, then we link with external libraries, and finally, the loader loads the program into memory as a process. I outline this process to show you that the compiler itself is not actually all there is to the story. While it might arguably play the biggest role in the pipeline, it is good to keep in mind that we are looking at a fraction of the process needed to convert code into an executable.

Internals of the Compiler:



The diagram above represents the compiler pipeline ie. the steps that the compiler goes through (not including preprocessor, assembler, linker, or loader). The internals of the compiler can be broken up in to 2 primary subcategories: The font end, called the Analysis Phase (top half, tinted blue), and the back end, called the Synthesis Phase (bottom half, tinted green). The Analysis Phase is concerned with breaking up characters into words or symbols called **lexemes** and creating an intermediate language that is not yet ready for execution. The Synthesis Phase is concerned with comparing the intermediate language with the symbol table to create our output language. In laymans terms, analysis can be though of as analysing the raw text, and synthesis can be thought of as generating the actual target language.

As you can tell from the chart, the flow goes from top to bottom, from the source code, to the target program. Technically, there is an additional step which between the source code and the lexical analyser we will be covering, called the buffer. Here is an overview of each step in the compilation process:

- **Buffer:** The buffer is a simple program which allocates a temporary buffer in memory (on the heap). It then copies the raw text stream from the source code file byte by byte into the buffer. This is done so that we can easily manipulate the data contained within the buffer since we want to leave the source code file untouched. In C, this might be accomplished by creating a struct with a const char\* that will represent the buffer itself, and can be dynamically resized using realloc() if it ever gets full. Another function might read from the file one character at a time, and yet another that might write characters into the buffer. Various other functions might need to be considered, such as how to clear the buffer, resize the buffer, move the read and write position of the file pointer, etc. A final note about the buffer is that we can create multiple instances of it such as a buffer for lexemes or string literals which we will talk about later.

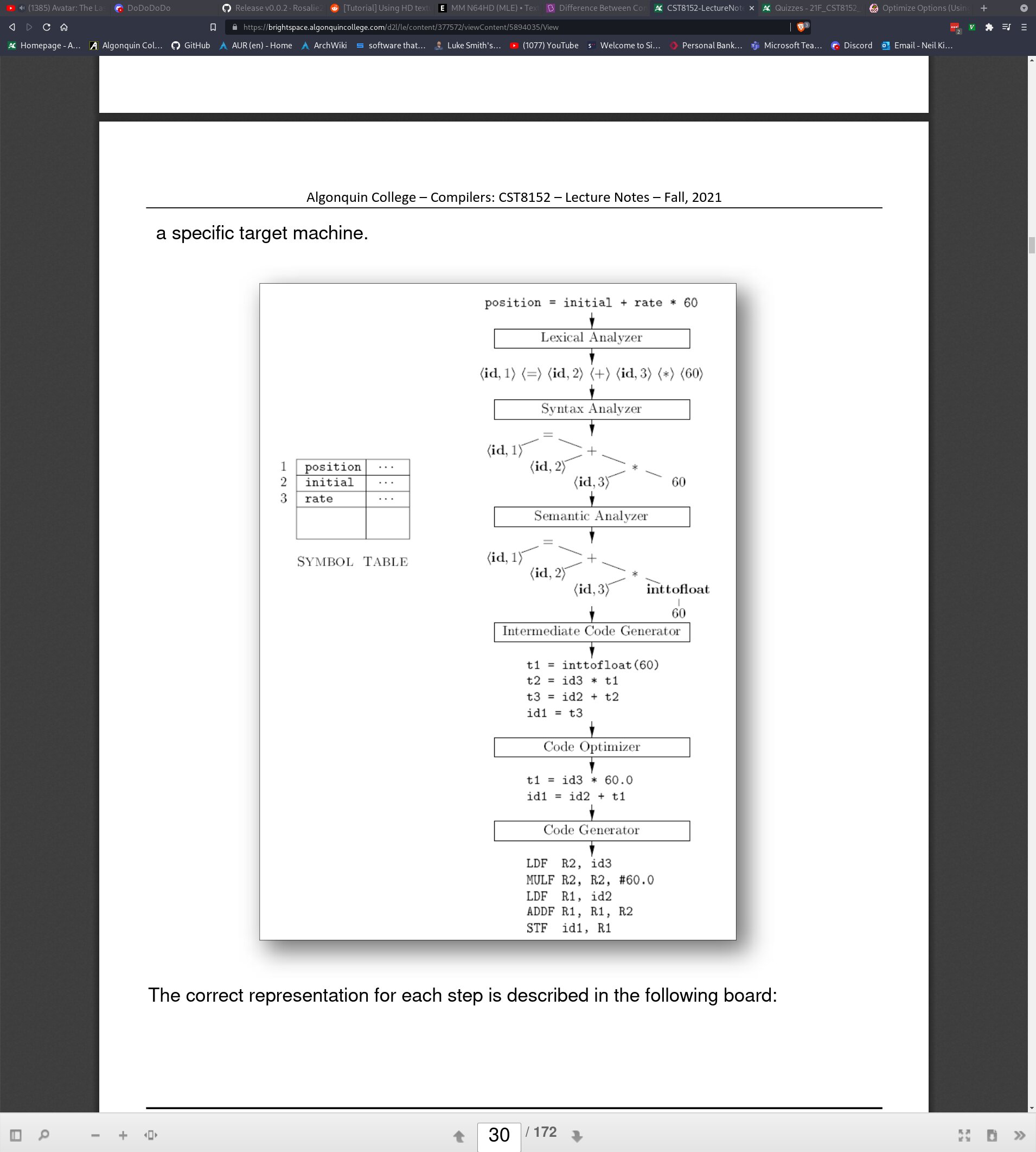
- **Lexical Analyser/Scanner:** The lexical analyser is in charge of taking the contents of the buffer (which is really just a long string of characters), and parsing them into different lexemes which will eventually become tokens. The challenge with the scanner is deciding on what counts as a lexeme; when to create a new lexeme, and when to end it. We will discuss methods of doing this during the scanner section.

- **Syntax Analysis/Parsing:** The syntax analyser creates a tree structure called a parse tree with tokens as nodes of the tree. The tree structure here is not necesarily a binary tree, and does not share similarities with the tree data structure. The parse tree is not ordered by any numeric value, but rather by a grammatical structure akin to that of an actual spoken language.

- **Semantic Analyser:** The semantic analyser is what actually checks that our parse tree is grammatically correct and makes sense. It does this by comparing the parse tree with the information in the symbol table. Technically the semantic analyser is also responsible for creating the intermediate language representation. Taking C as an example, the semantic analyser will translate it to assembly language. Our target language is not assembly, it is machine code (the actual binary representation of assembly), so we are not done yet. This is where we get into the back end of the compiler though.

- **Code Optimization:** The optimization phase will take the intermediate langauge (we’ll stick to assembly as our example), and will optimize it as best as it can. This can often produce very significant increases in performance since assembly has many ways of saving cycles and even executing multiple instructions at once. In a compiler like GCC, you can specify the -O flag with a number between 0 and 3 (there also exists such a thing as -Ofast which is even better than -O3) to specify the level of optimization. Level 3 and -Ofast can often break the code since it attempts to use the most clever but dangerous tricks to save cycles.

-  **Code Generator:** The code generator is a bit strange, but essentially, when we declare variables in the source language, the compiler is meant to record information about these variables, such as their data type, name, value, size in bytes, and address in memory. The code generator will look at the optimized intermediate language and search for variables, and map them to the correct variable definition in the appropriate namespace.

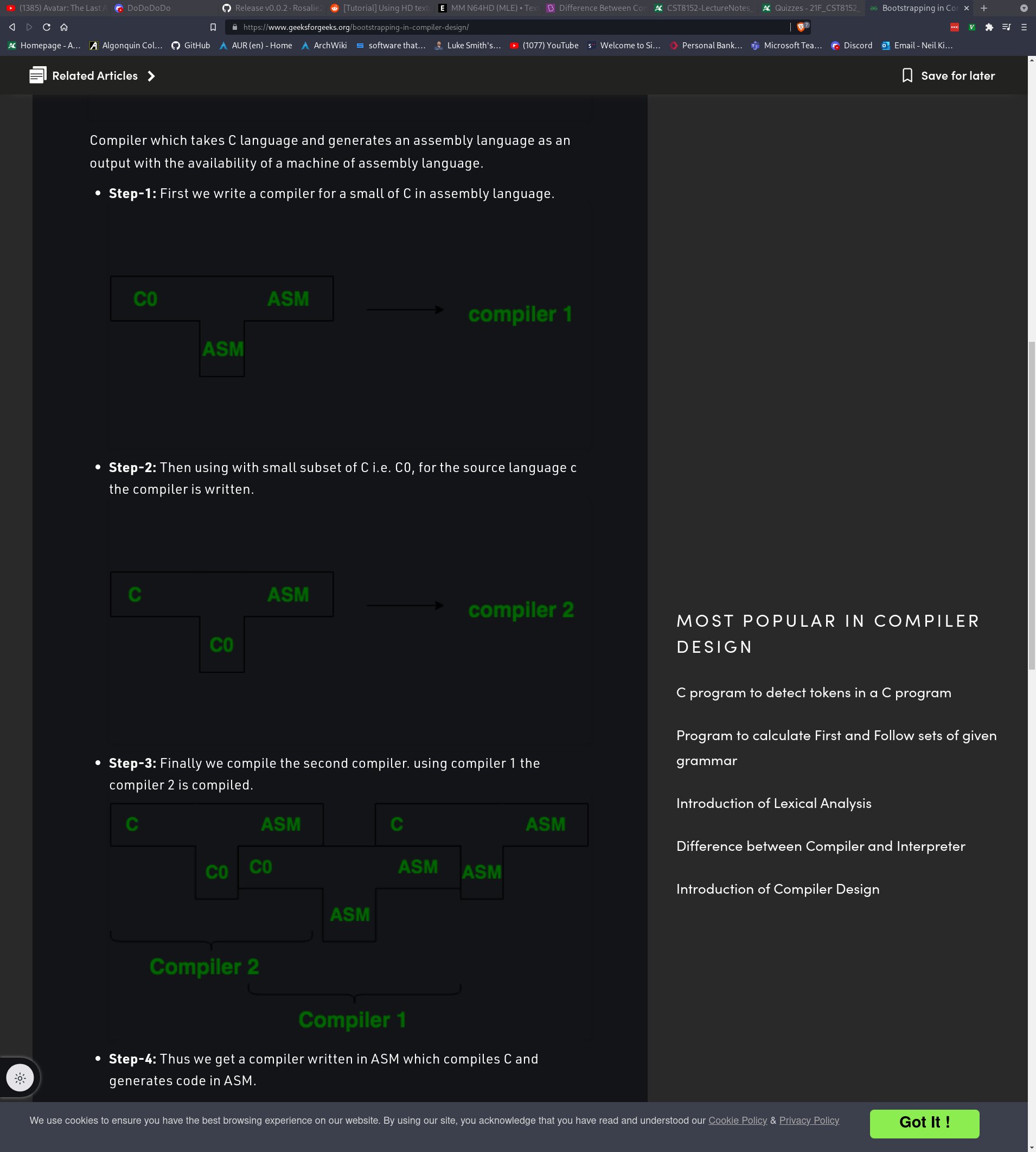


As you can see from the image above, we have the symbol table on the left. Recall that this will contain all of our variable names and information about them such as datatype, size in bytes, address, and value. The first line is source code from the input language (we’ll use C again as an example). The buffer allocates enough space on the heap to copy each character from the source file into itself. Next, the lexical analyser (the scanner), splits the string into multiple lexemes (denoted by the “<>” brackets). The syntax analyser (parser) places each lexeme and any additional tokens into a parse tree according to a grammar structure that we will discuss shortly. The semantic analyser then ensures that the code complies with the grammatical structure defined by the compiler. It might make minor changes implicitely, such as converting the int literal 60 to be a float. It also produces the intermediate assembly code (keep in mind, this does not necesarilly need to be assembly). The code optimizer makes changes to produce more efficient code. And finally, the code generator outputs the target language (in our case, machine code).

Bootstrapping:

I’d like to discuss one more thing before I delve into the stuff above in a bit more in depth. Bootstrapping is the process of creating a compiler for the language that the compiler is being written in (meta, I know). The original assembler was written in raw machine code; 1s and 0s. From there, we could assemble assembly back to machine code. Using assembly, we were able to create a compiler for C. It’s sort of a game of which came first, the chicken or the egg, except that the chicken is the compiler, the egg is the target language (C), and the chicken definitively came first ie. The compiler needs to be written before the language is able to actually do anything. Compilers like GCC and Clang are written in C. Writing a compiler that outputs the source language as the target language is called bootstrapping after the English expression “Pulling oneself up by their bootstraps”.

It also needs to be noted that the first compiler for C (or any language for that matter) cannot possibly implement the entire language. For example, think about how many libraries are used in C that had to be written in C. These could not have existed when first compiling C code. Therefore, we generally create what’s called a subset of the language (let’s call it C0). C0 is not the full implementation of C, and does not have all of the features, keywords, or libraries, but we don’t care. We want to write a compiler in C0 for C. This is typically shown in a diagram called T-diagrams. I will try to explain an example to you but honestly, I feel like T-diagrams are super forced and don’t actually make sense; schools just want them to make sense...

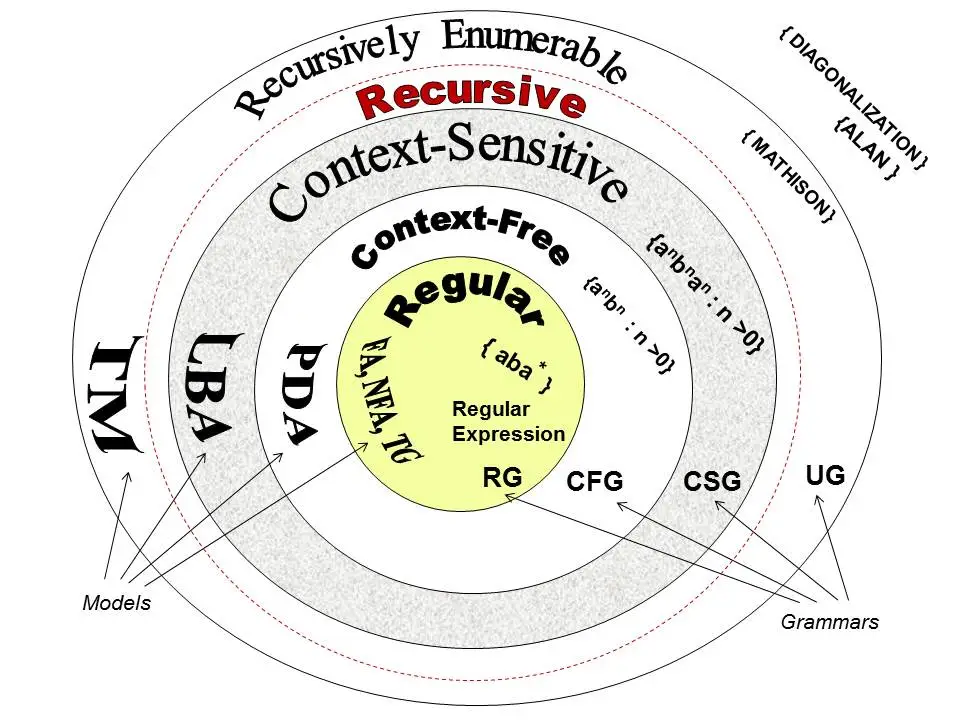


Here you can see some T diagrams. The idea is that the input language is on the left, the output/target language is on the right, and the bottom means that it was “written in” that language. So compiler 1 was written in assembly, it took in the subset of C, C0, and it output assembly. Compiler 2 was written in C0, took in C, and output assembly. Now that we can output C code to assembly code, we can write our compiler in C, and turn it into a binary executable which is denoted on the right. As for the wack ordering of the T-diagrams in step 3? I have no clue. The idea is that the language that the compiler was written in (bottom of the T) needs to be aligned next to the input language of another T-diagram, and they must match. Notice C0 touches another C0 between compiler 2 and compiler 1, and that ASM touches ASM between compiler 1 and... compiler 3? I’m not even really sure what the right T-diagram denotes. T-diagrams are dumb, but school makes you learn them, so I guess you’ll just have to figure this one out for yourself.

Lexical Analyser/Scanner:

I’m sure most of what you’ve read thus far has been at least mildly confusing since I have not really defined certain terms. There are a few very important distinctions that must be made when we speak about the compilation pipeline to avoid utter confusion (because trust me, I was confused during my lectures on the subject). The first important distinction to make is the difference between **tokens** and **lexemes**. As the name “lexical analyser” suggests, this phase of the compilation deals with taking our stream of characters and creating lexemes. If you were non-the-wiser like myself, you would likely come to the conclusion that the terms “token” and “lexeme” can be used interchangeably, but this is not the case. Lexemes are essentially any sequence of letters. An extremely basic language can generate an infinite set of lexemes, however, only a few of these lexemes are actually valid tokens within the target language. Often times, lexemes are associated with variable literals ie. Valid values that a variable might contain. Lexemes can also include method/function names, or variable names, etc. Tokens include all valid lexemes, as well as things like operator symbols (eg. +, -, \*, /, &, $, %, etc.), keywords, or other symbols like brackets, curly braces, etc. Therefore, forget about tokens momentarily, because those will be relevant during the syntax analysis/parsing phase of the compilation pipeline. This section will deal with how we can decipher what constitutes as a valid lexeme. For example, what are some valid sequences of characters that an int could contain.

There are a few ways in which we can decide whether or not a sequence of characters is a lexeme, but this is where another distinction should be made. We need to be aware of automata, because only certain automata can accept certain types of grammar. There are 4 types of automata, each with elevating complexity from the previous. The 4 types in order of least complex to most complex is as follows: Finite State Automaton, Pushdown Automaton, Linear-Bounded Automaton, and Turing Machine. I’m sure you may have heard of Alan Turing who conceptualized the Turing Machine even before computers existed. Unfortunately, we will only be covering Finite State Automaton, because while they are the least complex, they are still capable of producing complex languages. As I mentioned, each of the four automaton can process different types of grammar. If you aren’t aware of Noam Chomsky, I recommend that you read a bit of his biography, as he is the father of a lot of linguistic theory, as well as a phillosopher, cognitive scientist, historian and more. He proposed 4 types of grammar which align with the 4 types of automaton: type 0 (unrestricted), type 1 (context sensitive), type 2 (regular grammar) and type 3 (regular grammar). Here is a chart to help visualize the increasingly complex grammars, as well as which automaton can process which grammars:



|  |  |  |  |
| --- | --- | --- | --- |
| Grammar Type | Grammar Accepted | Language Accepted | Automaton |
| Type 0 | Unrestricted Grammar | Recursively Enumerable | Turing machine |
| Type 1 | Context-Sensitive Grammar | Context-Sensitive Language | Linear-Bounded Automaton |
| Type 2 | Context-Free Grammar | Context-Free Language | Pushdown Automaton |
| Type 3 | Regular Grammar | Regular Language | Finite State Automaton |

Now, as I mentioned, we will only be looking at Finite State Automaton. Such systems understand Regular Grammar ie. Regular Expressions aka. Regex, which you may have studied at some point. Therefore, in order for us to decipher if a lexeme is valid or not, we can use regexes to find matching patterns within a sequence of characters.

Regex:

Since the following topics will require some knowledge of regular expressions, we will be going over the notation that I’ll be using going forwards. Note that there are different flavours of RE, similar to various flavours of assembly language. They all do the same thing, but sometimes certain symbols mean one thing or another depending on the flavour. Here are the definitions of each symbol, and it’s connotated meaning:

**Alternation/OR:** The pipe ‘|’ refers to alternation, or in other words, a logical OR operation. Given two symbols a and b, the pipe would suggest that a match can be found if either a OR b appears eg. “a | b”.

**Brackets:** Brackets will be used for nothing other than grouping symbols with precedence. This follows the same logic as mathematical expressions like (2n)^2. Similarily, the RE (a|b)\* will first apply the alternation from the pipe, followed by the kleane closure.

**Kleane Closure:** The kleane closure (pronounced “cleany closure”), represented by ‘\*’, is used to suggest that 0 or more instances of the pattern can be matched. As we’ll learn later, this is actually a recursive function which is very poweful in linguistics. The example that I used in the previous definition (a|b)\* means that a OR b can be matched 0 or more times.

**Positive Closure:** The positive closure, ‘+’, is the same as the kleane closure, except that it’s used to mean 1 or more instances rather than 0 or more. For example a+ is equivellant to aa\* because the first a is required, and the second can have 0 or more instances ie. One or more instances.

**Character Class:** A character class is a set of symbols that often times have some relation, but not always. For example, the character class [A-Z] is comprised of capital letters A-Z. [a-z] would be lower case letters from a-z and [A-Za-z] would be all letters, capital or non capital. Similarily, we often use [0-9] to denote digits 0-9.

**Complement Character Class:** The complement character class is often used to mean “starting with” in many regex flavours, but in our case it will be used as the negation of character class. [^A-Z] would mean any character except for capital letters A-Z, as if it had a NOT operation applied (eg. ![A-Z]).

**Optional:** The optional character ‘?’ means that the preceding character was optional, and is not necesarry for a pattern match. The RE ab? could either match with the sequence a or the sequence ab since b is optional.

**Alphabet:** (ignore the upside down question mark). The Sigma symbol (the 18th letter of the Greek alphabet) refers to an alphabet. Alphabets are a special set of characters (ie. no duplicates) that can make up a language. Our alphabet is as follows:

Similarily, the binary alphabet is

**Empty String/Epsilon:** The symbol ‘’ denotes epsilon or empty character. This is different from the alphabet character ‘’ because it is a meta symbol which is only used for regular expressions. An empty character ‘’ could reasonably exist within an alphabet, but epsilon could not.

With these meta symbols in place, we can begin building regular expressions. We will begin looking at some examples of how we could construct lexemes for variable literals within our target language. For example, creating a lexeme for an integer literal, byte literal, float literal, character literal, string literal, etc. We begin by defining lexeme classes which are basically just identifiers associated with a RE. The syntax for this is simply “lexeme\_class = regex”, as if it were a key value pair. This might seem redundant, but by creating lexeme classes which correspond to matching REs, we can then more easily create other lexeme classes with existing lexeme classes. For example:

Lexeme Classes: Regular Expression:

L (alpabet of letters) = {a, b, c, d, e, f ... z}

D (alphabet of digits)= {0, 1, 2, 3, ... 9}

IL (integer literal) = D+ (any sequence of digits from the alphabet D, so long as there is at least one)

SL (string literal) = L+ (any sequence of letters from the alphabet L, so long as there is at least one)

FPL (float point literal) = D+.D+ (any sequence of at least one digits, followed by a period, and any sequence of at least one digits)

Notice that by using the positive closure for IL and SL (as opposed to kleane closure), we avoid the epsilon . This is important because if a lexeme class can be defined as nothing, then we get errors in our compiler. As we will see shortly, our Finite State Automaton transitions from a starting state and continuously visits neighboring states until it reaches a final state. The condition under which it transitions between states depends on the lexeme classes. If one such lexeme class were to be set to the epsilon, the automaton could reasonably transition states without any condition. This is known as an epsilon transition.

Finite Automaton:

Recall that there are 4 types of automaton: Finite State Automaton, Linear-Bounded Automaton, Pushdown Automaton, and Turing Machines. As I mentioned, we will only really be looking at Finite State Automaton, which are capable of reading regular expressions. It gets a bit more confusing though because there are actually two types of Finite State Automaton: Deterministic Finite Automaton (DFA) and Non-Deterministic Finite Automaton (NFA). These automaton are the same, except that DFAs do not accept epsilon transitions as I referred to a moment ago, whereas NFAs do. Before we delve into DFA and NFA, however, I would like to briefly discuss the Kleane Theorum.

Kleene Theorum:

The Kleene Theorum is not directly related to automata, but it may help us understand what’s going on a bit better. In 1956, Stephen Cole Kleene formulated and proved the following theorum:

“A language can be defined by any of these three methods:”

1. Regular Expressions / Regular Grammar

2. Transition Graph / Transition Tables

3. Finite Automaton / Finite State Machine

As you can see, we’ve touched on Regular expressions and are now skipping to the third method of defining a language: Finite automaton. In the next section, we will also be covering transition tables. Essentially, these three methods are different, but serve the same purpose: defining a language. The Kleene Theorum proved that these three methods were equivellant, and could therefore create the same identical language.

Deterministic Finite Automaton:

Finite Automaton (both DFA and NFA) consist of the following parts:

**Finite Set of States:** There is a limited number of states that a finite automaton can transition towards. There must always be a start state, and at least one final state (called final states, halting states, terminal states, or accepting states). This is in contrary to a Turing Machine, which can essentially define new states for itself to transition towards.

**Alphabet:** We discussed alphabets in regular expressions, and automaton also require at least one alphabet. The symbols within the alphabet can determine the next state of the automaton. For Deterministic Finite Automaton (DFA), transitions to other states must always be determined by a symbol from the alphabet. This cannot be unambiguous either; a symbol can only lead to one state at a time ie. The letter a cannot simultaneously determine that the automaton transitions to states 2 and 3 at the same time. Since Non-Deterministic Automaton can use epsilon transitions, they do not necesarilly require alphabet symbols to determine the next state, however, alphabet symbols can be used. NFAs are also allowed to have ambiguity ie. a letter can simultaneously determine that the automaton transition to states 2 and 3 at the same time.

**Set of Non-Terminals:** We will get into terminals a bit more when we take a look at Context-Free Grammars in the parser section, but essentially, these can be thought of as tokens within a heirarchical tree-like structure. The starting state of the automaton is the root node of the tree, and the set of non-terminals are parent nodes within the tree.

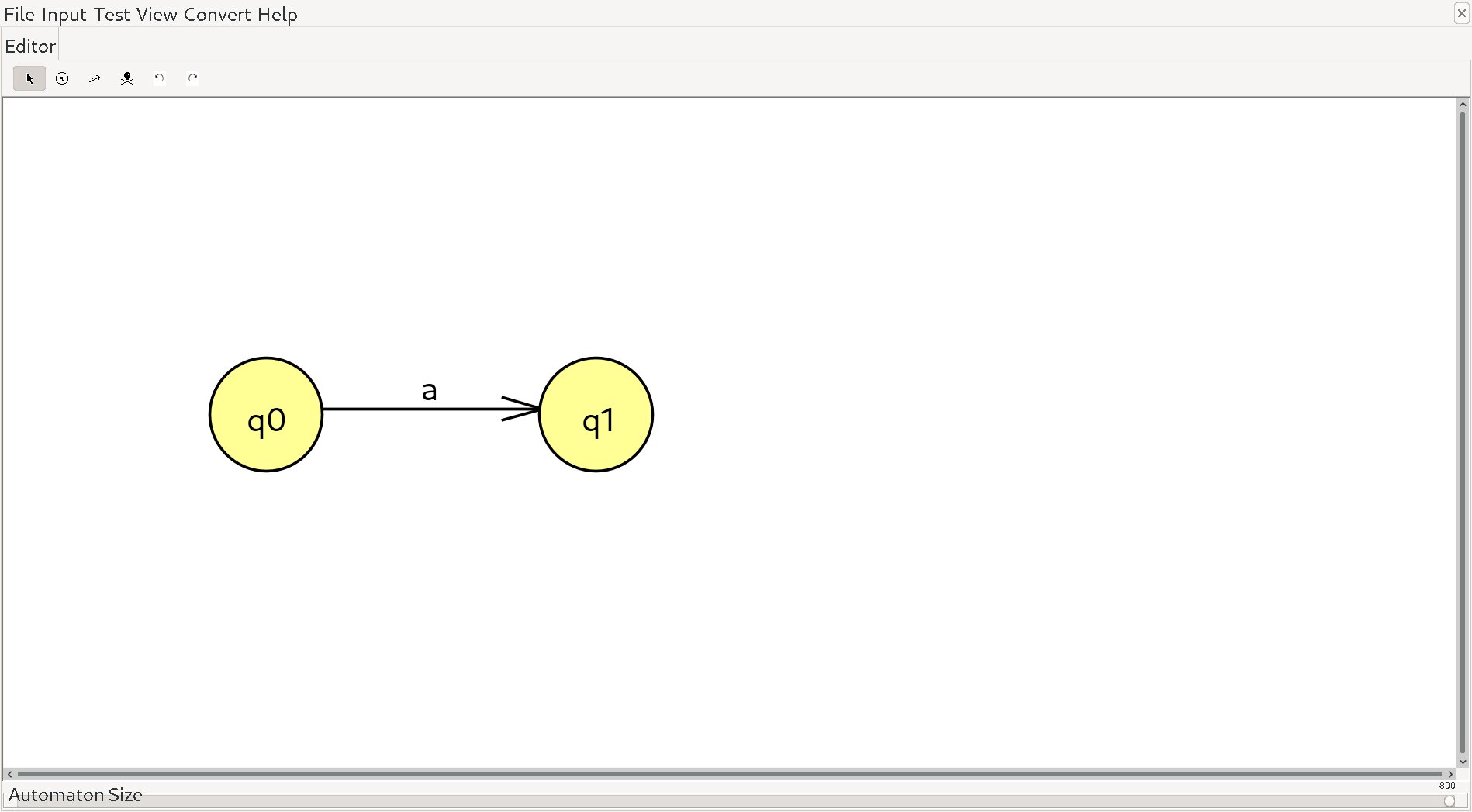
**Set of Terminals:** The set of terminals are the leaf nodes of the tree structure which I mentioned previously. These would be like the lexeme classes that will eventually become tokens. Things like method names, variable names, variable literals, etc.

Non-Deterministic Automaton:

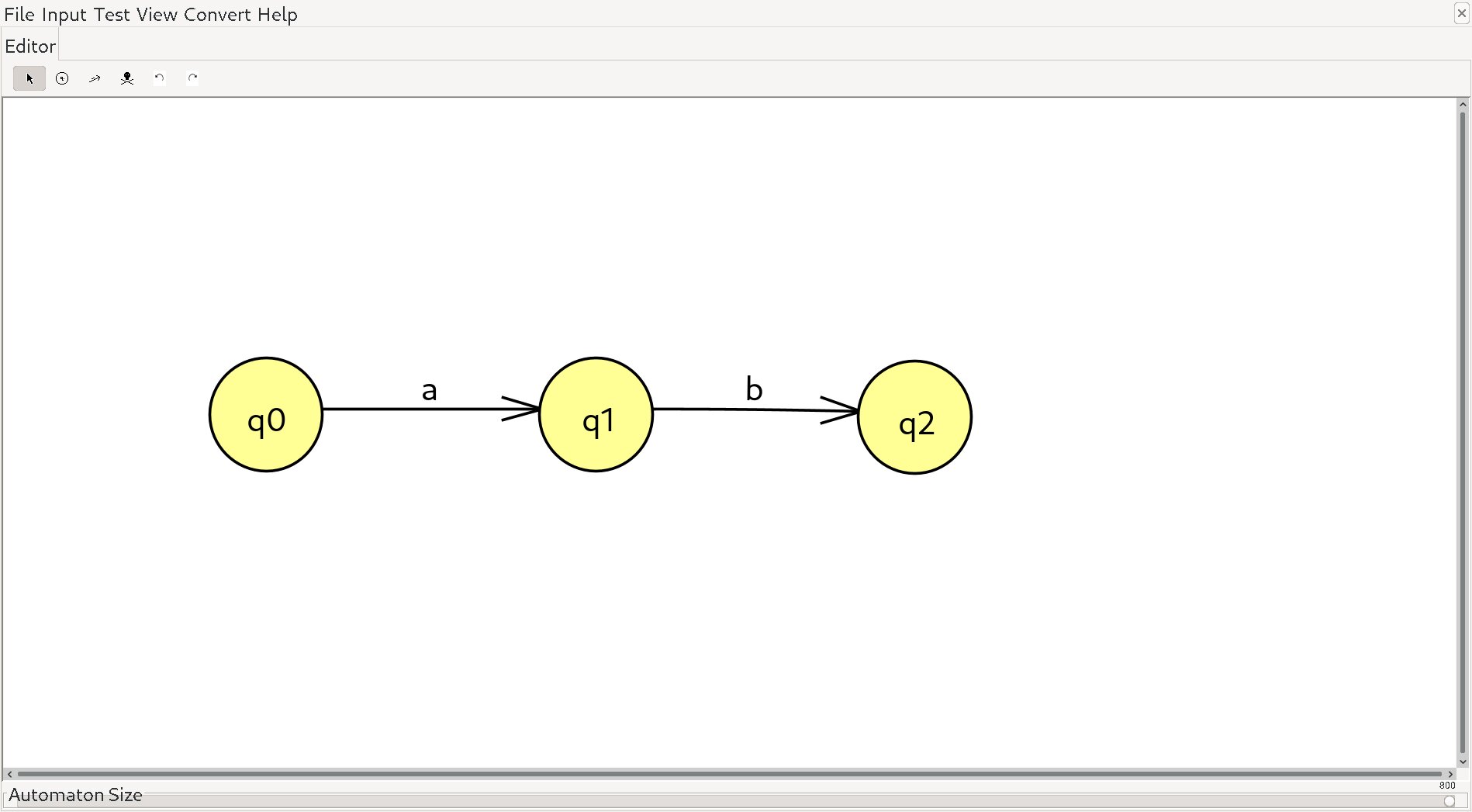
Let’s discuss the differences between DFAs and NFAs. As I’ve mentioned, NFAs can have epsilon transitions ie. Change state without recieving any sort of determining input. However, NFAs do not necesarilly require epsilon transitions. There happens to be a more broad factor that determines whether or not a Finite State Automaton is a NFA. That factor is ambiguity when transitioning between states. The epsilon transition guarantees that a Finite State Automaton is an NFA, but we ought to also look for cases where a certain symbol is allowing the automaton to choose between 2 different states. This is why we call the two kinds of Finite State Automaton DFA and NFA; it is either always conclusive as to what the next state should be, or it isn’t. We will begin to look at how we can represent Finite State Automaton, as well as what non-deterministic transitions might look like now.

Representing Finite State Automaton Transitions:

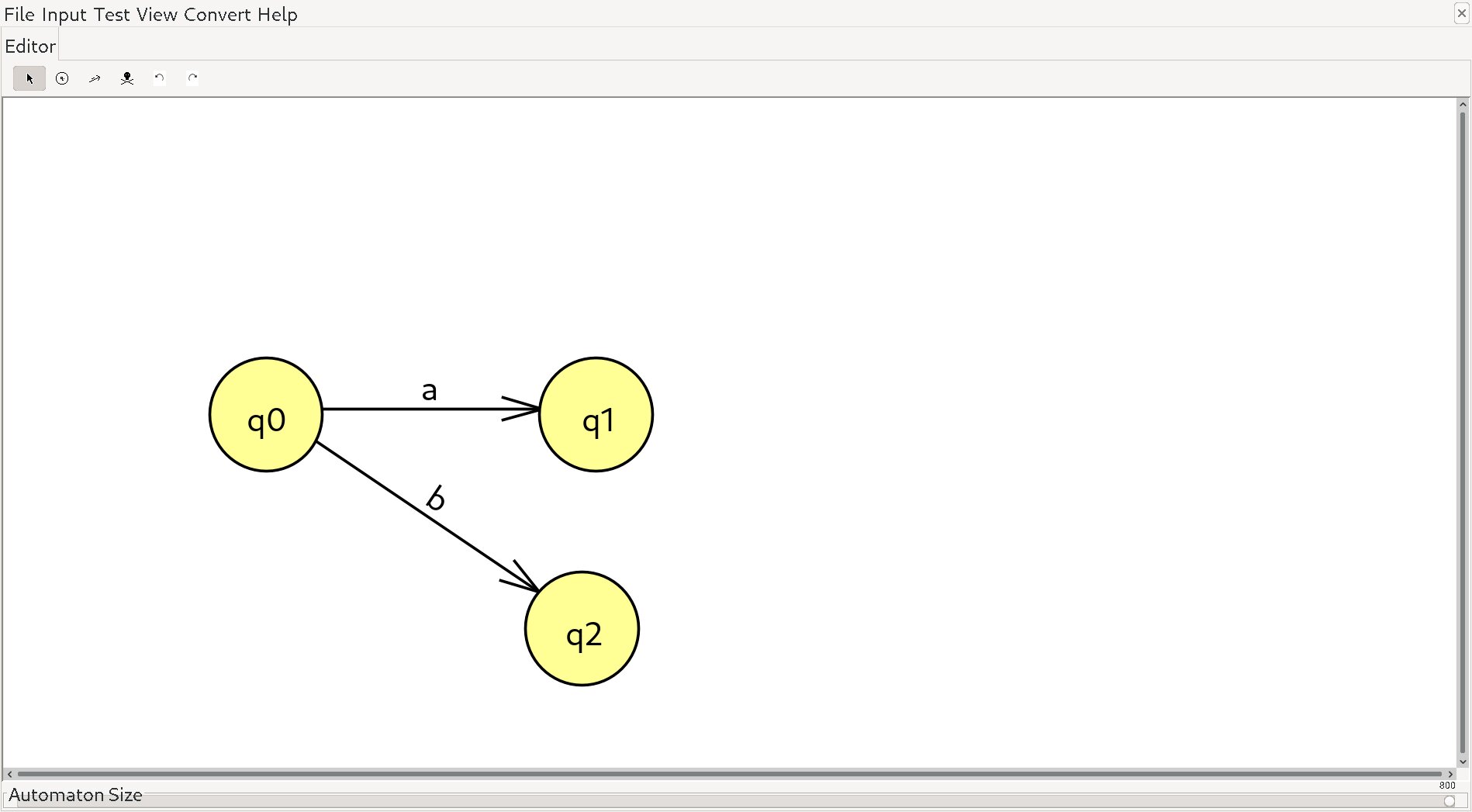
As I mentioned earlier, each Finite State Automaton will begin with some sort of start state which we can just represent as a circle labeled 0 or start (the labels don’t actually matter, they just help us visualize the flow of the transitions between states).



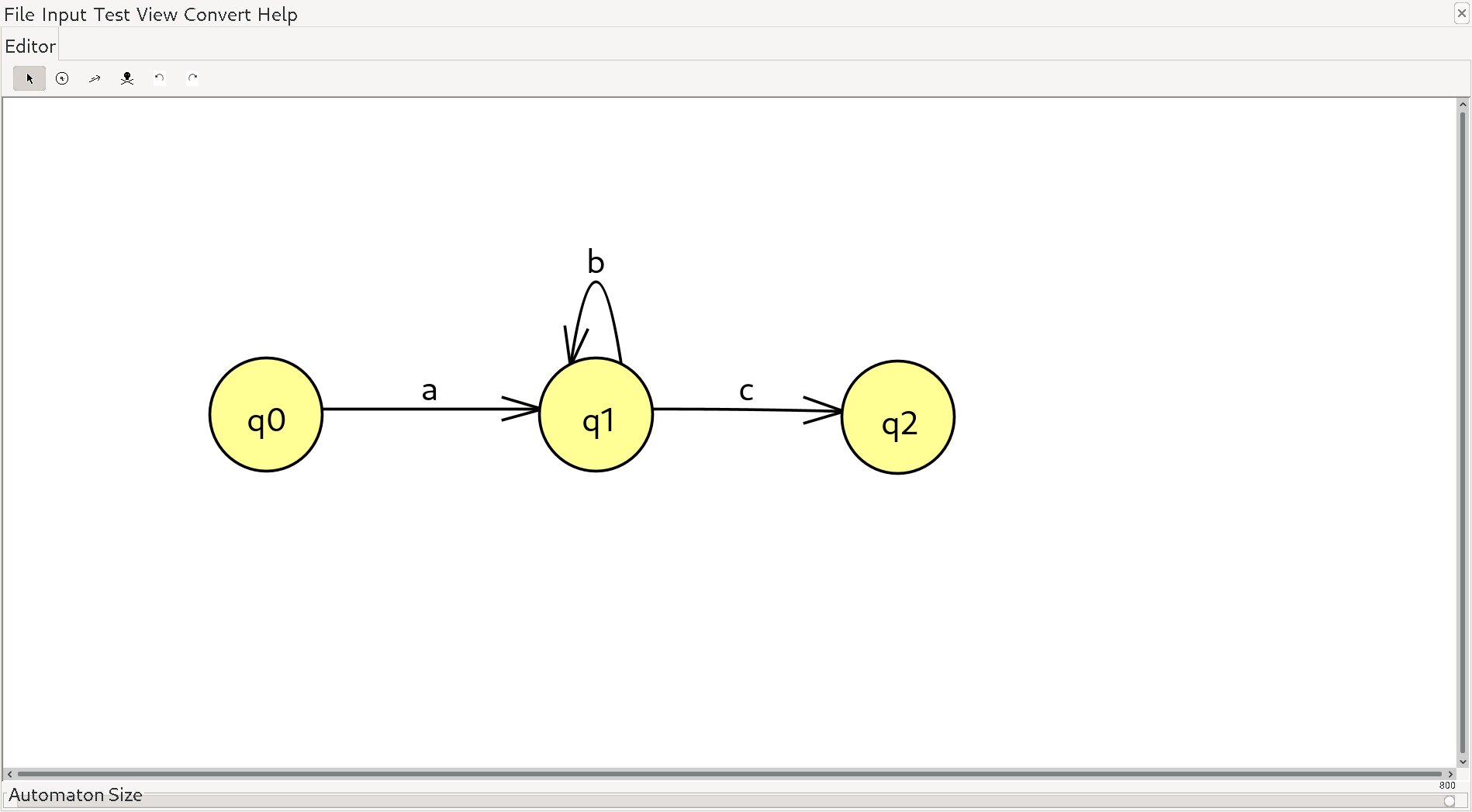
In a more formal definition, the start state would usually be represented as Q, and each following state q1, q2, q3, etc. Here, our start state Q will transition to the second state, q1, if the automaton processes the symbol a. The regular expression is simply “a”.



Here, we have a regular expression ab ie. we are concatenating a and b together. The automaton first reads a, transitions from Q to q1, then reads b, and has condition to transition to q2.



This represents the RE a|b ie. alternation. Starting from the input state Q, if we read an a, we transition to state q1, or if we read a b, state q2. Note that this is still very much deterministic, and can be considered a DFA. If it is reading a, the transition to q1 is determined, and b, the transition to q2 is determined.

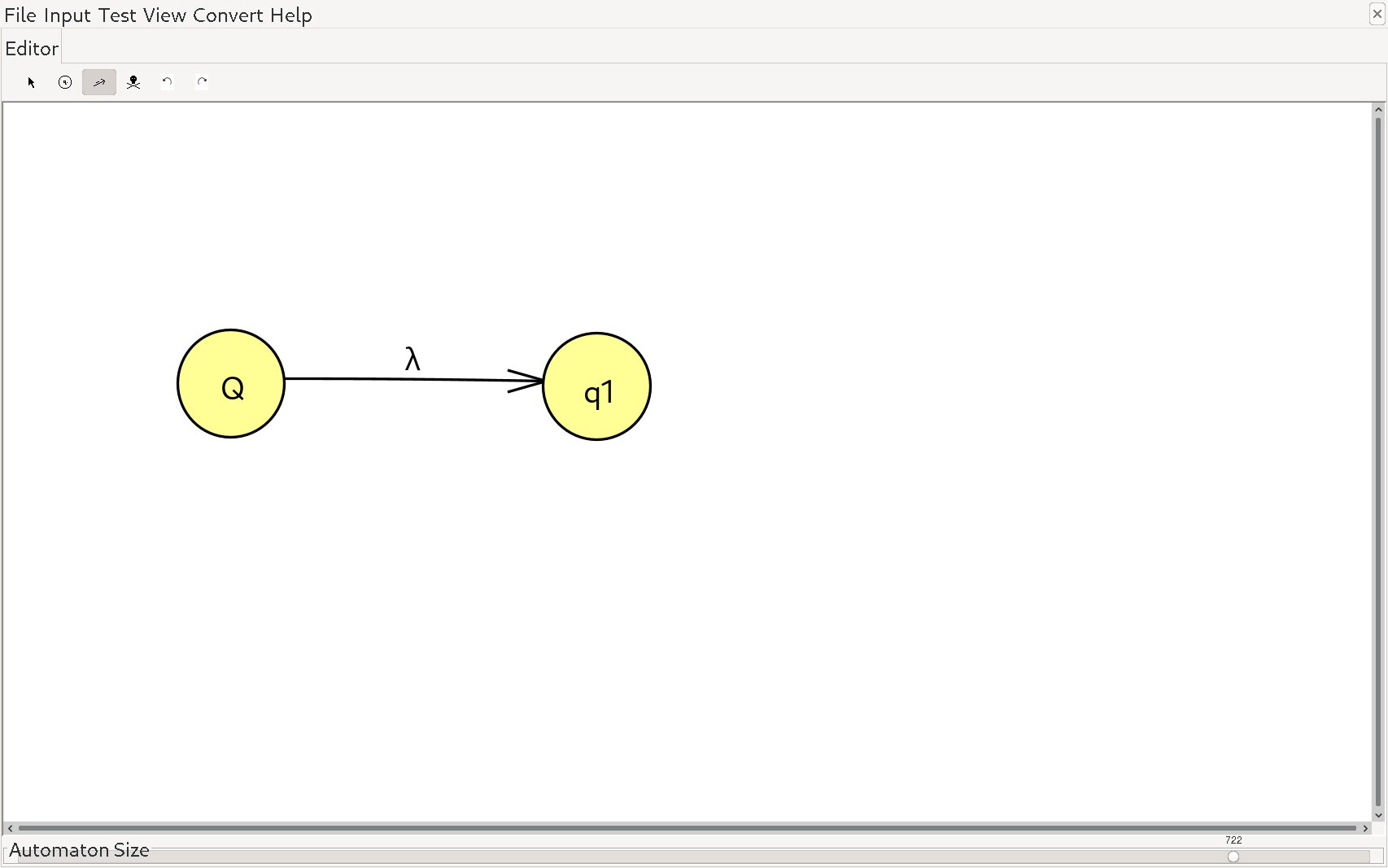


Here, we have a recursive functionality which can be viewed as a kleene closure in regex. The regular expression for this DFA is ab\*c. Starting from Q, we transition to q1 if the automaton reads an a. Then, b can be read 0 or more times. This is possible because if we read b, we transition back to the same state, q1. Finally, if we read c, we transition to q2.

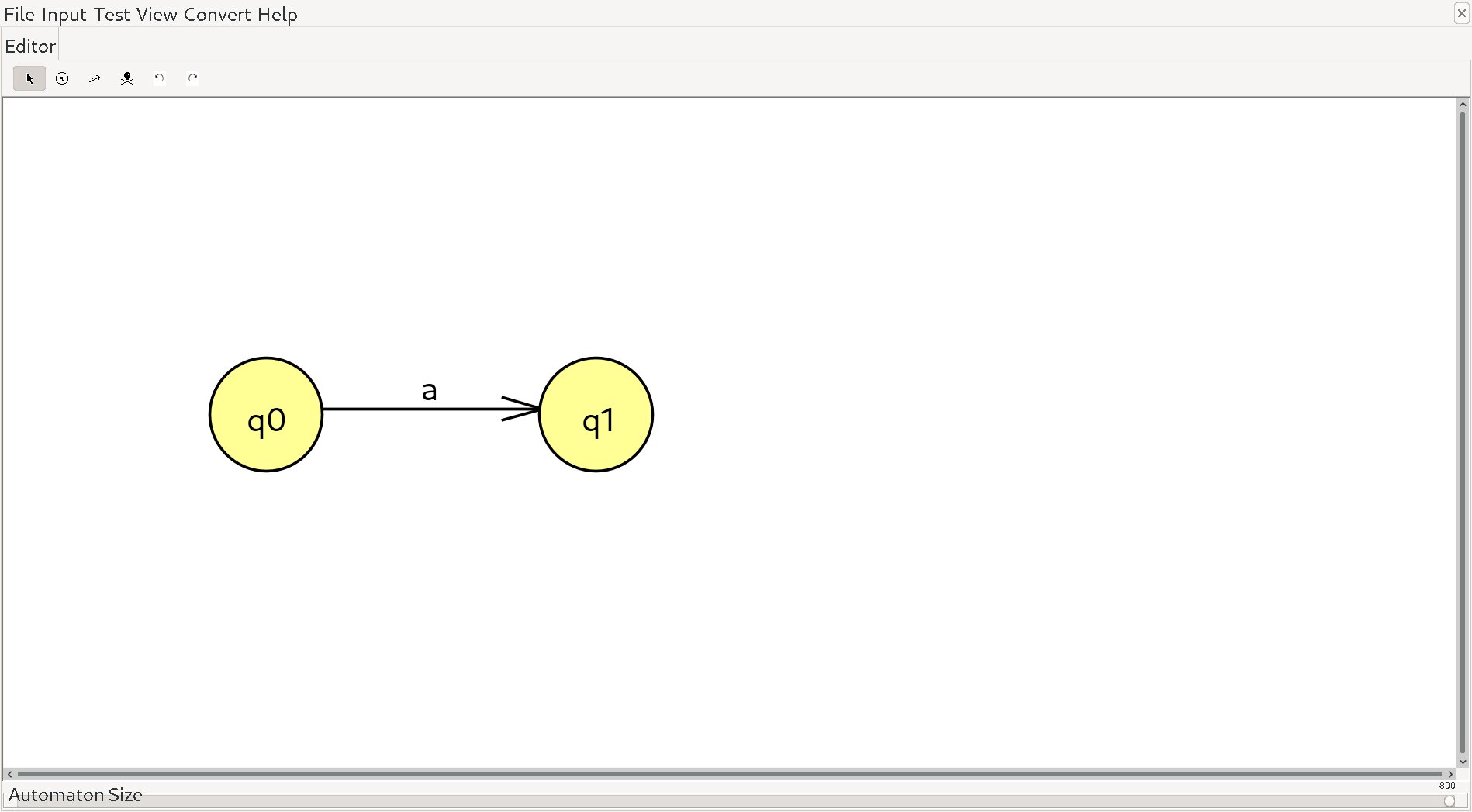
Thompson’s Construction Algorithm:

I did not go into much depth about how we can represent Finite State Automaton, but that is okay because I will explain as we go, and it should be quite comprehensible. The fact that we can follow such a diagram and deduce what sorts of lexemes the automaton can produce is a good start, however, it is often times more useful for us to be able to convert regular expressions into NFA. This is not always such a simple task. According to the Kleene Theorum that we covered, Regular Grammars and Finite Automaton are equivellant. But as I mentioned, NFAs are also problematic if we are trying to implement them into code; which if we’re writing compilers, would be the case (keep in mind that the execution of code is mostly deterministic since statements and switch statements always move the Instruction Pointer to a specific state). Luckily, it is possible to convert NFAs into DFAs which will be our next point of discussion. First though, we must convert RE to NFA. We do this using Thompson’s Construction Algorithm. This is not a simple algorithm by any means, and as I am a very visual learner, the only way for me to really describe the algorithm is with diagrams. All of the following material was inspired by this article <https://medium.com/swlh/visualizing-thompsons-construction-algorithm-for-nfas-step-by-step-f92ef378581b> which I highly recommend checking out. I will be attempting to make the algorithm even easier to understand than the article, but I’m not so confident in my abilities so definitely check out the article if my explanation is not satisfactory. The article begins by describing a few rules about Thompson’s Construction Algorithm:

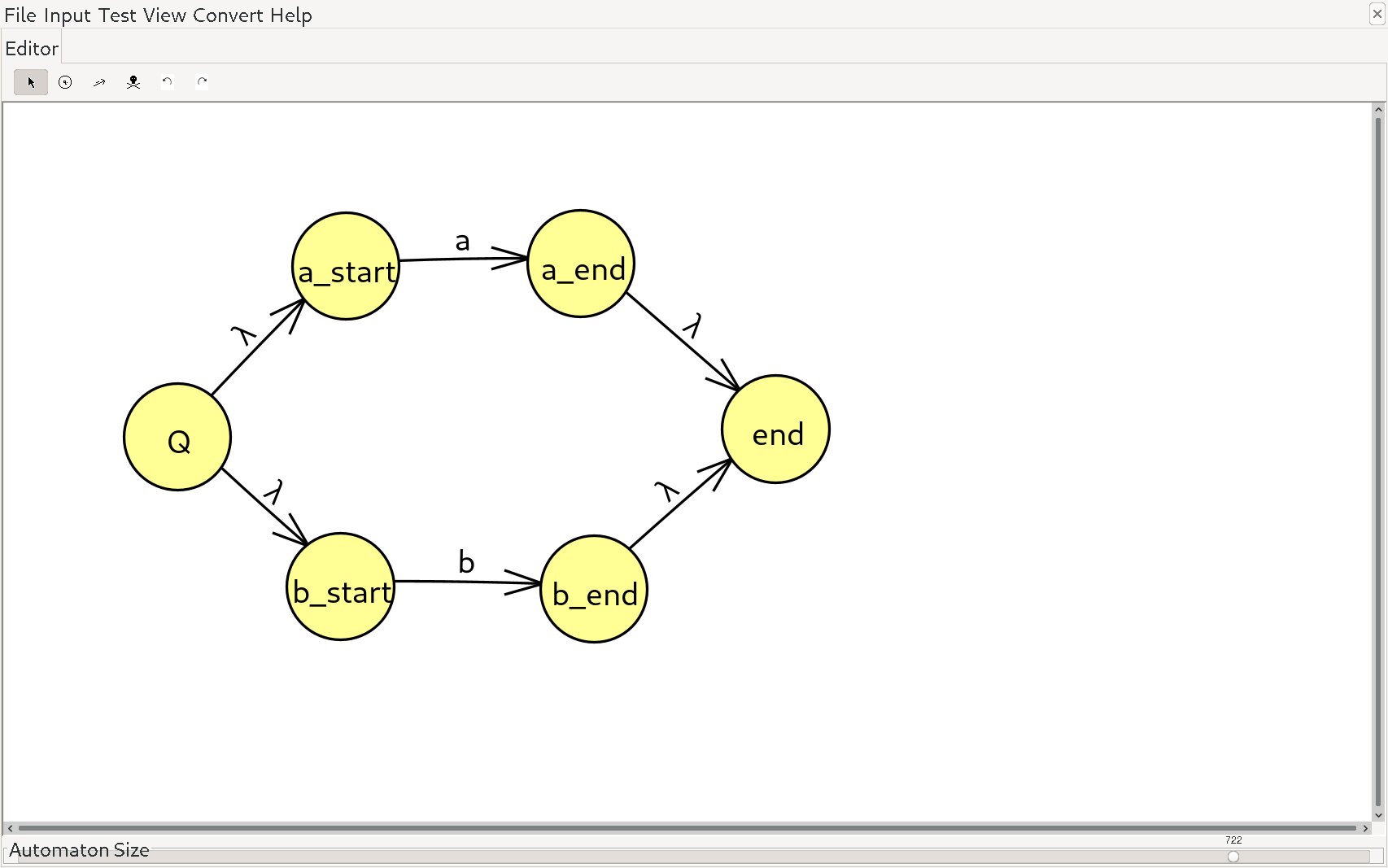
**1. Epsilon Transition:** An empty regular expression will be converted to an epsilon transition:



**2. Symbols:** If the automaton reads a RE containing one symbol eg. a, it is converted as follows:

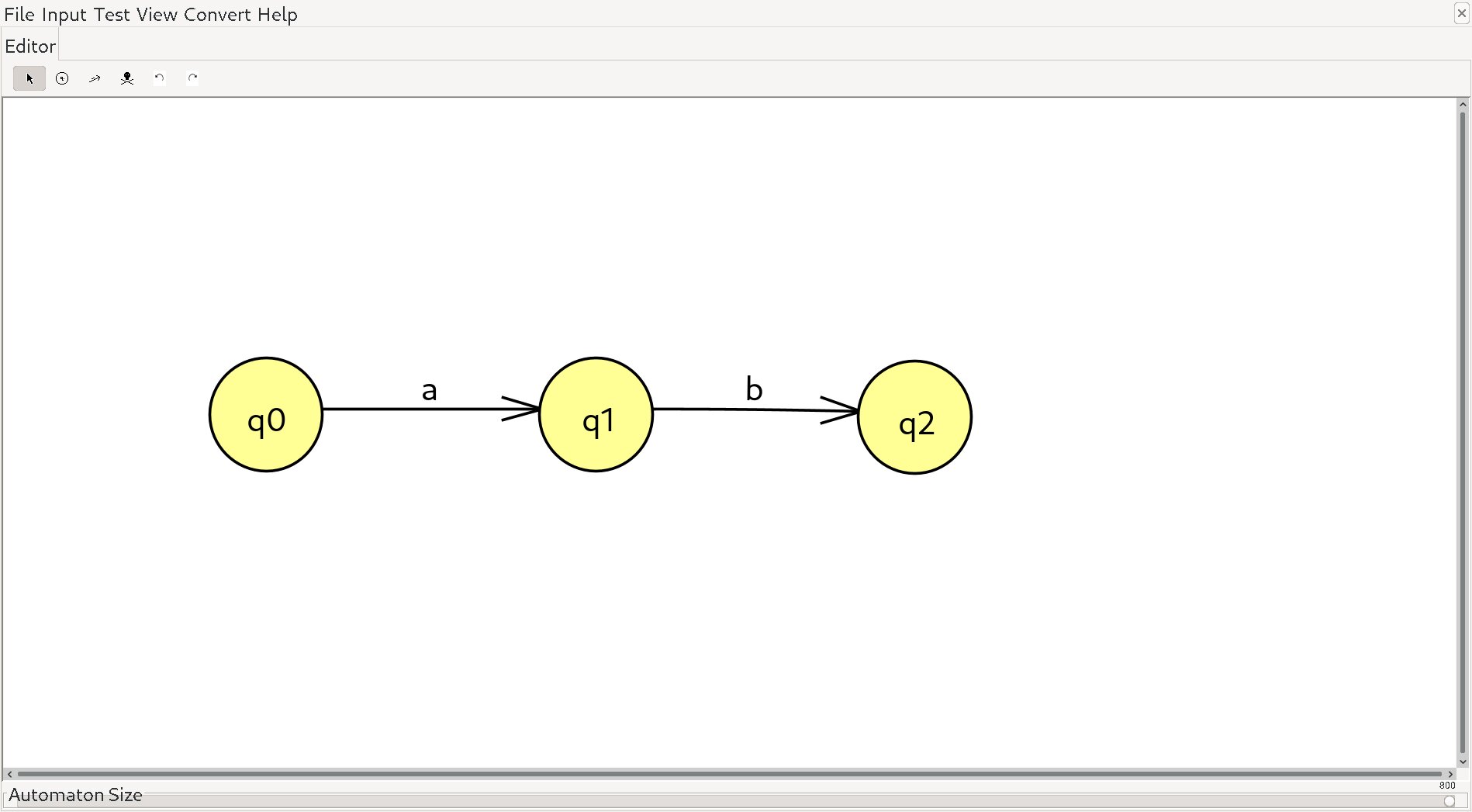


**3. Union AKA Pipe ‘|’:** The pipe is considered to be a union operation between two sets in the mathematical study of set theory. Notice it must be between two sets, ie. two transitions in our case. a|b can be represented as follows:

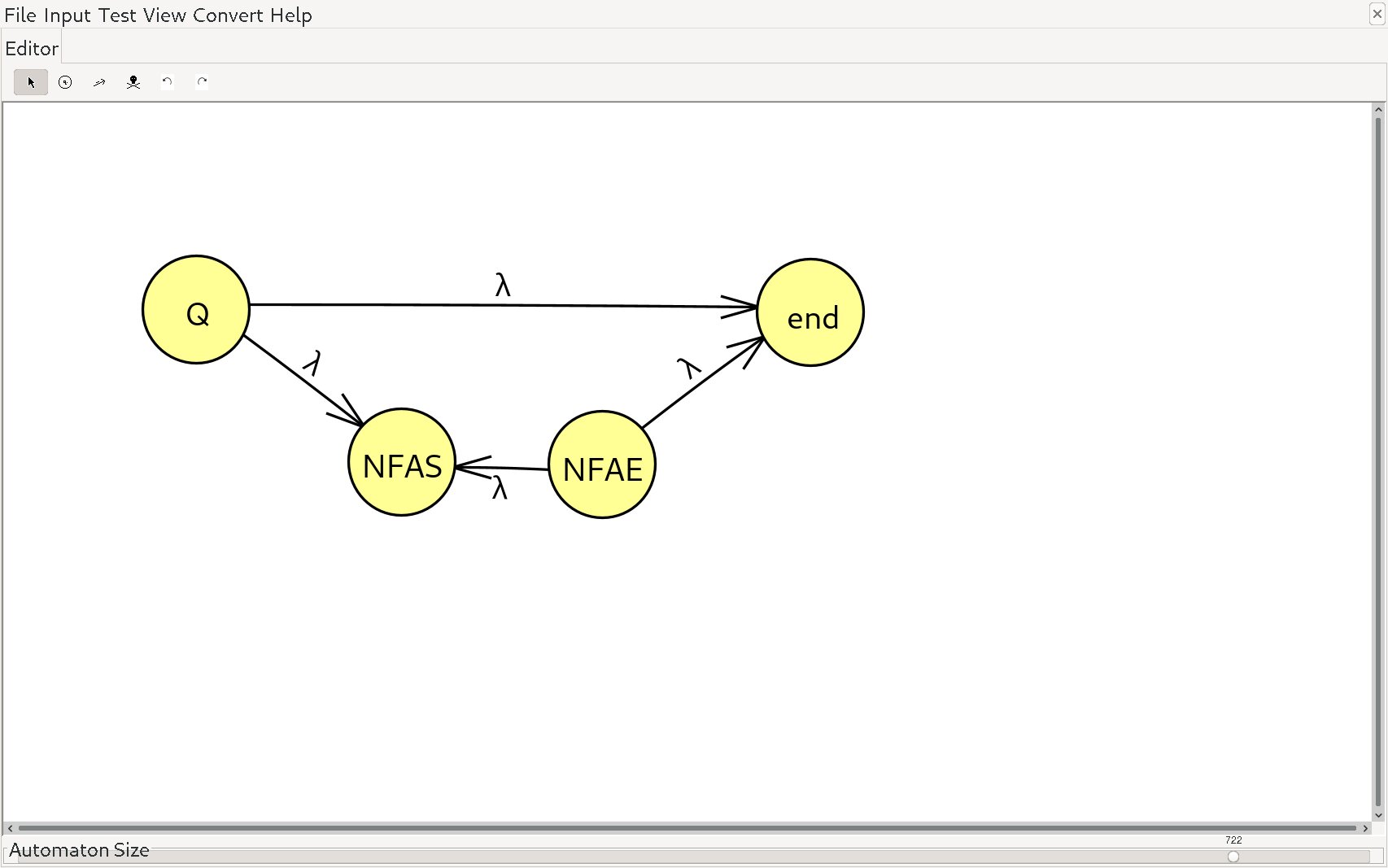


As you can see, the rule whenever alternating between 2 transitions is to add two additional nodes. In this case, I’ve labeled them Q and end, but they don’t necesarilly need to be the actual start and end of your automata. Epsilon transitions connect these things together denoted by the lambda symbol in the program I’m using.

**4. Concatenation:** Concatenation made by connecting 2 symbols is exactly the same as I’ve described before. Given two transitions: {Q -> a ->1} and {Q -> b -> 2} and the RE ab, we can create the following transition:



**5. Kleene Closure:** If we are given a kleene closure (\*) within a RE, we do something similar to what we did with alternation. We add two more nodes to either side, but this time we create epsilon transitions between all neighboring nodes. Example:

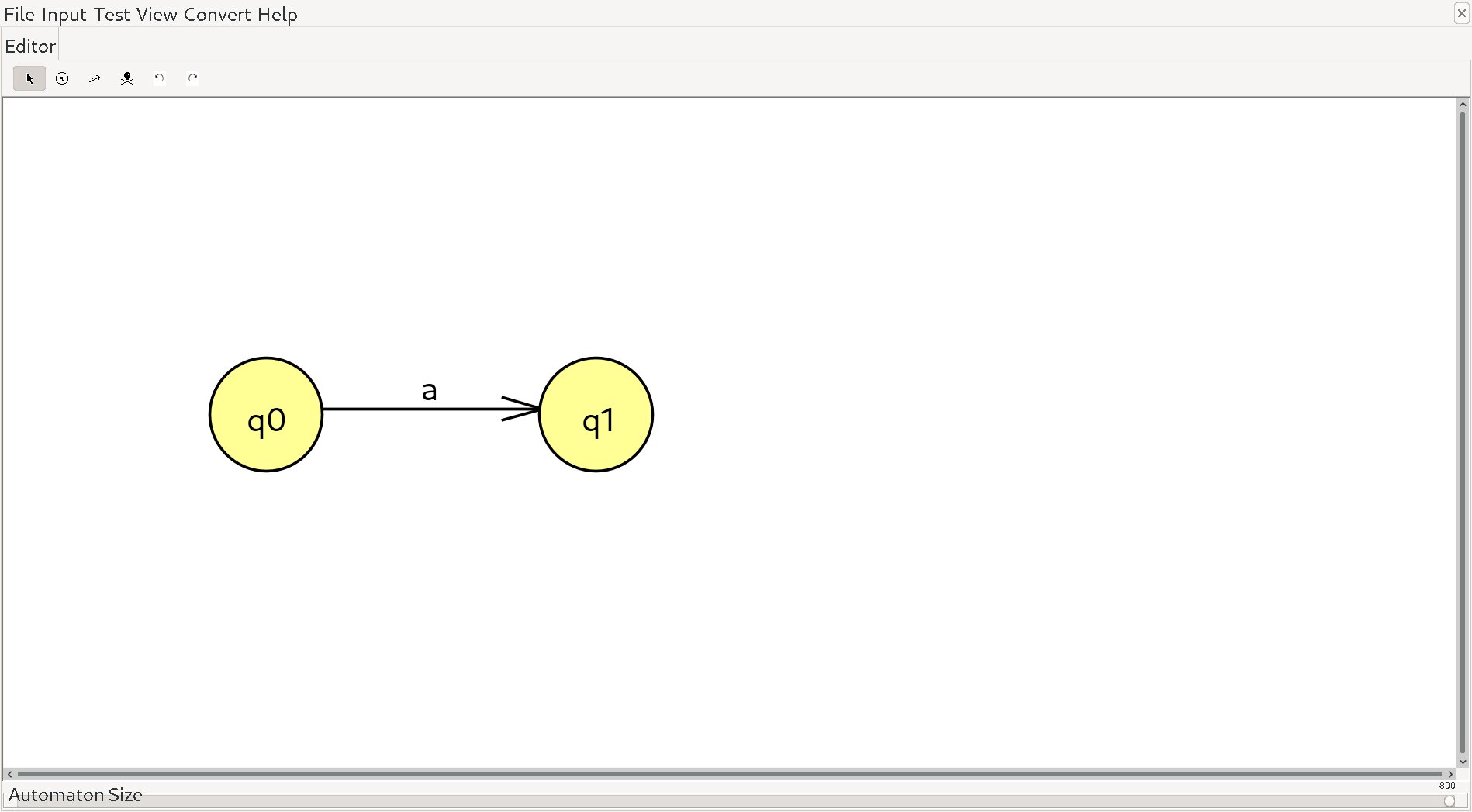


Let’s say you had the RE (a|bc)\* The red box would represent the enclosing RE (a|bc) and the new states, Q and end, would be the kleene closure.

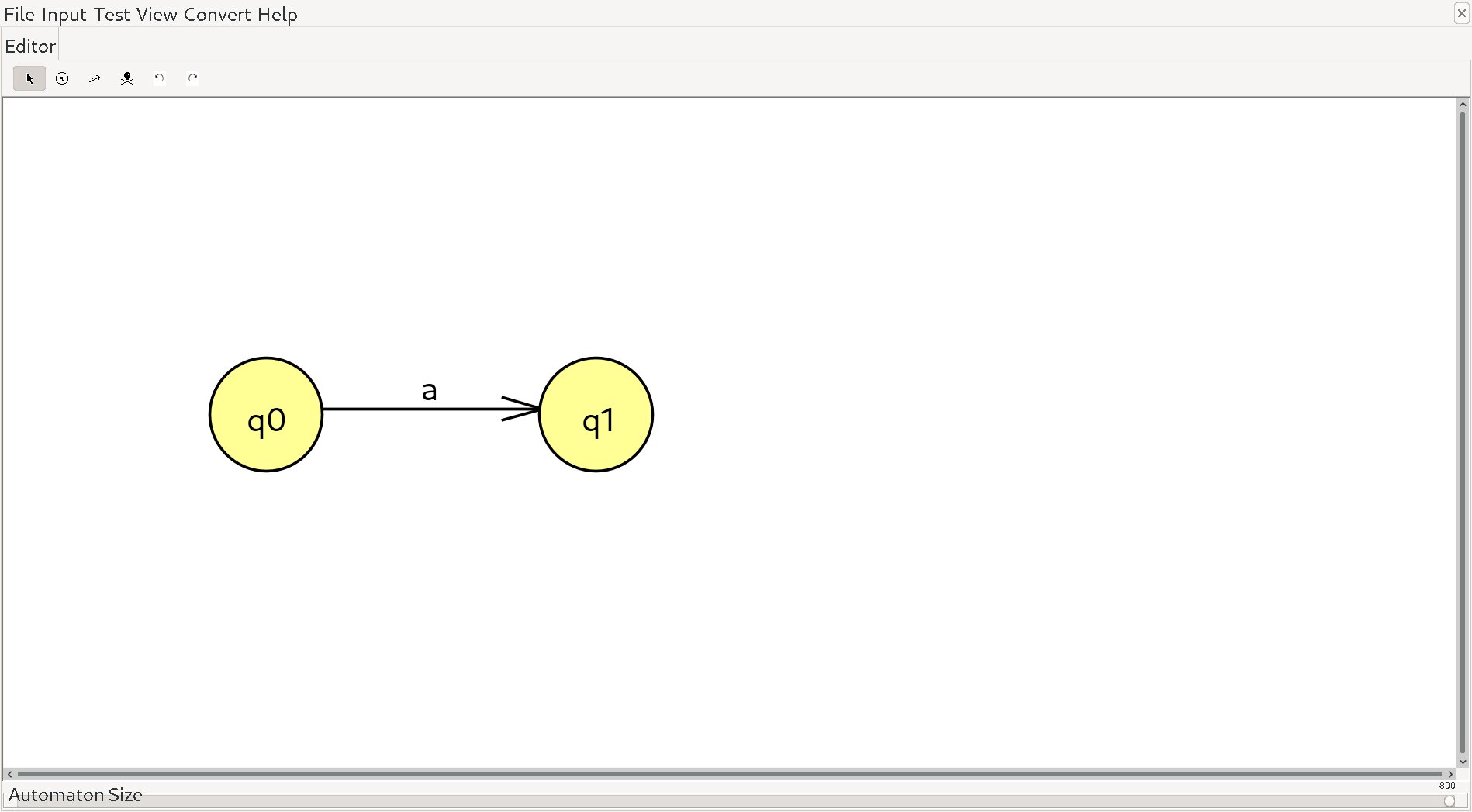
Example of Thompson’s Construction Algorithm:

I will be taking this example from the article I mentioned previously. They use the analogy of a stack which I will try to represent as a sort of black box with no lid. The regular expression that they use (in our RE notation at least) is (aa|b)\*b. The key is to simply read it left to right, applying the 5 rules whenever required.

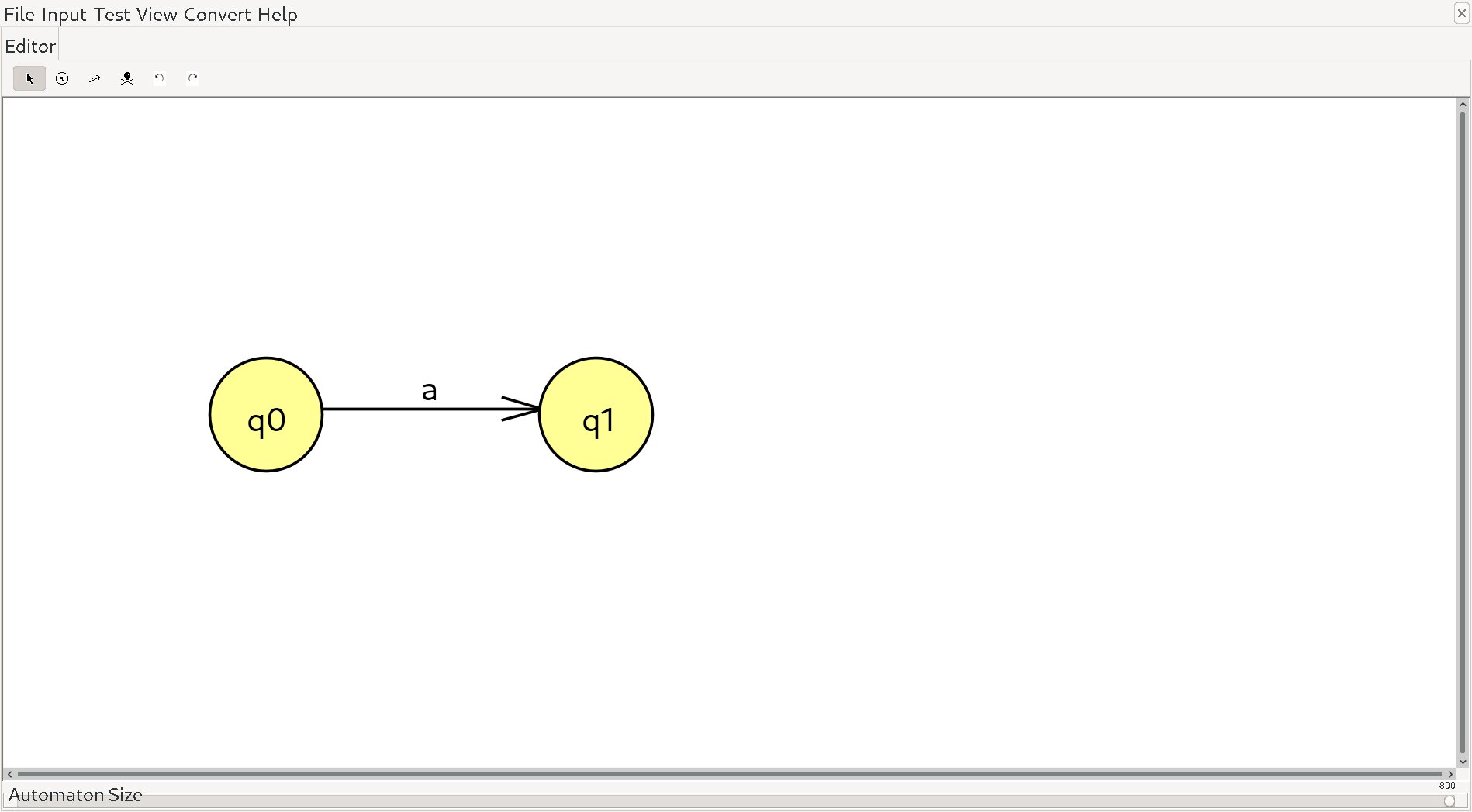
**Step 1:** First we read (aa|b)\*b. The 2nd rule applies here: Any non empty symbol becomes a transitional state:

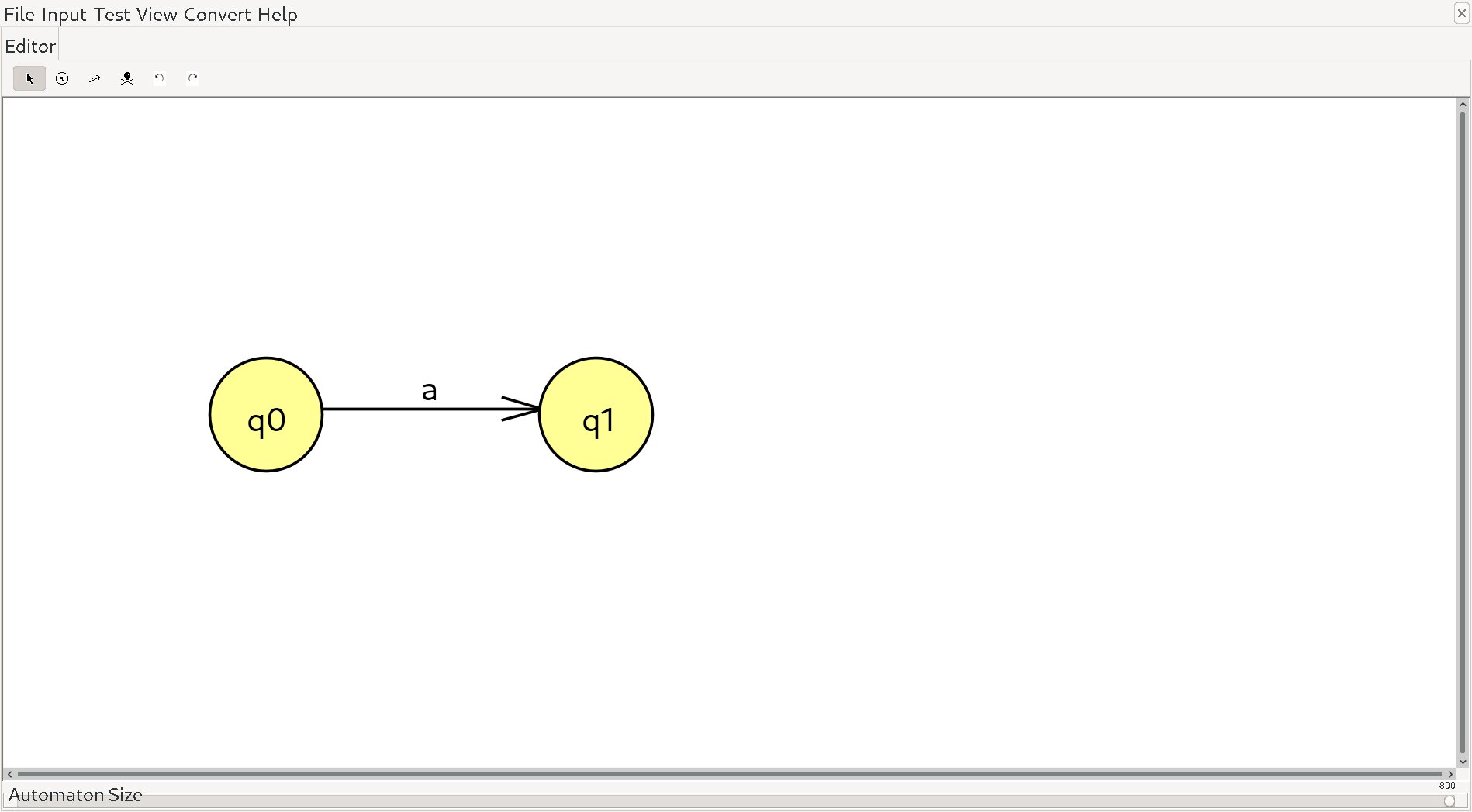


Now, we also need to push this onto our imaginary stack:

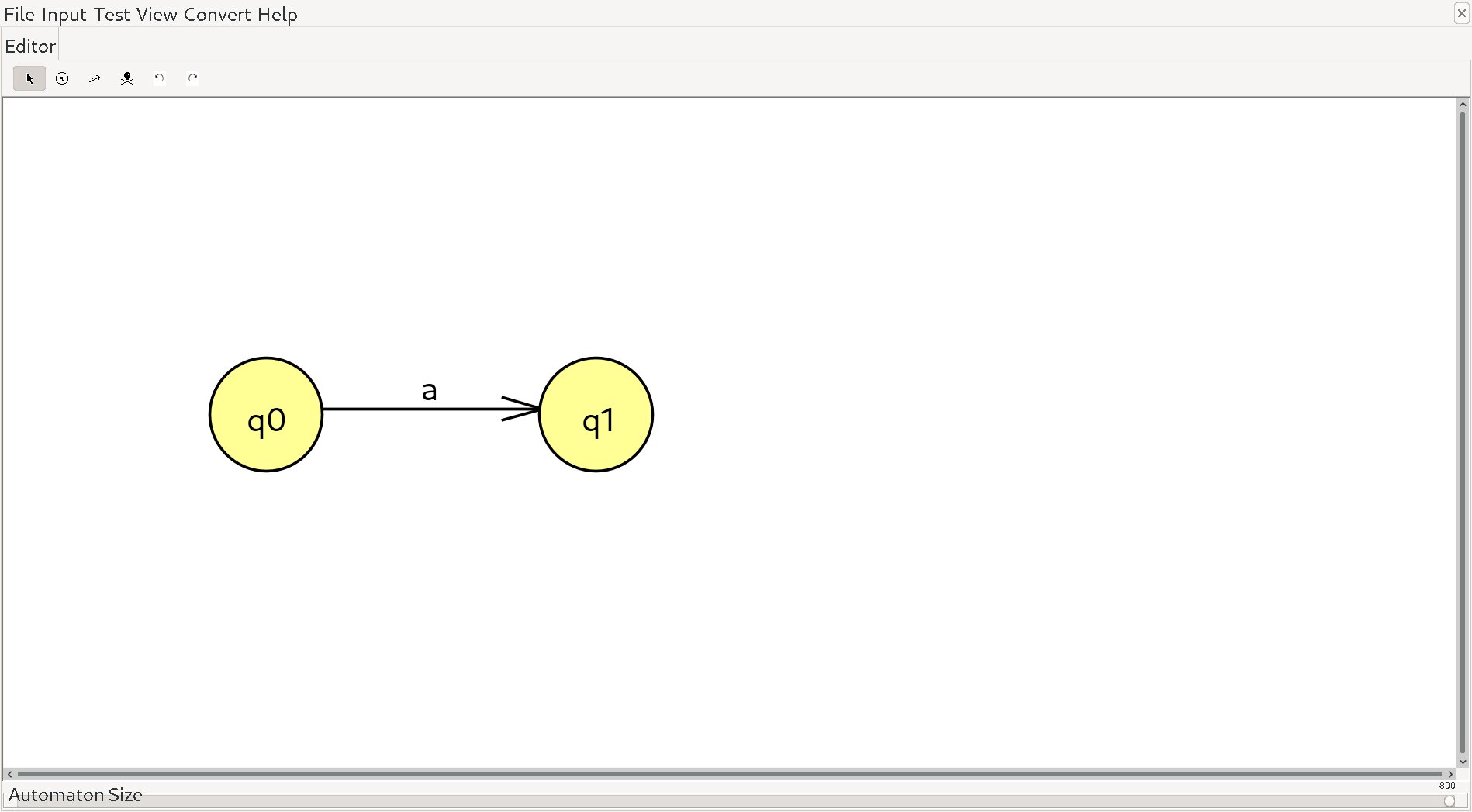
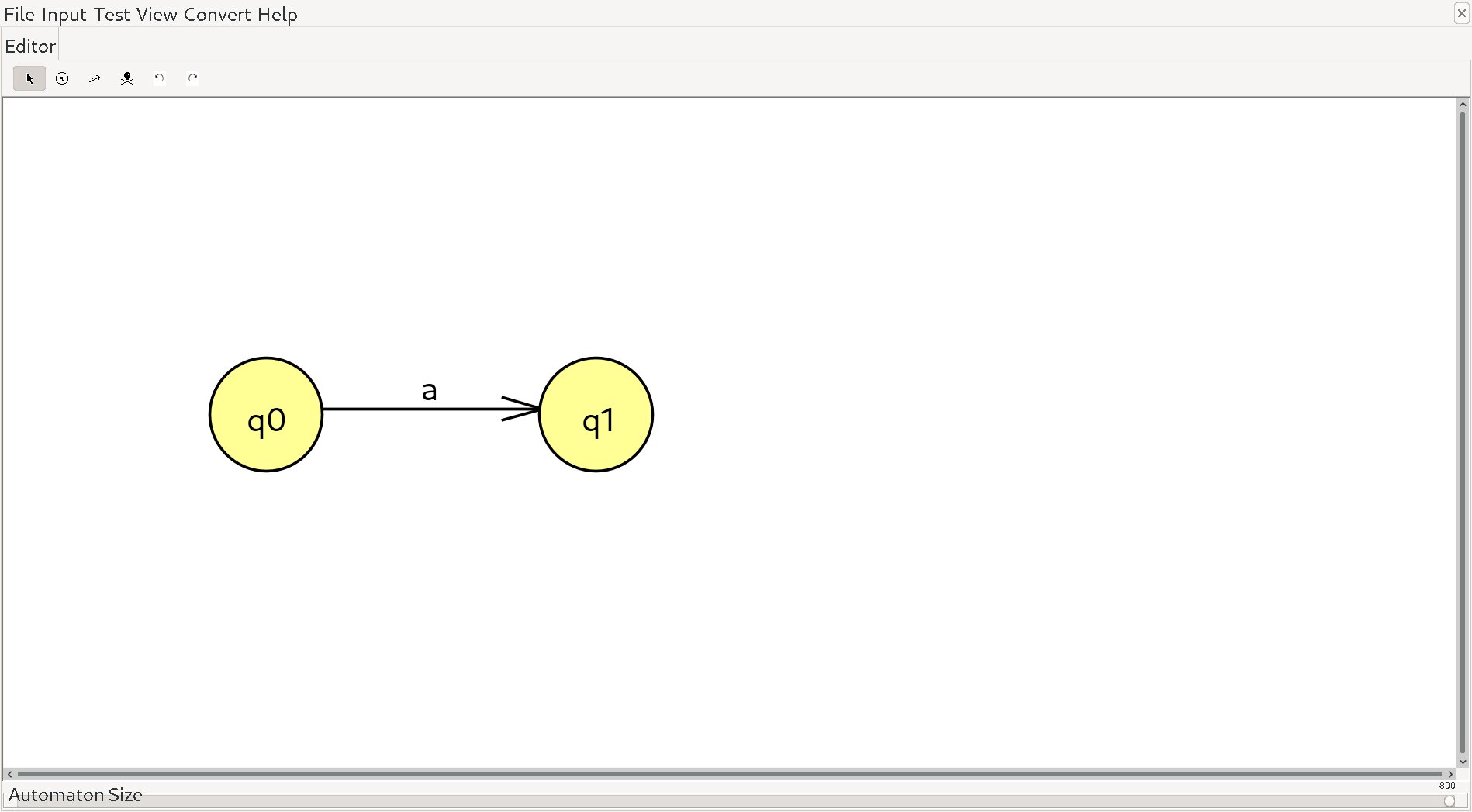
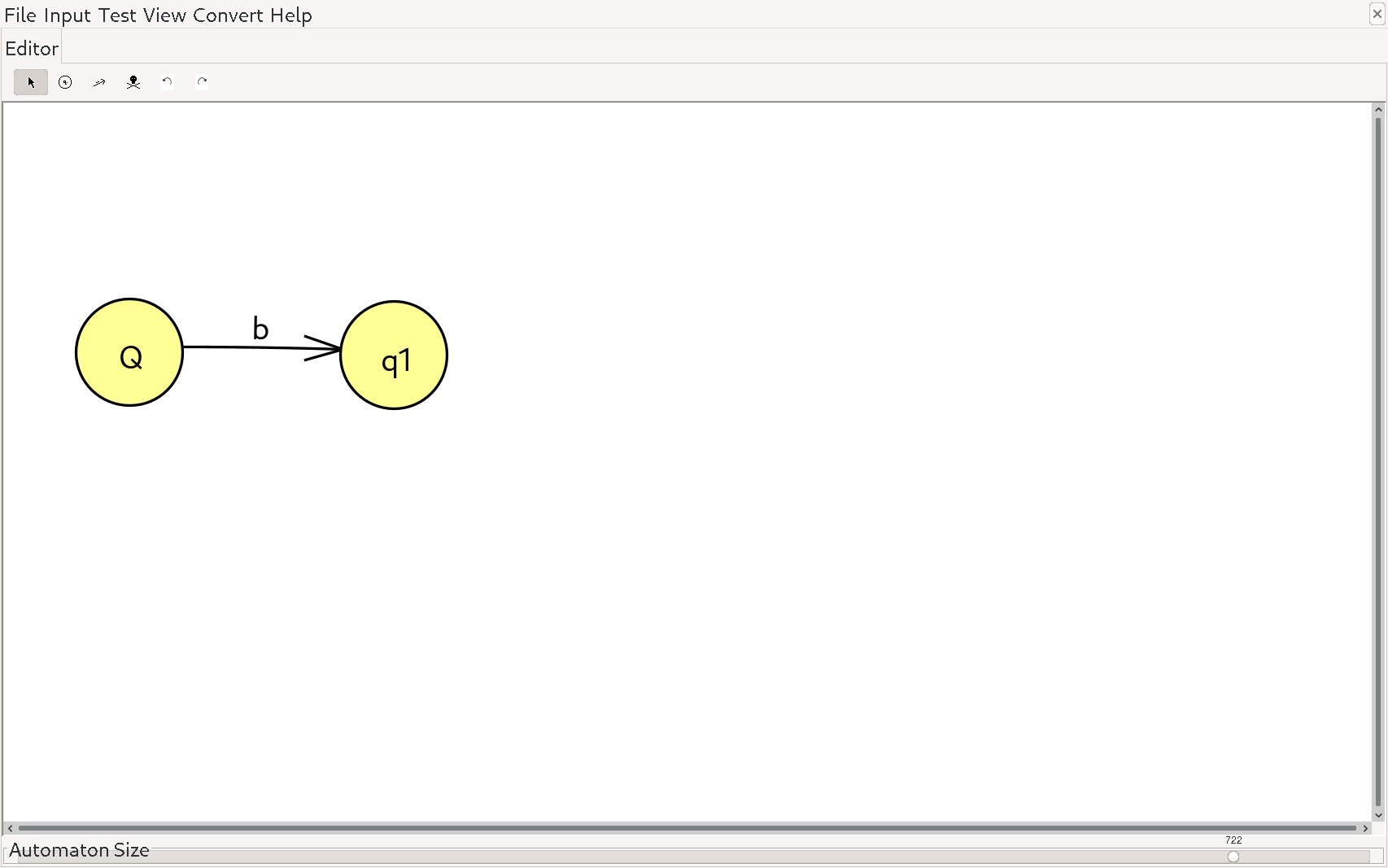


**Step 2:** Next, we read (aa|b)\*b. The transition for this is the same as previously, so we add it to the stack as well:

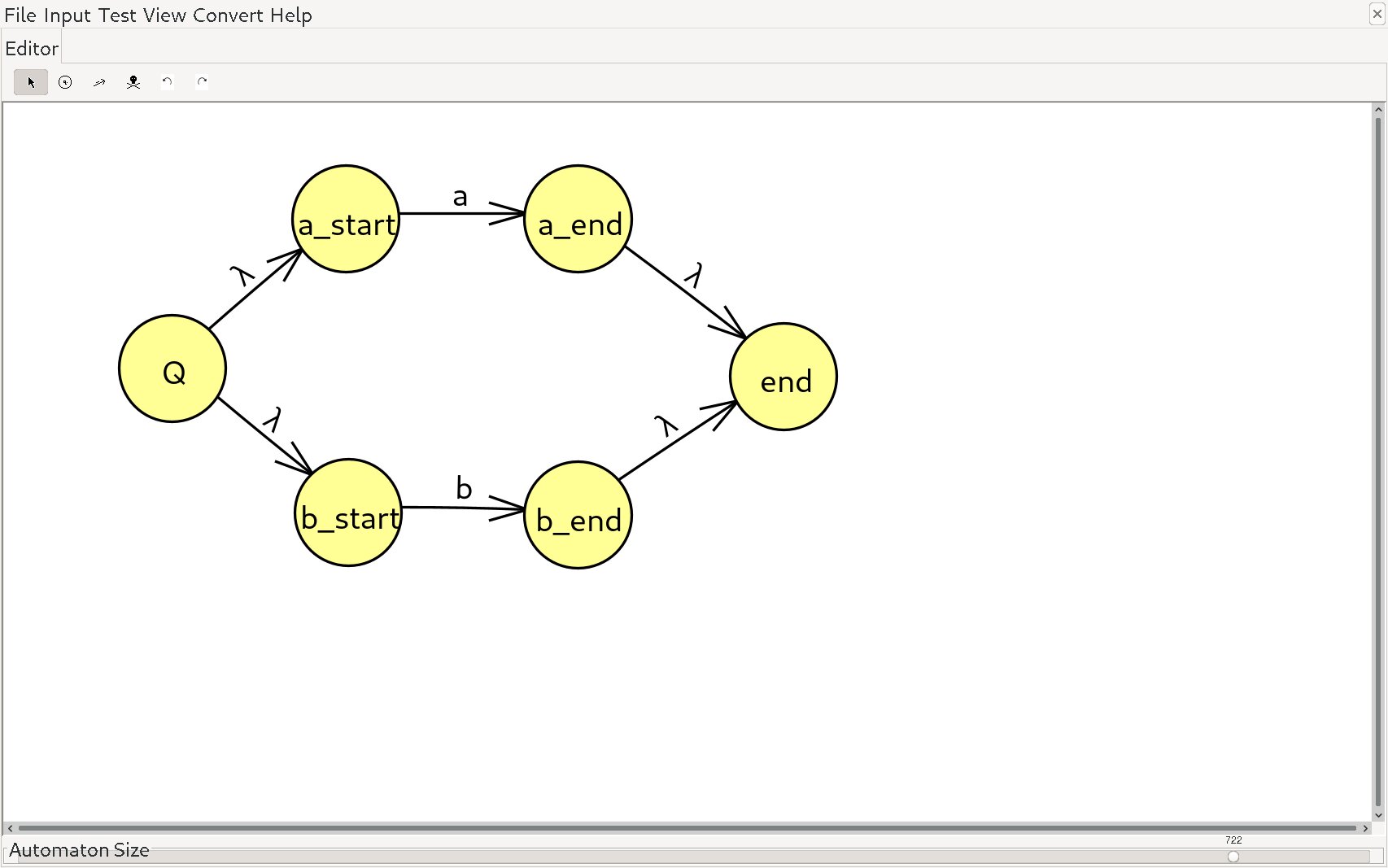




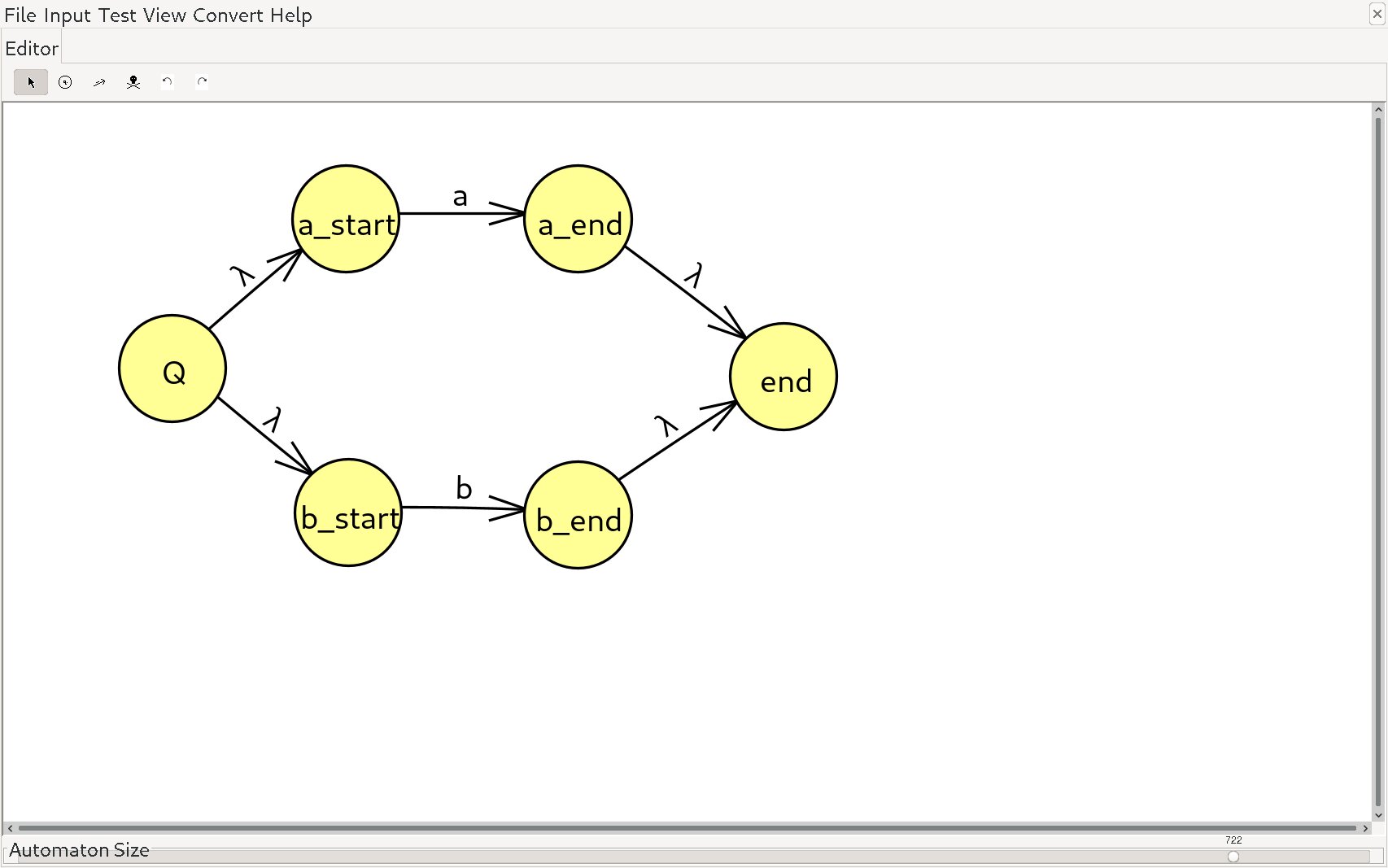
**Step 3:** Next we read (aa|b)\*b, however, keep in mind that there is alternation occuring between b and a. Whenever we find alternation, we need to pop off the previous transition from our imaginary stack, and apply the alternation rule between the two.

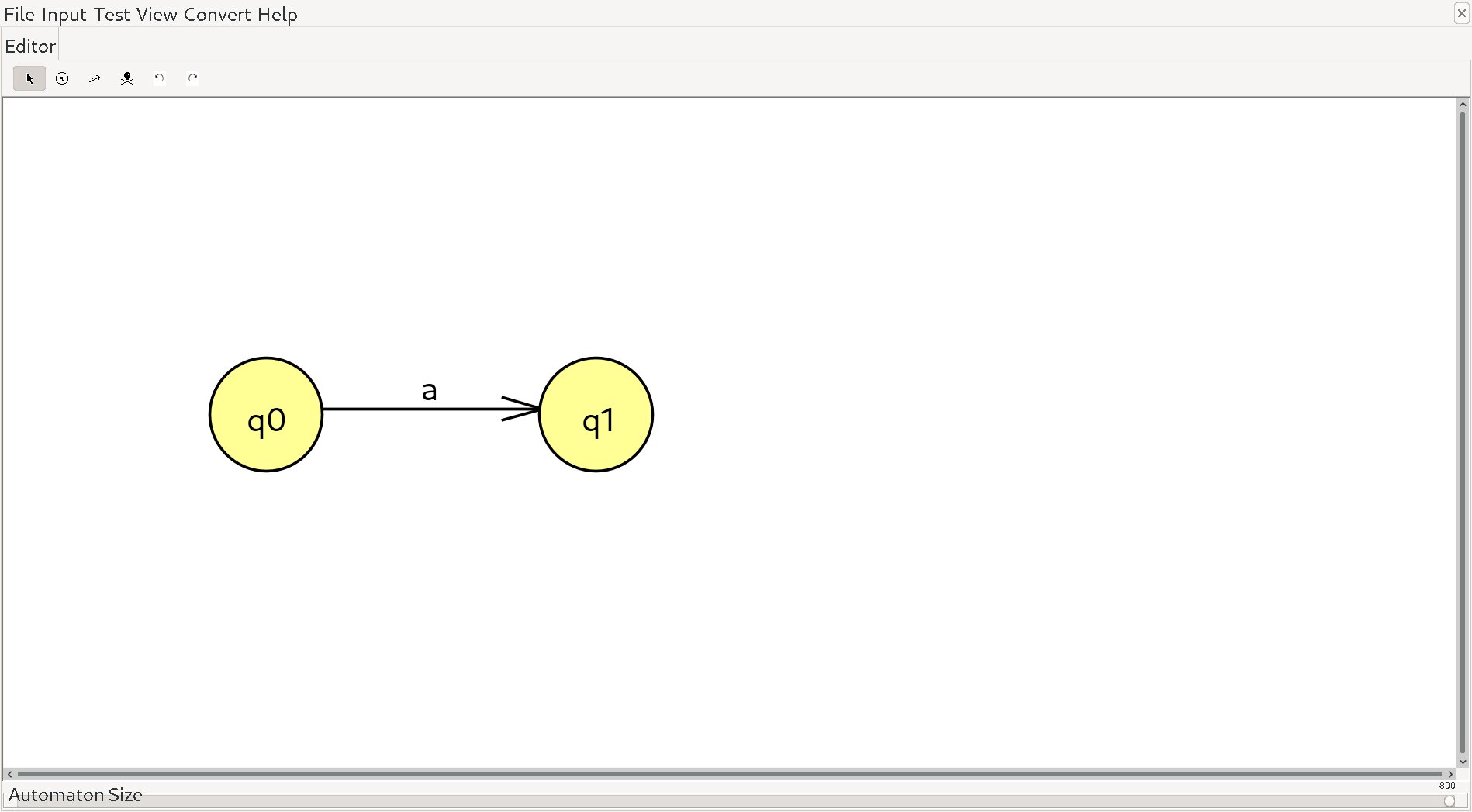
 |

=

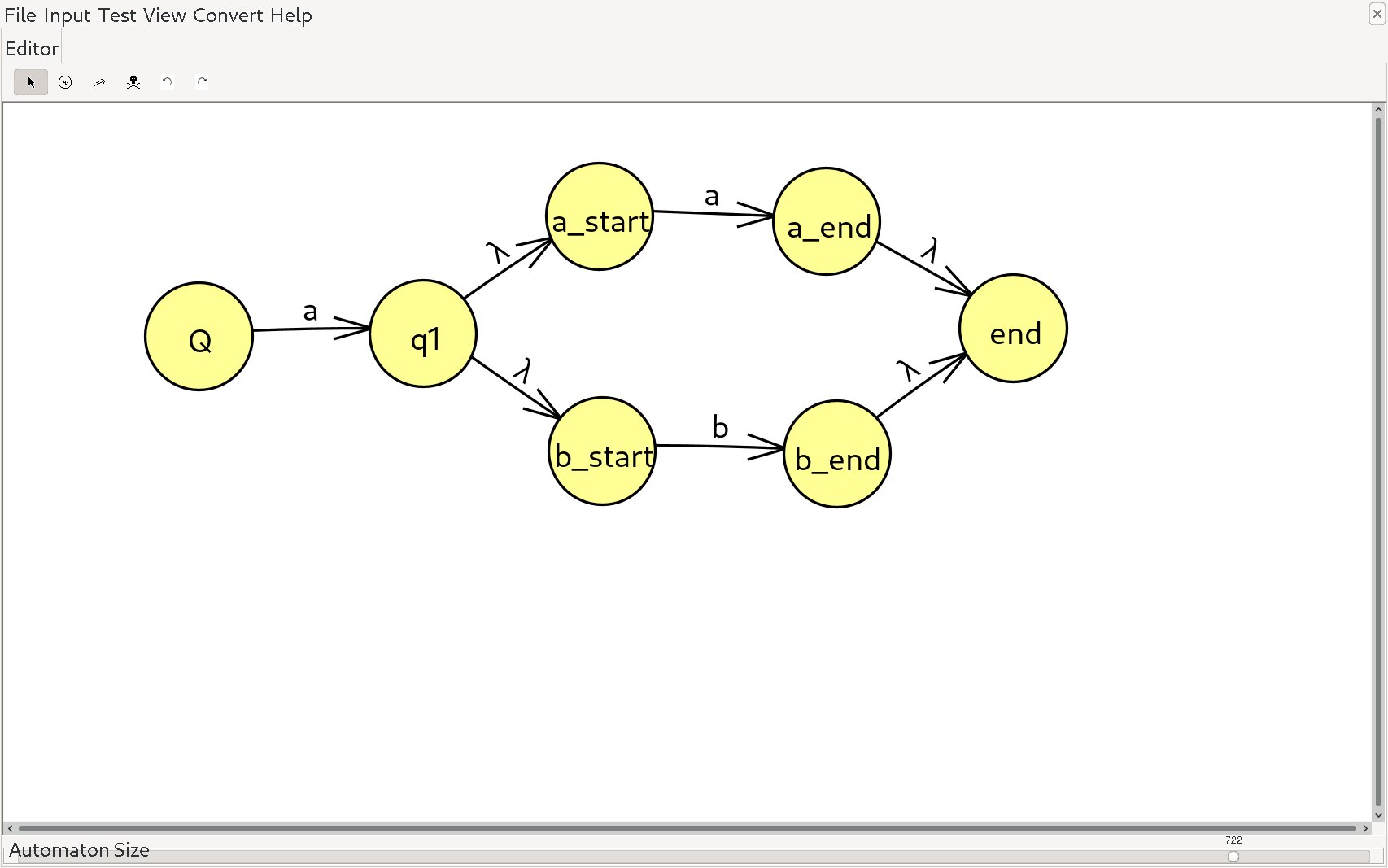


And now, we put this back onto the stack:

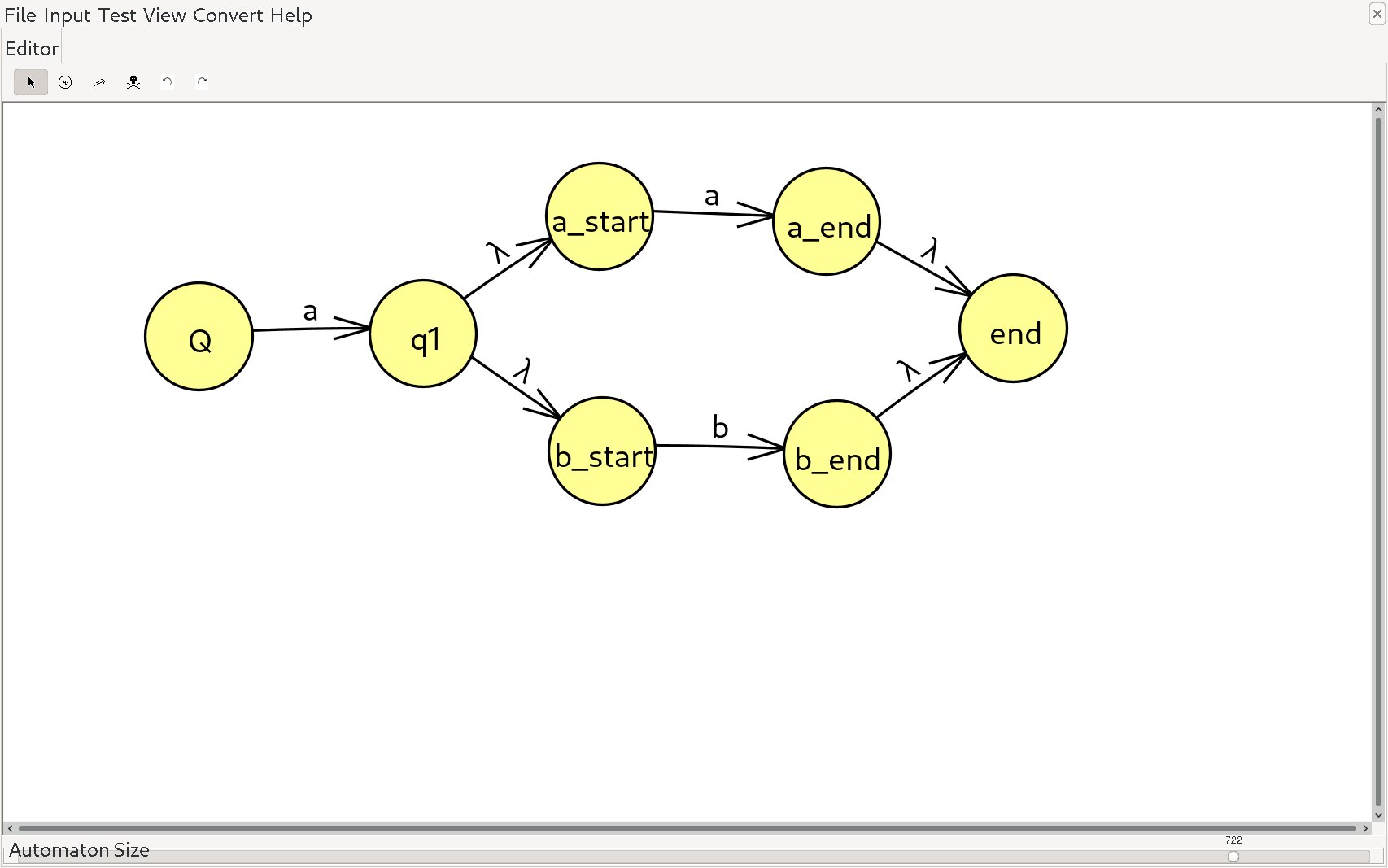




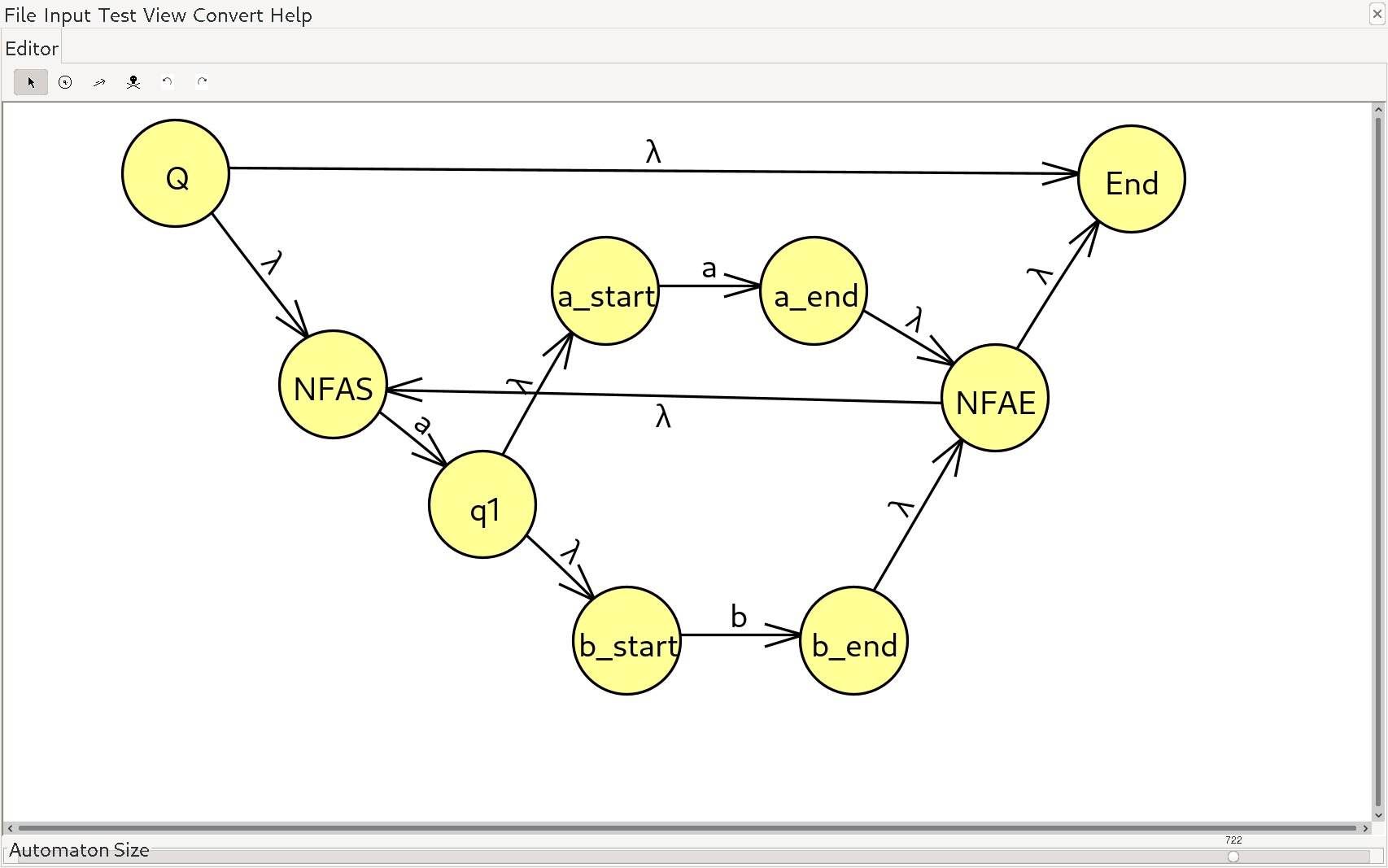
**Step 4:** Now recall that the brackets denote precedence over the kleene closure, so we need to apply the concatenation rule between the two transitions in our stack (aa|b)\*b



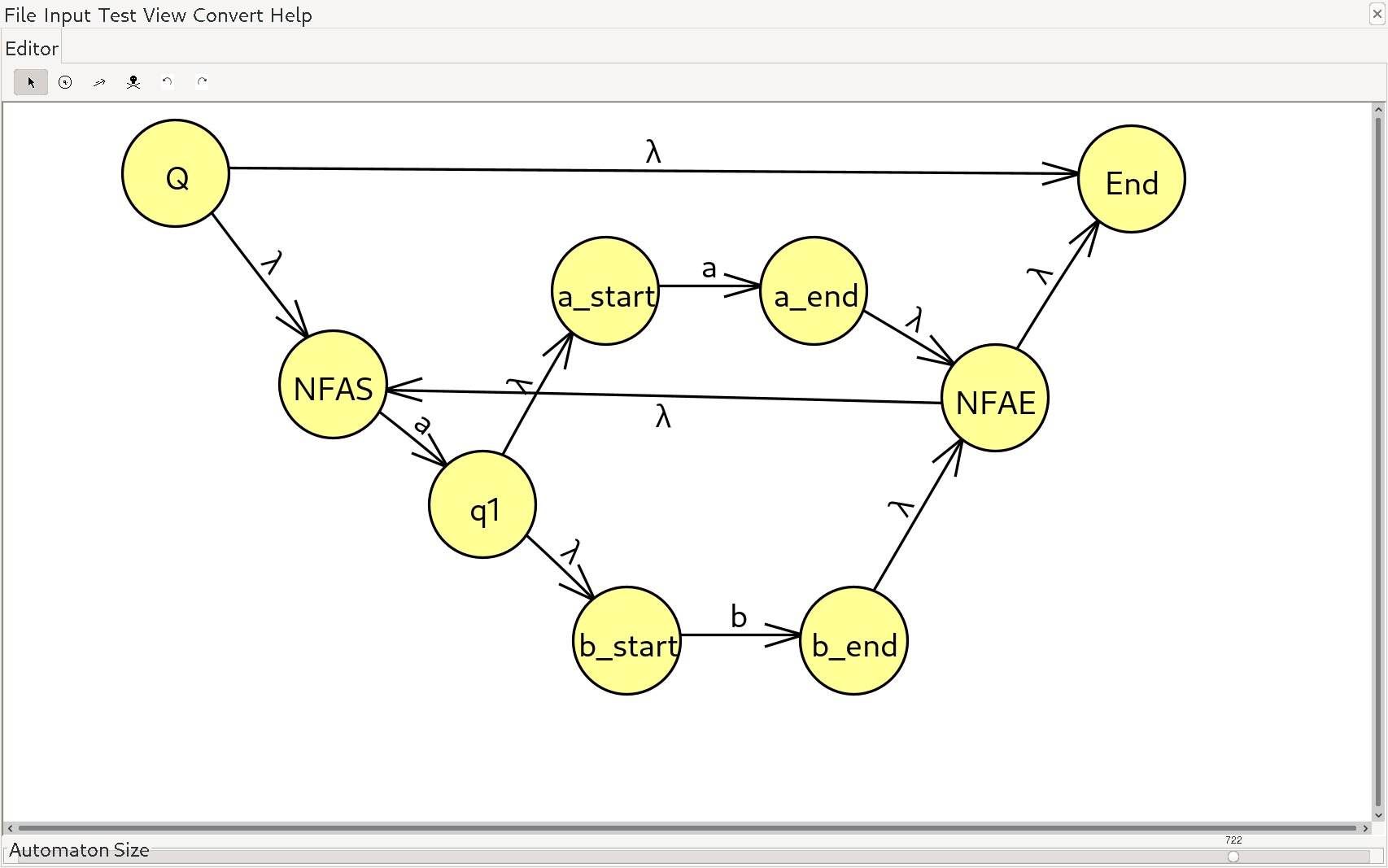
And once again, we only have one item in our stack:



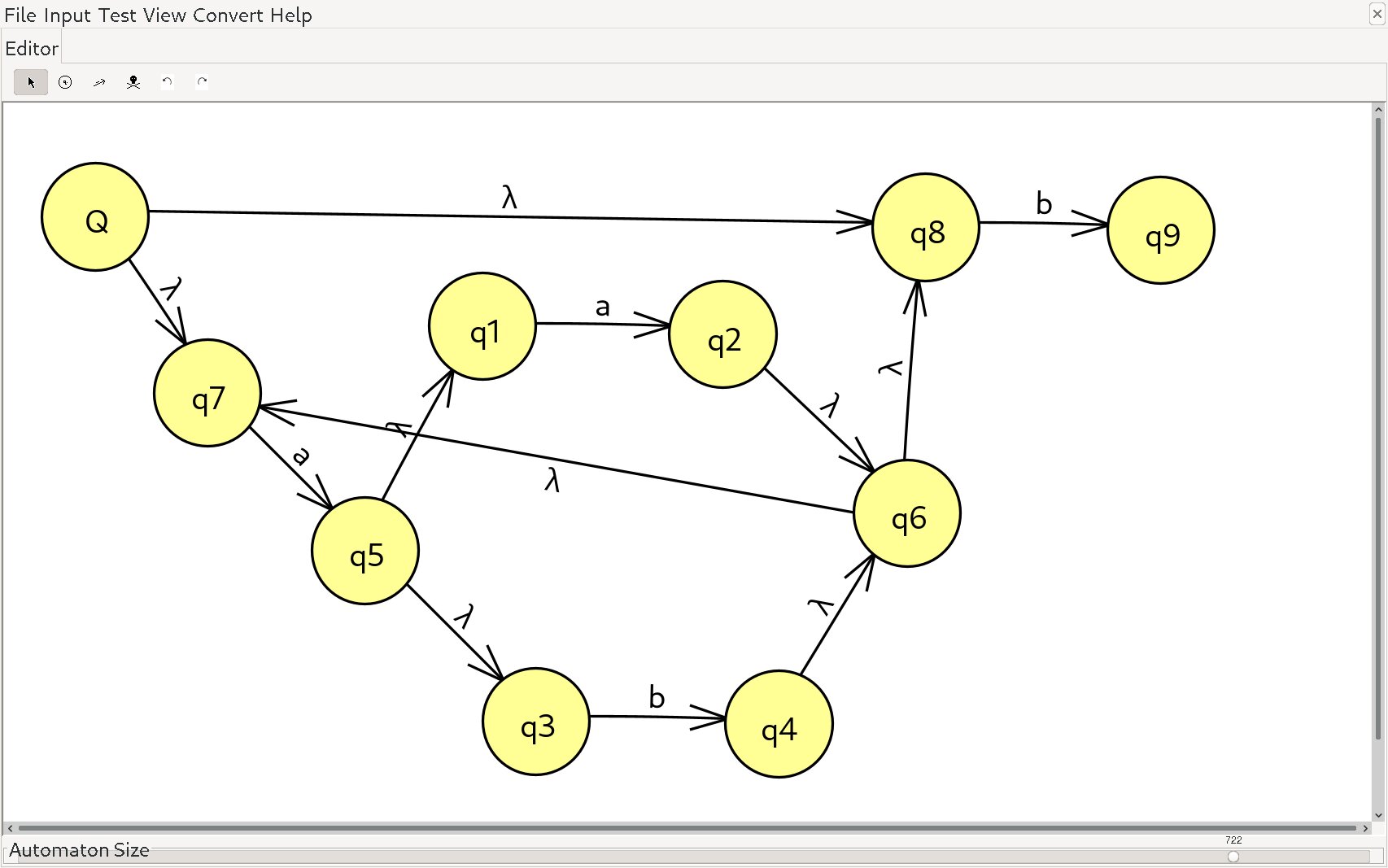
**Step 5:** Now, it’s time to apply that kleene closure to our transition (aa|b)\*b:



Now it’s starting to look messy, but it’s okay because we only have one more step to go. First though, we add this to our stack:



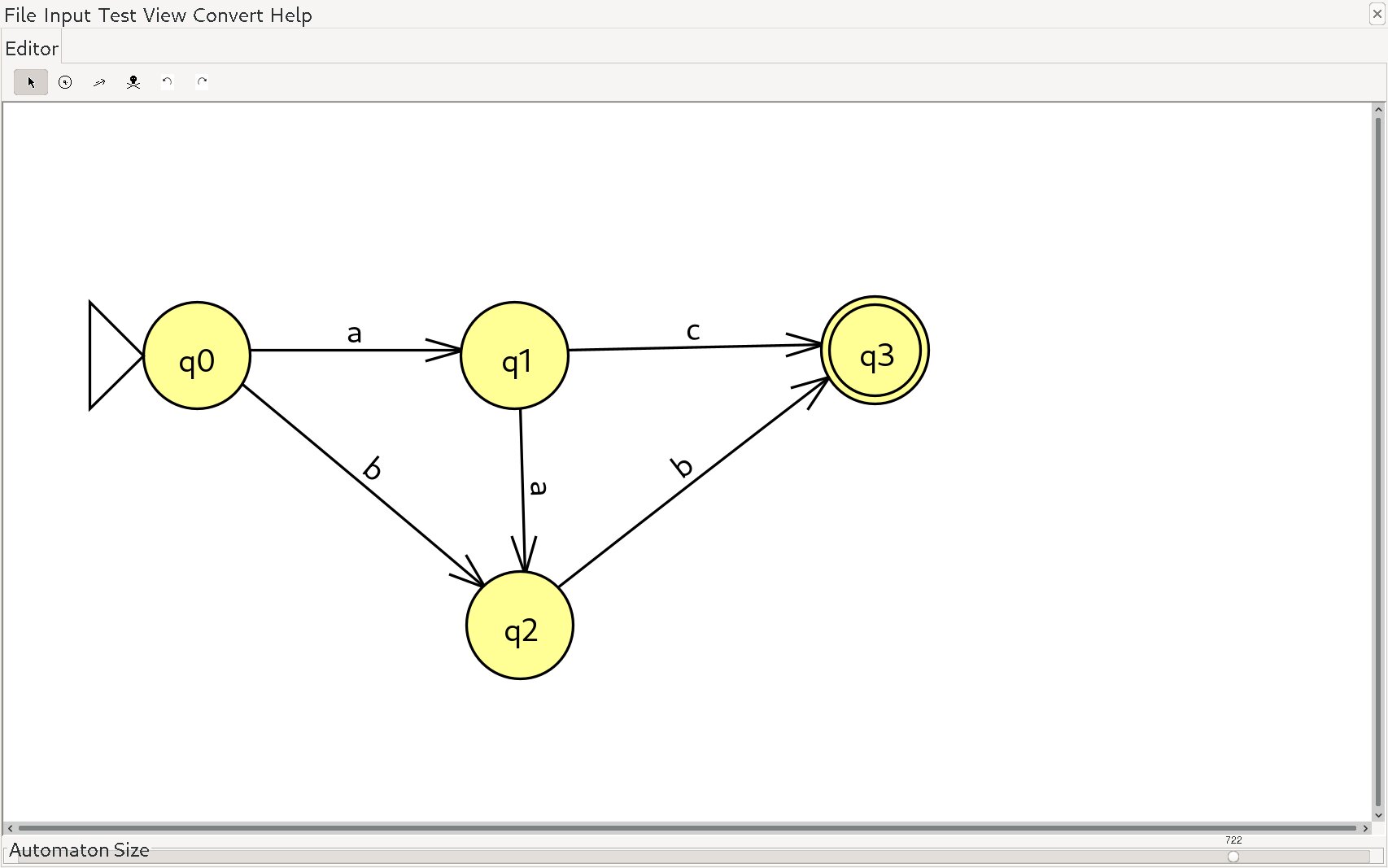
**Step 6:** Finally, we must concatenate what we have with b and we will get our final NFA (aa|b)\*b:



And that’s it! We’ve successfully converted the RE (aa|b)\*b to a NFA. We can input some sequence of letters to test whether or not it is a valid lexeme. For instance, the following sequences should all be valid: “aab”, “abb”, “b”.

Transition Tables:

Note that we will return to the concept of Finite State Automaton and coverting NFA to DFA, however, we need to take a quick detour to catch you up to speed on Transition Tables (TT) aka. Transition Diagrams. Transition Tables are essentially just a 2D grid of columns and columns where columns represent symbols that our automaton might read, and rows represent each state of the automaton. Note that Transition Tables only work for **DFA**. TTs are deterministic by nature since each column must point to a specific state row/state. Transition Tables will be the key to helping us solve the NFA to DFA conversion, but first, let me show you an example of a TT.



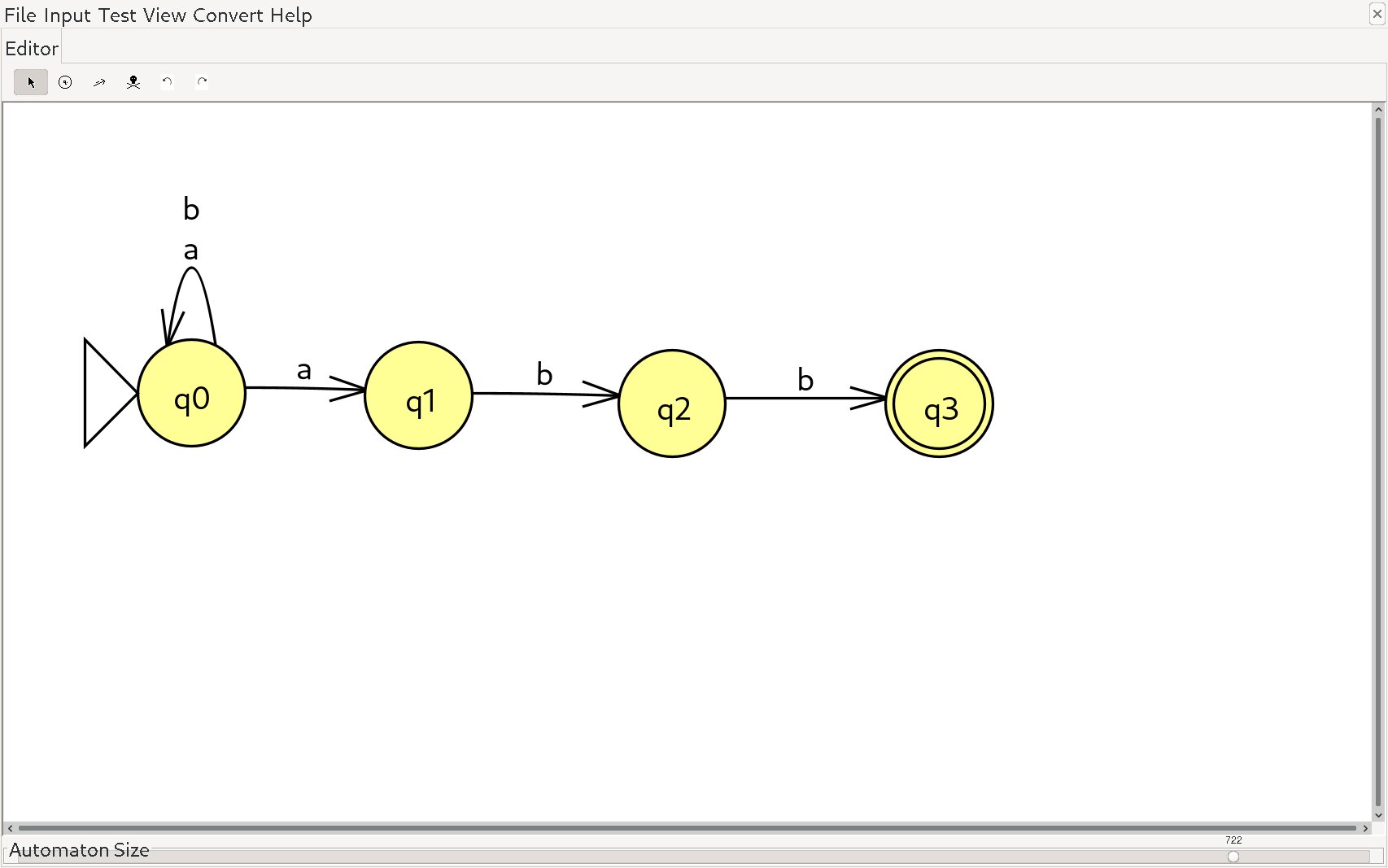
Note that I’ve added two additional details: the arrow represents the start state, Q, and the double circle represents the final accepting state/halting state. Now, since we have 3 symbols that our automaton can read (a, b, and c), we will have 3 columns, one for each. Since we have 4 states, we will have 4 rows.

|  |  |  |  |
| --- | --- | --- | --- |
|  | a | b | c |
| q0 | q1 | q2 | - |
| q1 | q2 | - | q3 |
| q2 | - | q3 | - |
| q3 | - | - | - |

This is quite a basic Transition Table to give an illustration of how it can be implemented. On the left-most column, we have our states {q0, q1, q2, q3} and on the first row, we have our valid symbols {a, b, c}. When we are in state q0, and we read an a, we transition to state q1. When we are in state q0 and we read b, we transition to q2. We do this for all states in the automaton. Notice that final accepting states such as q3 never transition since they are meant to finalize the computation.

Converting NFA to DFA:

In order to convert some NFA to DFA, we typically create a TT and work out the kinks when it comes to ambiguous transitions by creating new states. DFAs will therefore always be larger in size and computational power than NFAs but make up for it through ease of use. We can go over computational costs with Big O momentarily.



Even though this automaton does not contain any epsilon transitions, recall that I stated that the method of identifying whether or not a Finite State Automaton is NFA or DFA depends on whether or not it contains ambiguity between state transitions. In this case, from the start state Q aka q0, if we read a, it is ambiguous as to whether we should transition back to q0 or transition to q1. Be warned, grasping how to fix this ambiguity made my head spin, but after staring at examples online long enough, I feel as though I can serve justice, and present an easy-enough to follow guide. We begin by creating our basic TT. If ambiguity occurs, we create a set of states that are possible to traverse to from the current state. Because the transition from q0 when reading a is ambiguous, we create a new state called {q0,q1} (the two possible outcomes) and label this as state 4.

|  |  |  |
| --- | --- | --- |
|  | a | b |
| q0 (Q) | {q0, q1} (new state 4) | q0 |
| q1 | - | q2 |
| q2 | - | q3 |
| q3 | - | - |

Now, what we want to do is continue comparing the first row, q0, with each consecutive row of the table and adding new states as we find them. Due to the fact that reading a from state q0 can lead us to either q0 or q1, we will add a new state (4) to the table. Then we will compare q0 with q1, q0 with q2, q0 with q3, and so forth, creating new states anytime we find ambiguity. I will highlight which rows we are comparing at each stage as well as narrate what I am doing to provide additional context.

|  |  |  |
| --- | --- | --- |
|  | a | b |
| q0 (Q) | {q0, q1} (state 4) | q0 |
| q1 | - | q2 |
| q2 | - | q3 |
| q3 | - | - |
| q4 = {q0, q1} | q4 = {q0, q1} | q5 = {q0, q2} (state 5) |

So, as I mentioned, we compare q0 with q1 to begin. Since there is no transition at q1 when reading a, the unionized set {q0, q1} U {} = {q0, q1} ie. nothing changes and we place that at our new state 4. Then we compare q0 and q1 at row b. We union {q0} U {q2} = {q0, q2} which is a new state that we’ll label q5. This will be added to our table in the next iteration and this time we will be comparing q0 with q2.

|  |  |  |
| --- | --- | --- |
|  | a | b |
| q0 | {q0, q1} (state 4) | q0 |
| q1 | - | q2 |
| q2 | - | q3 |
| q3 | - | - |
| q4 = {q0, q1} | q4 = {q0, q1} | q5 = {q0, q2} (state 5) |
| q5 = {q0, q2} | q4 = {q0, q1} (state 4) | q6 = {q0, q3} (state 6) |

Perhaps now you can see the pattern if you weren’t beforehand. We compare q0 with q2 at row a. The union is {q0, q1} U {} = {q0, q1} which is state 4 as we’ve established. Another ambiguity arises in column b between {q0} U {q3} = {q0, q3} which we label as a new state q6.

|  |  |  |
| --- | --- | --- |
|  | a | b |
| q0 | {q0, q1} (state 4) | q0 |
| q1 | - | q2 |
| q2 | - | q3 |
| q3 | - | - |
| q4 = {q0, q1} | q4 = {q0, q1} | q5 = {q0, q2} (state 5) |
| q5 = {q0, q2} | q4 = {q0, q1} (state 4) | q6 = {q0, q3} (state 6) |
| q6 = {q0, q3} |  |  |

Note: Finish section

Comparing NFA and DFA with Big O:

Hopefully you are an efficiency nerd like myself, but if not, this section will be short so do not fret. Let us say that we have a particular regex, r, and a particular input string that the scanner is reading, x. The length of r is denoted as |r| and the length of x is denoted as |x|. In terms of time complexity, NFA will always take longer because the processor must consider all possible transitions and choose one based off some criteria. As for DFA, there can only ever be one transition per state, making it fairly efficient. The Big O for time is as follows:

DFA: O(|x|)

NFA: O(|r| \* |x|^2)

In this case, DFA is O(|x|) because it will begin at the start state, Q, and transition over each symbol in the input string once. As for NFA, well I don’t actually know what makes it |r| \* |x|^2, but the point is that it takes longer to transition through.

In terms of how many non-terminals the Finite Automaton must traverse through, DFA starts off better, but NFA will quickly surpass DFA as the size grows exponentially for DFA, and linearly for NFA. Therefore, the Big O for how much space NFA and DFA occupy is as follows:

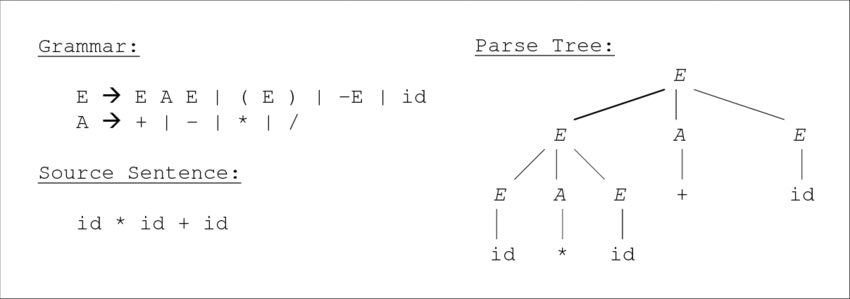
DFA: O(2^|r|)

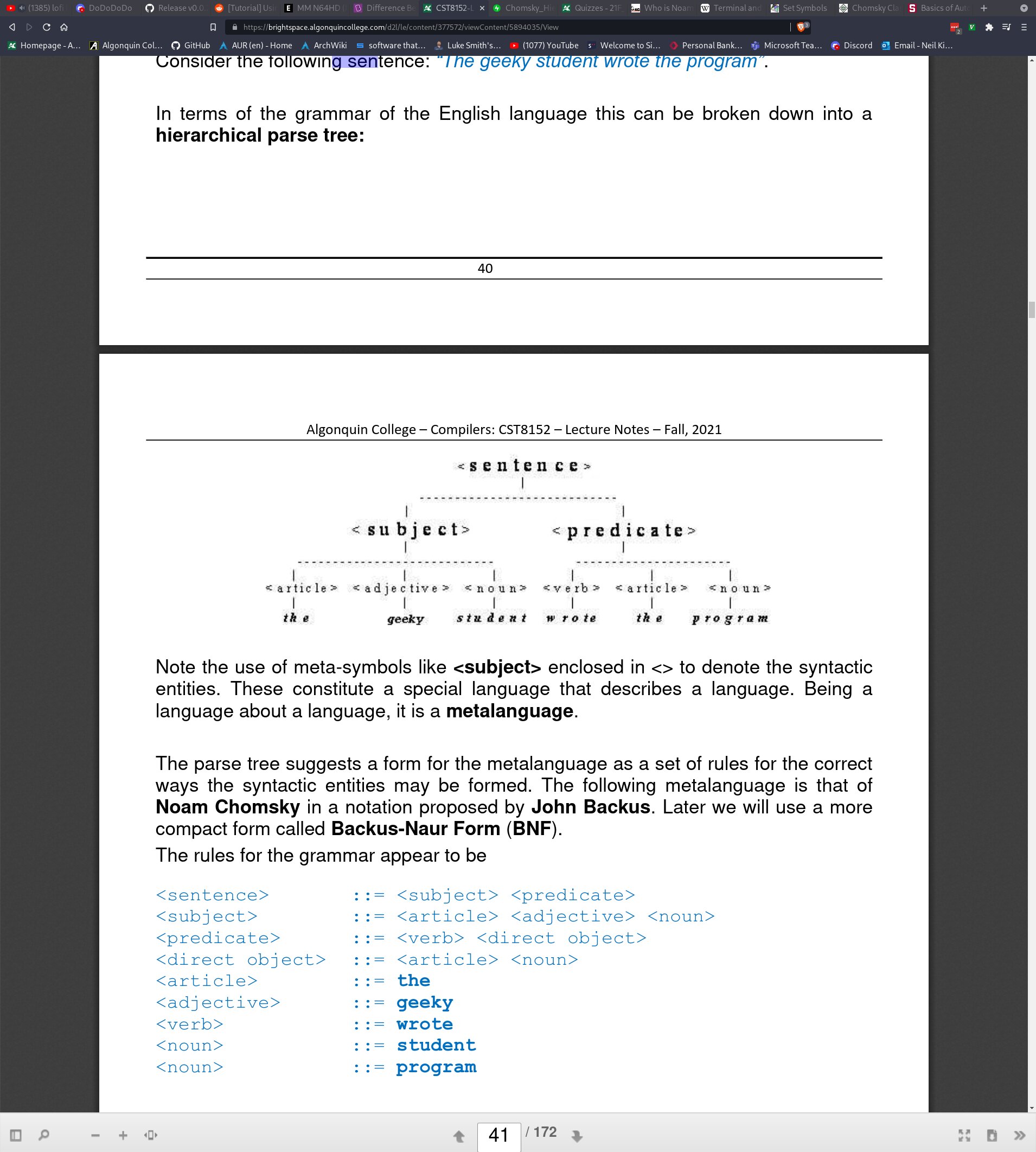
NFA: O(|r|)

The Syntax Analyser/Parser:

Believe it or not, that is actually it for the scanner. To summarize the previous section, we can most easily describe lexemes via RE, DFA, or TT. These are all methods of identifying valid lexemes which will become tokens within our symbol table. As a quick reminder, the syntax analyser or parser phase of the compilation pipeline is designed to evaluate a sequence of tokens and enforce a grammatical structure upon them. We can enforce the structure using Content Free Grammar (CFG). If you recall from Noam Chomskey’s diagram of grammars, and which automaton can understand them, CFG is recognized by Pushdown State Automaton. But we can disregard this notion momentarily, as it is not important for describing CFG itself yet. The end goal of the parser is to create what we call a parse tree. Think of an inverted binary tree, only the tree does not necesarily need to be a binary tree, and it does not need to be balanced like how it ought to be when discussing algorithms. I’ve already subtly introduced this tree to you when we discussed Finite State Automaton. A grammar labled G can be said to be made up of 4 parts: G(Vt, Vn, P, S) where Vt is the set of terminals (leaf nodes of the parse tree), Vn are the non-terminals of the parse tree (parent nodes excluding the root node), P are the productions ie. rules which describe the connections between nodes, and S ie. the start state, or start node.

Below is an example of a parse tree being implemented. Note that the notation used here will bear a striking ressemblance to how we defined lexeme classes when we discussed RE. Well in essence, it’s because this is the same thing that we did for regular expressions; after all, REs are a type of grammar. However, we are now discussing Context Free Grammar, so while it does share similarities with Regular Grammar, it is not the same.

  
Let’s take a look at a more familiar parse tree, even if you are not proud of your English class grades from high school. Below, we Have a grammar G(Vt, Vn, P, S).



The set Vt is the set of terminals. For this grammar, Vt = {“the”, “geeky”, “student”, “wrote”, “the”, “program”}. The set Vn is the set of non-terminals. Vn = {<sentence>, <subject>, <predicate>, <article>, <adjective>,<noun>,<verb>}. The start symbol is <sentence>. In other words, this is essentially what we are hoping to build as we work our way from the bottom up.

Productions:

Productions are essentially rules. If you are familiar with make files in C, you may be familiar with the fact that rules are made up of recipes. Productions are quite similar to this idea of a recipe, where, given ingredients (Vt, Vn), we can generate our target, S. The notation that we use to describe these productions is called Backus Naur Form (BNF) and it is quite simple. In BNF notation, non-terminals are called “syntactic entities” and they are always enclosed in triangle braces <>. We also have a special symbol ::= which means “comprised of” or “made up of”. Terminals are still referred to as just terminals. We can use the OR operator ‘|’ to state that a non-terminal can be comprised of one thing OR another thing. Also note that terminals cannot have production rules. You could argue that they are comprised of a sequence of characters since they are our lexemes, but as far as the parser is concerned, they are the leaf nodes of the parse tree and are the base units of defining our grammar. Here are the production rules in BNF notation for the previously defined grammar:

<sentence> ::= <subject><predicate>

<subject> ::= <article><adjective><noun>

<predicate> ::= <verb><direct object>

<article> ::= The

<adjective> ::= geeky

<verb> ::= wrote

<noun> ::= student | program

As you can see, we only need 7 productions in order to create a grammar which can enforce the sentence “The geeky student wrote the program”. However, there is still a flaw. Notice that a noun can either be the lexeme “student”, or the lexeme “program”. This means that the sentence could reasonably be “The program wrote the geeky student” as far as CFG is concerned. In this case, our meta-language fails because it would consider both sentences to be syntactically correct when only the first is. Checking the validity of the sentence would be the job of the semantic analyser.

The Symbol Table:

We briefly discussed the symbol table early on in the document. To give a refresher, the symbol table (sometimes called a dictionary), records the relationship between attributes and names. For example, the symbol table would be in charge of recording variable names and their corresponding attributes eg. value, data type, address, etc. From a theoretical viewpoint, it can be thought of as a database where records are stored so that they may be accessed at a later time. This is important because a variable may be referenced multiple times during a programs lifespan, so it’s attributes must be stored.

In terms of storing records, there are various methods in which this can be accomplished:

**By scope –** We can record entities according to their scope. We may have multiple inner tables which correspond to global entities, functions, or conditional statements like if statements or for loops.

**By Ordered List –** An ordered list is a simple way of storing records. Simply add the variable that was last defined at the end of the list. This gives better access/read times since we can know where the variable exists in the list, however, it reduces write times since we require some sort of sorting algorithm.

**By Hash –** Likely one of the best options is a hash map. This allows for faster read and write times (O(1) commonly).

In terms of writing a symbol table manager system, we often want to consider including the following functions:

\* Create a new symbol table

\* Insert a record into symbol table

\* Lookup/find a record

\* Update a record

\* Delete a record

\* Delete/destroy symbol table

\* Misc: sort, print, stack

LL and LR Grammars:

Grammars can be sub-categorized into LL and LR grammars.

**LL:** Input to the parser starts at the left of the input string, and the leftmost derivative is taken.

**LR:** Input to the parser starts at the left of the input string, and the rightmost derivative is taken.

Typically, a number is added at the end of LL or LR, indicating how many tokens that the grammar is reading at a time. For example, LL(1) grammars look ahead by one token. Most programming languages use both LL and LR when implementing a particular grammar.

Derivations:

Earlier, we took a look at the parse tree. However, there is another way in which we can think about the parser, namely, as a series of derivations. A derivation essentially means to take the syntactic-entities (the left hand side of our BNF notation) and see what non-terminals/terminals it is composed of (anything to the right of the ‘::=’ symbol). Instead of using BNF notation, we use a much more mathematical looking notation. Non-terminals generally take on the form of a capitilized letter such as ‘A’, and any terminals are denoted by lower case letters eg. ‘b’.

For example, we could describe a production for an expression as <Expr> ::= <Expr> + <Term> in BNF, but in terms of derivations, this can be written as E => Eα where E is the non-terminal “expression”, and α is “+ T”. Greek letters typically denote some sort of grouping of non-terminals and terminals. In this case, ‘+’ is a terminal, because it is it’s own unique token, presumably. T is a non-terminal, because a Term is composed of other non-terminals/terminals. Therefore α is a grouping of the terminal + and the non-terminal, T.

The aim of derivations is to end up with a particular sentence. This is similar to how the aim of the scanner was creating tokens, except now we are creating a valid sequence of tokens. The final output of the parser does not necesarilly need to be semantically correct (that is the duty of the semantic analyser), however, it should be syntactically correct ie. sentences generated through derivations should match their production rule.

Moving back to LL and LR grammars, knowing what we know now, let’s try to comprehend the definitions a bit better. For LL, we said that we start at the left of the input string, and take the leftmost derivation. Given the following grammar:

S -> cAd

A -> aXb | X

X -> e | f

and given the input string cafbd, we want to check that cafbd is a valid string. Remember, cafbd is not a sequence of lexemes ie. symbols. It is a composition of individual terminals ie. tokens.

With LL, the way that this is achieved is to start at the left of the input string (first L in LL) which is ‘c’. S is our root/start node in this case. It only has one derivative, cAd, so this is the leftmost derivation (second L in LL). Note that it starts with c which is important. If it didn’t begin with the first letter of our input string, we would have to continue searching for a match. cAd is composed of the terminal c, non-terminal A, and terminal d. Since A is a non-terminal, we next look at the production rule for A to see what other derivatives we can replace it with. The left most derivative of A is aXb. The ‘a’ in aXb matches the ‘a’ in our input string, so once again, this is acceptable. Now we are at caXbd. X is a non-terminal and looking at it’s definition we have 2 options. We look first at the left-most derivative, but e does not match with the third letter of our input string ‘f’. Luckily, the second left-most derivative is an f, so we end up with cafbd. In other words, our input string is a valid sentence! This can be written more precisely as S => cAd => caXbd => cafbd where each arrow denotes a derivation.

Now let’s look at LR. Given the following grammar:

S -> aABe

A -> Abc | b

B -> d

and given the input string abbcde, we want to check if this is a valid input string. In order to do this, we start at the left of the input string again (first L in LR) which is the letter ‘a’. Once again, S is our start/root node. The right-most derivative of S is aABe which matches the first letter a with the first letter of our input string. But now we have a dilema! aABe has two non-terminals. Because we are using LR, we take the right-most derivation first ie. B. B is composed of the terminal d, so our new string is aAde which still matches with our input string so far. A is our only non-terminal, so the right-most derivation will be on A. A is composed of either Abc or b. We look at the derivatives from left to right and get the string aAbcde. Finally, we take the right-most derivative once more which will be on the non-terminal A. Since aAbcbcde doesn’t work, we look to the next derivative and get b which matches our string. The simplified version is as follows: S=> aABe => aAde => aAbcde => abbcde