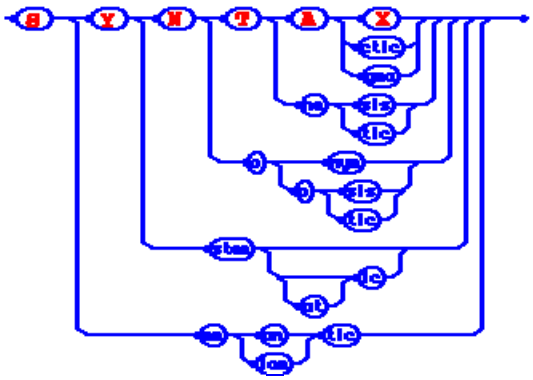
REGULAR EXPRESSIONS, BNF, SYNTAX DIAGRAMS AND REVERSE POLISH NOTATION



REGULAR EXPRESSIONS

NATURAL LANGUAGE

In our everyday life, we learn one or more languages as a child. Regardless of which one, they all have features in common. These are called natural languages and are usually both spoken and written by humans. It is probably a good idea to look back at the section on Declarative languages such as Prolog from the earlier specification.

Natural languages have two main features in common:

- They comprise a set of words (or 'pictures representing words' for some languages)
- They are governed by a set of rules that define the word order for both written and spoken communication. These rules define the grammar (or syntax to use a more technical term) for the language.

A construct may be valid but still lack meaning. Using an example from a previous AQA CPT4 paper, "The peanut ate the monkey" is a valid construct but lacks meaning.

What is the difference between syntax and semantics? Not only do we need syntax rules to construct sentences, we need to consider the relationship between sentences and real world concepts as well. Semantics defines the actual meaning. Consider the Natural Language sentence "My bicycle needs oiling badly". Syntactically it is correct in English but what does it mean? Does it literally mean that my bike needs oiling but not to be carried out very well or does it mean that my bicycle is in urgent need of oil?

Constructing rules for a natural language is difficult. I once experimented with a simple natural language translator program from English ↔ German. When I wrote the source phrase "My bicycle needs oiling badly" and then auto translated it to German, it produced a phrase in German that was syntactically correct but with the wrong meaning i.e. translating it back manually to English I got "My bicycle needs oiling not very well!!

Natural languages are thus often ambiguous. This is why we must use *Formal* languages to communicate with computers. Natural language auto translation from Russian to English was an important research area (not well done) during The Cold War.

FORMAL LANGUAGES

Formal languages are studied in the fields of logic, computer science and linguistics. Their most important practical application is for the precise definition of syntactically correct programs for a programming language. The branch of mathematics and computer science that is concerned only with the purely syntactical aspects of such languages, i.e. their internal structural patterns, is known as **formal language theory**. Formal languages are defined using two parts, their alphabet and the syntax rules.

A *formal language* is a set of '*words*', i.e. finite strings of *letters*, or *symbols*. The 'store' from which these letters are taken is called the *alphabet* over which the language is defined. A formal language is often defined by means of a formal grammar. Formal languages are a purely syntactical notion, so there is not necessarily any meaning associated with them.

The alphabet of a formal language is precisely defined as a finite set and the elements of the set are called symbols of the language.

There is a very interesting example involving a possible formal language to describe the hardness of pencils in the A2 textbook. Its alphabet is {H, B, 2, 3, 4} and the strings are exemplified by {"HB", "2B"….} etc. "HB" is not a symbol, it is a string as shown by the double quotes around it.

Normally, we don't have a finite and very small number of valid strings for a formal language. Instead, we use a notation to express the rules. These are called *metalanguages*. Two *metalanguages* commonly used in Computing are Regular Expressions and Backus-Naur format. Now the COMP3 specification starts to make some sense!

REGULAR LANGUAGE

Back to Finite State Machines (FSM) from the AS course! A regular language can be defined as any language that a FSM will accept. A simple example used in the A2 textbook would be a 'language' composed of the strings generated by the *Regular Expression* **a (a | b)***. In this context, | means OR and * means zero or more instances. E.g. {a, aa, ab, aab, abb, …}.

Equivalent ways of defining a regular language are:

- A regular expression,
- A deterministic (i.e. totally predictable) finite state automaton or
- A non-deterministic (i.e. NOT totally predictable) finite state machine.

In order to find a language that is NOT regular, we have to construct a language that requires an infinite number of states.

REGULAR EXPRESSIONS

One of the most basic and important tasks in computing is the manipulation of text. This could involve word processing documents or parsing Pascal or Delphi programs. Pattern matching is also an important example e.g. trying to find a search string in a document or deciding how to analyse and compile a high level language.

In computing, **regular expressions** provide a concise and flexible means for identifying strings of text of interest, such as particular characters, words, or patterns of characters. Regular expressions (abbreviated as **regex** or **regexp**) are written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.

Another example from the A2 textbook involves a check to see if an email address is syntactically possible. This is relatively easy to do. Regular expressions offer a way to specify such patterns and also to solve the corresponding pattern matching problem.

REGULAR EXPRESSION NOTATION

The notation a (a | b)* given earlier in the previous section is an example of a regular expression or REGEX for short. RGEXES describe a set of strings e.g. the set of all binary strings with an even number of 1's for a 16 BIT binary number (think back to the AS FSM workshop last year!) or the set of valid email addresses (there are lots but still a finite number). In the worst case examples, there may be an infinite number of valid strings so we can't write them all down. Consequently, regular expressions are used to give a concise description of a set without having to list all possible elements.

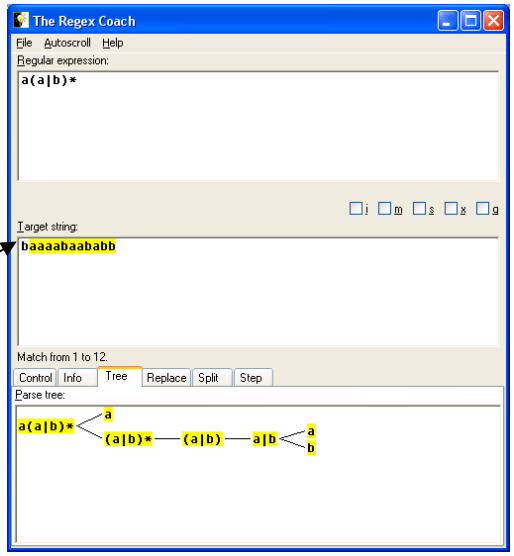
The simple example given above describes the infinite set {a, aa, ab, aab, abb, ….} but note that {b} on its own isn't allowed. We can test this out using the activity below.

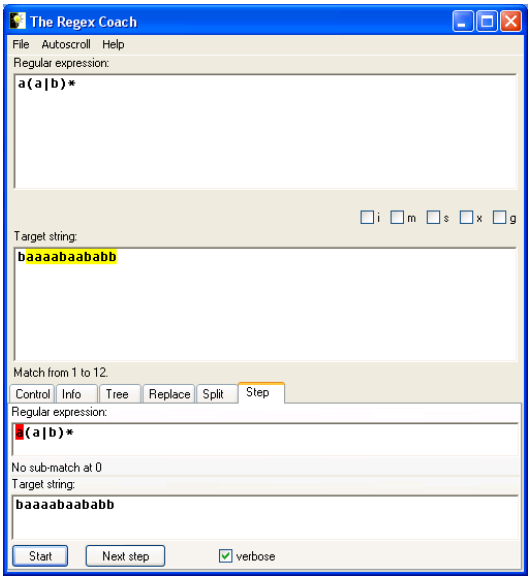
ACTIVITY

First explore the regular expression library at http://regexlib.com/

Next download the regular expression tool called Regex Coach from http://www.weitz.de/regex-coach/ then use it. Start first with the Regular Expression a (a | b)*

Note that the first symbol of the target string isn't highlighted in yellow because it isn't valid.





Stepping through the Target String using the Regular expression is also a very worthwhile activity.
Try these other examples:

abc
abc | bac
(a | c) b (a | c)
a+ Just a regular expression that matches 1 or more a's
a* a regular expression that matches a string with zero or more a's
ab*
(ac)*
a*ca*ca* defines a 'language' containing string with any number of a's but exactly 2 c's

There are several characters with special meaning. So far we have just used the vertical bar or pipe symbol | , the asterisk *, the plus sign + and opening and closing brackets ( ). See the A2 textbook for more information and a number of possible example questions.

APPLICATIONS OF REGULAR EXPRESSIONS

- Used by OS for pattern matching within commands
- Searching for files and folders
- Searching for word or phrase in a block of text i.e. search & replace
- Searching for identifiers in programming languages
- Searching for virus signatures.
- Validating data entry fields e.g Credit card numbers or Postcodes or Car reg info
- The Microsoft .NET Framework, which you can use with any .NET programming language such as C# or Visual Basic.NET, has solid support for regular expressions.

REGEX MAPPING

It is possible to map a Regular Expression to a FSA. Go to the URL:

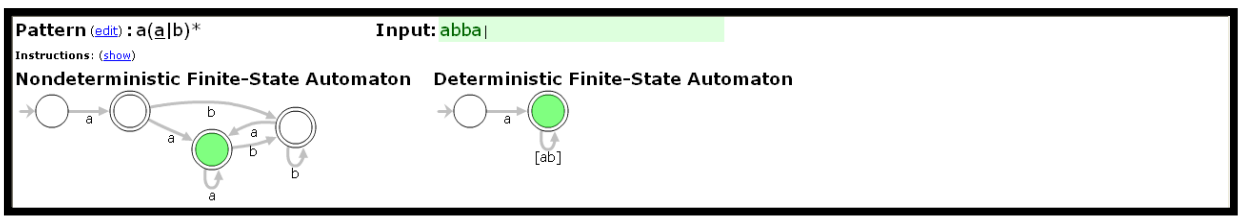
http://osteele.com/tools/reanimator/

This is a Regular Expression FSA Visualiser which maps an input string into a non-deterministic and then a deterministic FSA. As you type in the input string, it animates both FSA's in an amazing way.

Basically, you type some text in the "Input" box on the top left of the screen. The text will be matched against the regular expression in the "pattern" box which you have set. A Blue State indicates a possible match (**depending on the remaining text**); a green state indicates a complete match; and red text in the input string indicates a non-match.

In-built examples to try include : a*b | b*a, [ab] | [bc] | ab, \d+(\.\d*) \.\d+, a | ab | abc | abcd, abcd | bcd | cd | d In addition, why not try a (a | b)* that we used a short time ago?

The supplied regular expression is compiled into a nondeterministic finite-state automaton (the first graph). Most regular expression engines then reduce this to a deterministic finite-state automaton (the second graph). This is a bit like a board game. The input string is interpreted as a series of instructions to advance a "game counter" (the state) along the game board (the automaton). If it lands on a winning space (a final state), there's a match, so the state turns green. The sequence of input characters produces animation.



BACKUS-NAUR FORM

In computer science, **Backus–Naur Form** (**BNF**) is a metasyntax used to express context-free grammars: that is, a formal way to describe formal languages. John Backus and Peter Naur developed a context free grammar to define the syntax of a programming language by using two sets of rules: i.e., lexical rules and syntactic rules. The initially developed it to express the syntax of the programming language Algol in the late '50's / early '60's.

BNF is widely used as a notation for the grammars of computer programming languages, instruction sets and communication protocols, as well as a notation for representing parts of natural language grammars. Many textbooks for programming language theory and/or semantics document the programming language in BNF.

Defining the syntax of a formal language using a regular expression can be very tedious for languages with large alphabets. It would be nearly impossible to do this for even a language like Pascal. Some languages could not have their syntax defined by a regular expression e.g. any language which allows significant nesting of brackets within brackets.

In theoretical computer science, a **formal grammar** (sometimes simply called a **grammar**) is a set of **formation rules** that describe which strings formed from the alphabet of a formal language are syntactically valid within the language. A grammar only addresses the location and manipulation of the strings of the language. It does not describe anything else about a language, such as its semantics (i.e. what the strings mean).

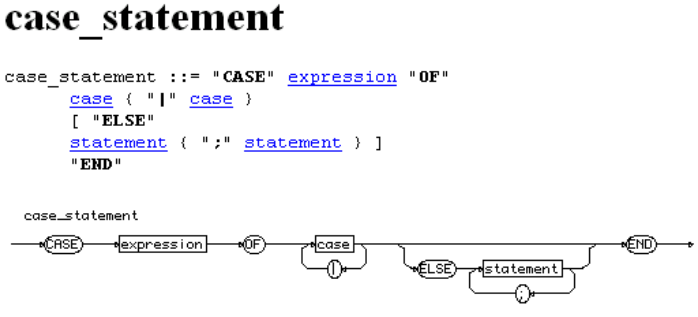
The A2 textbook uses BNF notation and a parse tree to represent the simple "formal language" related to pencil hardness. BNF can also deal with recursive definitions. BNF is used by writers of compilers to express the syntax of the programming language.

SYNTAX DIAGRAMS

Syntax diagrams are an alternative way of defining the syntax rules of a language. They can even be used to describe a subset of the English language. There are some good examples of this in the A2 textbook.

Use the following URL to investigate syntax diagrams for a number of languages. Modula 2 has a syntax based on Pascal and also written by Niklaus Wirth. Several other languages are covered including SQL.

http://cuiwww.unige.ch/db-research/Enseignement/analyseinfo/BNFweb.html



EXTENDED BACKUS-NAUR FORM

The basic BNF had the problem that options and repetitions could not be directly expressed. Instead, they needed the use of an intermediate rule or alternative production defined to be either nothing or the optional production for option, or either the repeated production or itself, recursively, for repetition. However, the same constructs can still be used in EBNF.

First, you need to download the following URL:

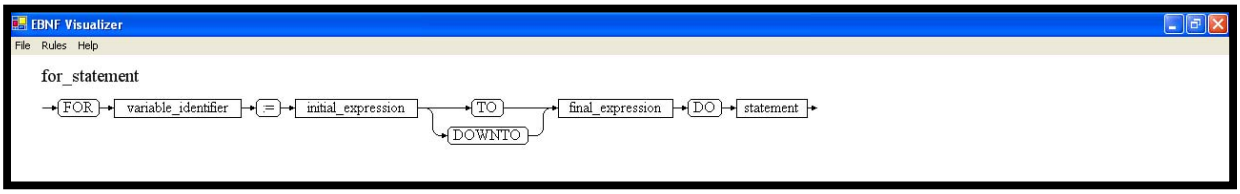
http://dotnet.jku.at/applications/Visualizer/

This program 'visualises' EBNF (Extended Backus Naur Form) grammar rules as syntax diagrams. A text file with the suffix .ebnf is required, which contains grammar rules written in EBNF. The program parses the rules, visualises them in form of syntax diagrams and is able to generate .gif files for further use (e.g. in Word or on Web pages). Furthermore the program allows users to manipulate the look and the layout of the generated syntax diagrams.

It is probably best to download the .zip file Ebnf-Visualizer.zip which is a .zip file with the Visualiser executable, a few sample grammars, the source code and a manual. The grammars for Java, Modula2 and a simple demo EBNF are included but I can probably send you a Pascal version as well. This pascal .ebnf then needs to be copied into the Grammars sub-directory of the application.

To be able to run this program, you will need the .NET framework.

Load and run the application, initially, a blank screen (except for the menu bar) similar to that below should appear.



You then need to select **File**, then **Load Grammar** … on the menu bar then select **pascal.ebnf** from the pull-down list for **Rules**. Then, select for_statement from the Rules option on the menu bar. If you select the for_statement from the list of rules, the application will produce something like the diagram above.

REVERSE POLISH NOTATION

Now we need to learn about **hsilop** (Reverse Polish of course!! exemplified? by 'jatiw'). Polish notation (or just **PN**) was a prefix notation introduced in 1920 by the Polish mathematician Jan Łukasiewicz. It was later further developed as a mathematical notation (RPN) wherein every operator follows all of its operands. This is also known as **Postfix notation** and is usually written parenthesis-free i.e. take out the brackets.

To understand this idea, your students need to revise the mathematical definitions of operators and operands. In the arithmetic expression 3 + 4, we have two operands (the 3 and the 4) and one operator i.e. the +. Essentially, an operator works on operands. The expression 3 + 4 is said to be an infix expression because the operator operates between the two operands.

More complex examples of infix expressions are things like 12 + 8 * 4 or (F – 32) * 5 / 9.

There is a potential problem with infix notation maths. In the first example, do we add the 12 and 8 together and then multiply this answer by 4 or do we first multiply the 8 and 4 together and add this result to the 12. Clearly 80 ≠ 44, a significant potential error. At GCSE level, the **P**lease **E**xcuse **M**y **D**ear **A**unt **S**ally phrase may help us (or use BODMAS or BIDMAS). We carry out multiplication before addition. These so called precedence rules proved difficult to implement in calculator hardware in the 1970's and 80's hence the use of RPN. If we needed to add first, we would use brackets (parentheses to give them their correct name) and could write the expression as (12 + 8) * 4. However, you would have to design the calculator hardware to deal with these brackets. Can you write an expression without using brackets which has one and only one order of evaluation and would be easy for a machine to evaluate? Yes you can and it is called postfix or RPN. Our expression simply becomes 12 8 + 4 *

The Reverse Polish scheme for computers was proposed in 1954 by Burks, Warren, and Wright and was independently reinvented by F. L. Bauer and E. W. Dijkstra in the early 1960s to reduce computer memory access and utilize the stack to evaluate expressions.

The notation and algorithms for this scheme were further developed by Australian philosopher and computer scientist Charles Hamblin in the mid-1950s.

During the 1970s and 1980s, RPN was even used by some members of the general public, as it was widely used in handheld and desktop calculators of the time – for example, the HP-35 series calculators. These cost about $395 at that time. HP completely understimated demand by an order of magnitude and there were long waiting lists for purchasing.

The HP-35 had numerical algorithms that exceeded the precision of most mainframe computers at the time. During development, Dave Cochran, who was in charge of the algorithms, tried to use a Burroughs B5500 to validate the results of the HP-35 but instead found too little precision in the former to continue. IBM mainframes also didn't measure up to the task.

One year later in 1973, Clive Sinclair in the UK produced the *Sinclair Scientific* which provided the basic trigonometric functions and log/antilog. It used Reverse Polish Notation (RPN) just like the HP-35, and gave results in scientific notation. Like many Sinclair electronic equipment it was available even cheaper as a self assembly kit. Lots of soldering! You also needed to be a bit of a mathematician to use it. Still, for us fledgling engineers, it was better than the unwieldy slide rule that we had to use in university finals at that time.

In computer science, postfix notation is often used in stack-based and concatenative programming languages e.g. Forth. It is also common in dataflow and pipeline-based systems, including Unix pipelines.

EXAMPLES OF INFIX AND POSTFIX EXPRESSIONS

For this section of the specification, just stress to students that *INFIX* means 'NORMAL' mathematical and *POSTFIX* means RPN expressions.

The advantages of RPN expressions over INFIX expressions are that brackets are not needed to show the correct order of evaluation. This makes them simpler to evaluate by a machine. Even if you use brackets in a programming language, brackets are stripped out during the compilation process and the expressions converted to RPN for prior to evaluation at run time.

This section concerning infix and postfix could be linked into the section on Tree Data Structures. Stacks are also extensively used in the evaluation of such expressions.

The infix expression 5 + ((1 + 2) * 4) − 3 can be written down like this in RPN: 5 1 2 + 4 * + 3 − The postfix expression is evaluated left-to-right, with the inputs interpreted as shown in the following table (the *Stack* is the list of values the algorithm is "keeping track of" after the *Operation* given in the middle column has taken place):

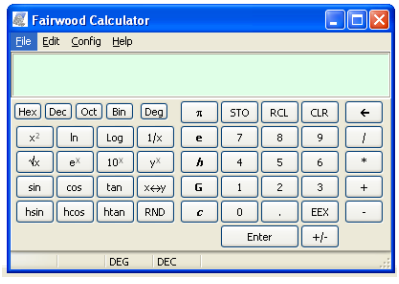
INFIX	POSTFIX
x - y	x y -
(a + b) / (a – b)	a b + a b - /
x + (y ^ 2)	x y 2 ^ +
5 + ((1 + 2) * 4) − 3	5 1 2 + 4 * + 3 −

Input	Operation	Stack	Comment
5	Push operand	5	
1	Push operand	5, 1	
2	Push operand	5, 1, 2	
+	Add	5, 3	Pop two values (1, 2) and push result (3)
4	Push operand	5, 3, 4	
*	Multiply	5, 12	Pop two values (3, 4) and push result (12)
+	Add	17	Pop two values (5, 12) and push result (17)
3	Push operand	17, 3	
–	Subtract	14	Pop two values (17, 3) and push result (14)

When a computation is finished, its result remains as the top (and only) value in the stack; in this case, 14.

HOW CAN I TEACH THIS?

Download a simple RPN calculator as a .exe file e.g http://sourceforge.net/projects/fwcalc/ or find a Java Applet that does broadly the same thing (just do a Google search to find one). Not all of these support exponentiation operations however.



First consider the example of (12 + 8) * 4. They should be able to evaluate this correctly as 80 using the PEMDAS precedence rules. Now run the Firwood Calculator program and enter the expression as 12 <Enter> 8 <Enter> + 4 * and you should get the same result.

Why not get your students to write down a number of expressions in terms of x and y or a and b, including some with bracketed terms, and then get them to write down the answers as postfix expressions? Finally get them to check these by substituting sensible integers for the values and carry out a manual calculation for the infix expression and comparing it with the result of entering the postfix expression in the RPN calculator of choice.

(x + y) * (x – 4) / 2

Weaker students might not feel very confident about abstract values involving x and y. Just get them to substitute some integers in for x and y and then apply the PEMDAS / BODMAS precedence rules which they will know from GCSE Maths.

For example, substitute x = 6 and y = 4 into the above expression. It should give you a value of 10. Next enter the string below into the RPN Calculator and you should get the right answer.

6 <Enter> 4 <Enter> + 6 <Enter> 4 <Enter> - * 2 /

EXTENSION PROGRAMMING EXERCISE

Get your competent programmers to code an Infix to Postfix expression converter. A very long time ago I wrote one in PL/1 to try to get to grips with the subject but I have also used a version written in Turbo Pascal from "Illustrating Pascal by Donald Alcock – Cambridge University Press ISBN-10: 0521336953" which should be easily convertible to Delphi Console Mode.

This has the advantage that it accepts symbolic expressions e.g.

A + (B – C) * D – F / (G + H) transforms to ABC – D * + FGH + / -

I have now found my source code again so I don't have to recode it from scratch. I will gladly send you a copy. I used to find this application very useful when teaching the pre 2000 specification on RPN.

EXAMINATION STYLE QUESTION & ANSWER

Backus Naur Form (BNF) is used by compiler writers to express the syntax of a programming language. The syntax for a part of one such language is written in BNF as follows:

<expression> ::= <integer> | <integer> <operator> <expression>
<integer> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<operator> ::= + | - | * | /

Do the following expressions conform to this grammar?

	Expression	Yes/No
1	4 * 9	
2	8 + 6 / 2	
3	-6 * 2	
4	(4 + 5) * 5	

(4 marks)

(b) (i) Express the infix expression

5 + 6 * 2

in reverse polish notation.

(2 marks)

(ii) Give **one** advantage of reverse polish notation.

(1 mark)

(a)

	Expression	Yes/No
1	4 * 9	Yes;
2	8 + 6 / 2	Yes;
3	-6 * 2	No;
4	(4 + 5) * 5	No;

4 marks

(b) (i) 5 6 2 ;*; +; 2 marks

(ii) No Brackets;
Easy to Compute; 1 mark

BIBLIOGRAPHY

'AQA Computing A2' by Kevin Bond and Sylvia Langfield, published by Nelson Thornes ISBN 978-0-7487-8296-3

'AQA Computing AS' by Kevin Bond and Sylvia Langfield, published by Nelson Thornes ISBN 978-0-7487-8298-7

http://en.wikipedia.org/wiki/Natural_language_processing

http://en.wikipedia.org/wiki/Formal_language

http://en.wikipedia.org/wiki/Regular_expression

http://en.wikipedia.org/wiki/Regular_expression_examples

http://regexlib.com/

http://www.weitz.de/regex-coach/

http://regexpstudio.com/TRegExpr/TRegExpr.html

http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form

http://en.wikipedia.org/wiki/Formal_grammar

http://www.garshol.priv.no/download/text/bnf.html

http://en.wikipedia.org/wiki/Concrete_syntax_tree

http://en.wikipedia.org/wiki/Reverse_Polish_notation

http://sourceforge.net/projects/fwcalc/

http://osteele.com/tools/reanimator/

http://dotnet.jku.at/applications/Visualizer/