VLex: Visualizing a Lexical Analyzer Generator – Tool Demonstration –

Alisdair Jorgensen
DSSE Group, ECS
University of Southampton
Southampton, SO17 1BJ, UK
alisdair@vlex-tool.net

Rob Economopoulos
DSSE Group, ECS
University of Southampton
Southampton, SO17 1BJ, UK
gre@ecs.soton.ac.uk

Bernd Fischer
DSSE Group, ECS
University of Southampton
Southampton, SO17 1BJ, UK
b.fischer@ecs.soton.ac.uk

ABSTRACT

Lexical analyzer generators such as lex and its many successors are based on well-understood concepts. Yet, students often have problems to intuitively grasp and visualize these concepts, especially in compiler engineering courses that emphasize the use of tools over fundamental algorithms. VLex is designed to close the gap left by existing visualization tools, and to help students to understand the approach taken and the algorithms used in lexical analyzer generators. It has the "look and feel" of a lexical analyzer generator, rather than that of a theory animation tool. It can handle multiple lexical states and accepting states can return different tokens. VLex visualizes the algorithms typically implemented in a lexical analyzer generator in the lex tradition, i.e., converting regular expressions via non-deterministic into a deterministic finite automata and then minimizing these automata. The visualization works incrementally, and the user can choose any location to control how the algorithms continue. VLex can also animate the different automata during the scanning phase.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*user interfaces*; D.3.4 [**Programming Languages**]: Processors—*compilers*; H.5.2 [**User Interfaces**]: Graphical User Interfaces

General Terms

Algorithms, Human Factors, Theory

Keywords

lexical analysis, visualization

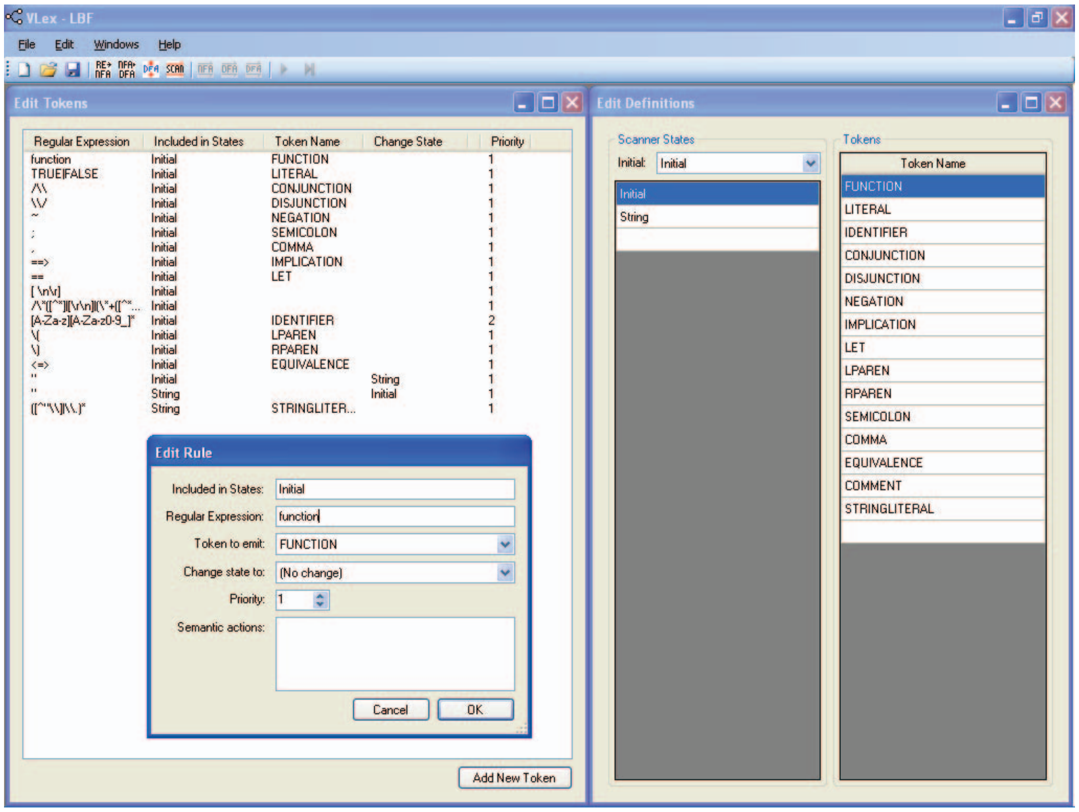
1. INTRODUCTION

Lexical analyzer generators such as `lex` [7] and its many successors are based on the well-understood concepts of regular expressions (RE), non-deterministic (NFA) and deterministic finite automata (DFA). Yet, students often have problems to intuitively grasp and visualize these concepts, especially in compiler engineering courses that emphasize the use of tools over the fundamental algorithms. Hence, they often find it hard to successfully specify the lexical analysis phase of "real" languages. VLex is a new visualization tool that helps students to understand the approach taken and the algorithms used in lexical analyzer generators.

Design Philosophy. VLex is based on the idea of *complete user control*, i.e., the user can at each step determine where the algorithms proceed. For example, the user can choose which regular (sub-) expression to expand during the NFA construction, or which equivalence class to split during the DFA minimization. This requires the presentation to be fine-grained and *incremental*, to allow the presentation of step-by-step results. Moreover, this requires the visualization itself to be inert, to minimize the visible step-by-step changes. This style allows students to easily make and test predictions about individual steps of the algorithms.

Graph Visualization. VLex uses a custom graph drawing algorithm to achieve a compact layout that is reminiscent of that used in many standard compiler textbooks, e.g., [2], and that is suitable for the incremental visualization style.

Related Work. A number tools have been implemented to visualize different aspects of the lexical analysis phase, but none of them retain the "look and feel" of a lexical analyzer generator. jFAST [10] allows the interactive construction and simulation of different types of finite state machines (including NFAs and DFAs) but does not support lexical specifications via REs and does not visualize any of the transformations between the different representations. The FSA Simulator [6] has similar restrictions. Braune et al. [3] describe a learning software that visualizes the different transformations; however, it uses a fixed example and thus amounts to a "canned demo" only. The GaniFA applet [5] lifts this restriction but remains aimed at automata theory rather than at compiler construction. JFLAP [4] is another interactive learning software aimed at automata theory. It visualizes the major algorithms but only partially automates the construction and instead guides the students through the algorithms, warning them about any errors. The HaLeX library [8] provides several Haskell datatypes and functions to represent and manipulate REs and FAs. However, its focus is on a concise formalization of the algorithms, rather than on visualization, and it only provides a built-in `dot` graph output that is not suitable for interactive visualization.



Regular Expression	Included in States	Token Name	Change State	Priority
function	Initial	FUNCTION		1
TRUE|FALSE	Initial	LITERAL		1
/\\	Initial	CONJUNCTION		1
\\/	Initial	DISJUNCTION		1
~	Initial	NEGATION		1
;	Initial	SEMICOLON		1
,	Initial	COMMA		1
==>	Initial	IMPLICATION		1
==	Initial	LET		1
[\n\r]	Initial			1
/*([^*]|[\r\n]|(*+([^*...	Initial			1
[A-Za-z][A-Za-z0-9_]*	Initial	IDENTIFIER		2
\(	Initial	LPAREN		1
\)	Initial	RPAREN		1
<=>	Initial	EQUIVALENCE		1
"	Initial		String	1
"	String		Initial	1
([^"\\]|\\.)*	String	STRINGLITER...		1

Edit Rule

Included in States:	Initial
Regular Expression:	function
Token to emit:	FUNCTION
Change state to:	(No change)
Priority:	1
Semantic actions:	

Scanner States

Initial: Initial

- Initial
- String

Tokens

Token Name
FUNCTION
LITERAL
IDENTIFIER
CONJUNCTION
DISJUNCTION
NEGATION
IMPLICATION
LET
LPAREN
RPAREN
SEMICOLON
COMMA
EQUIVALENCE
COMMENT
STRINGLITERAL

Figure 1: Editing a language within VLex

2. VLEX

VLex is designed to close the gap left by the existing tools. It can handle multiple lexical states and accepting states can return different tokens. VLex visualizes the algorithms typically implemented in a lexical analyzer generator in the `lex` tradition, i.e., converting an RE via an NFA into a DFA and then minimizing this DFA. The visualization works incrementally, and the user can choose any location to continue the algorithms; for example, the user can pick a regular (sub-) expression to drive the RE-to-NFA conversion, or a DFA-state and input character to drive the NFA-to-DFA conversion. VLex can also animate the different automata during the scanning phase.

2.1 Lexical Specification

The lexical syntax of a language is defined in VLex via lexical rules. Each rule associates an RE with a token, a priority, and lexical states. The token is returned when a string is matched against an RE, and the priorities are used to determine which token should be returned if a string can be matched by different REs. Lexical states are used to restrict which REs are active during the scanning process. The lexical state can be changed after the accepting state of an RE has been reached. VLex supports the usual RE operators, including character classes but excluding complementation. Fig. 1 shows VLex's interactive editing window. Specifications can be saved into and imported from an XML file format. VLex can also import lexical specifications in JFlex [1] format, but it ignores any semantic actions associated with the REs. The import thus requires some manual post-processing, in particular the definition of the tokens to be returned.

2.2 NFA Construction

VLex uses and animates the syntax-based McNaughton-Yamada-Thomson algorithm [2] to convert an RE into an NFA that accepts the same language. VLex can construct an NFA for a lexical state, or for a single rule. The visualization starts with an automaton where the transition(s) are labelled with the original regular expression(s). VLex provides two different ways of animating the expansion of states: (*i*) the user can step through the algorithm by double clicking on any transition that is labelled with an unexpanded RE, or (*ii*) they can follow the animation in the sequence determined by the tool. Fig. 2 shows the state of the animation after several REs have been expanded. The expansion process can be completed automatically by clicking on the *run-to-completion* button in the toolbar, which displays the final NFA.

2.3 DFA Construction

VLex uses and animates the standard subset construction algorithm to convert the constructed NFA into an equivalent DFA. The complete NFA is displayed in one window pane while the construction of the DFA is animated in a second pane (see Fig. 3). A transition table is used to display and track the algorithm's progress. Its first two columns show the mapping between a DFA state and the corresponding set of NFA states. The remaining columns display the possible transitions (on mutually exclusive character classes) in terms of both the NFA and DFA states. The first numbers

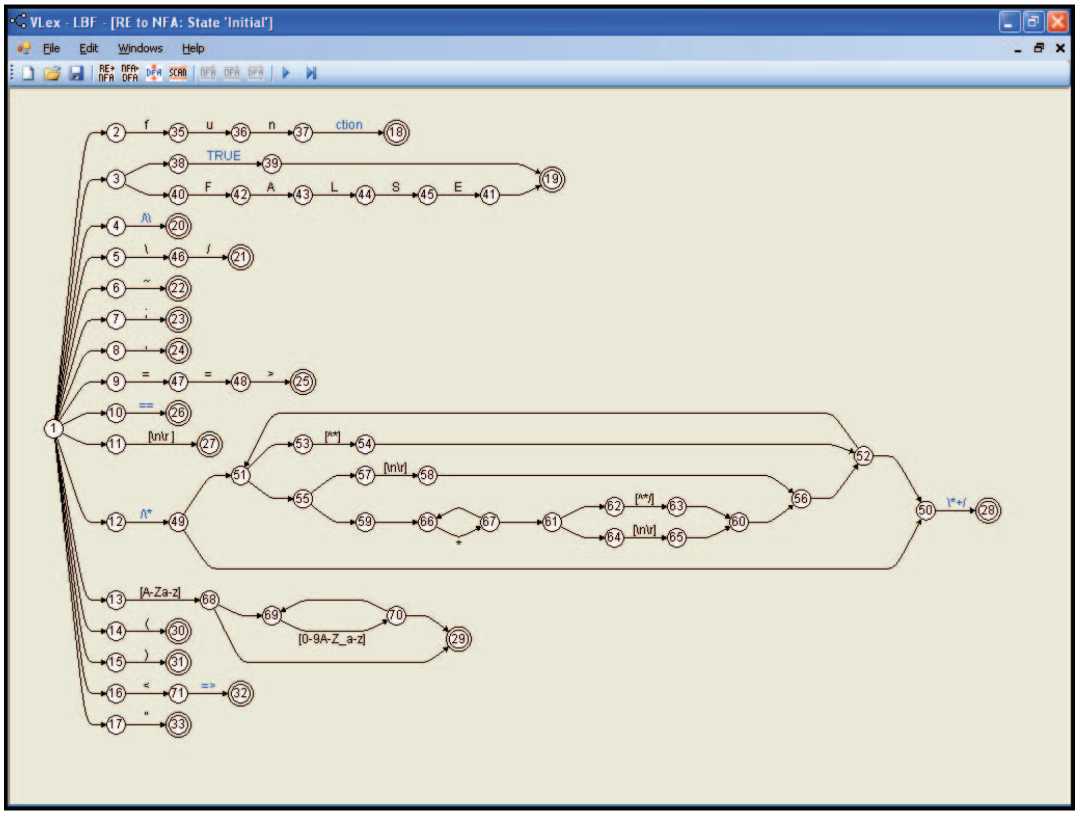
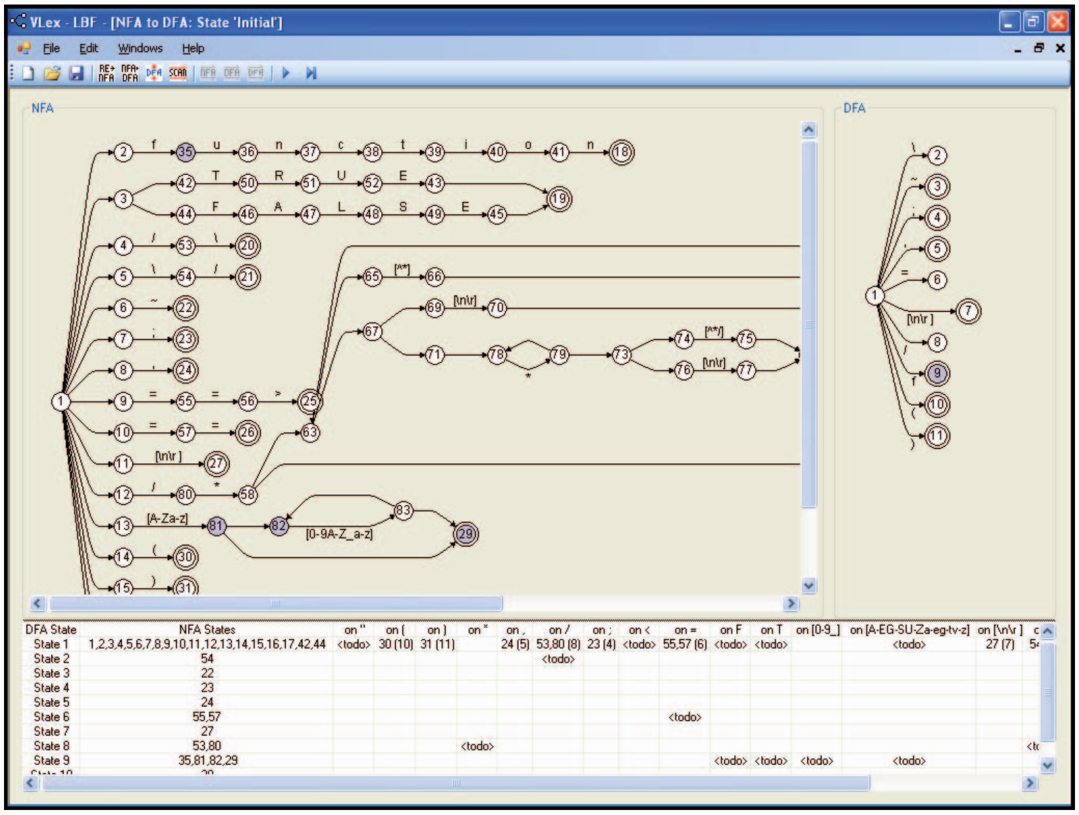
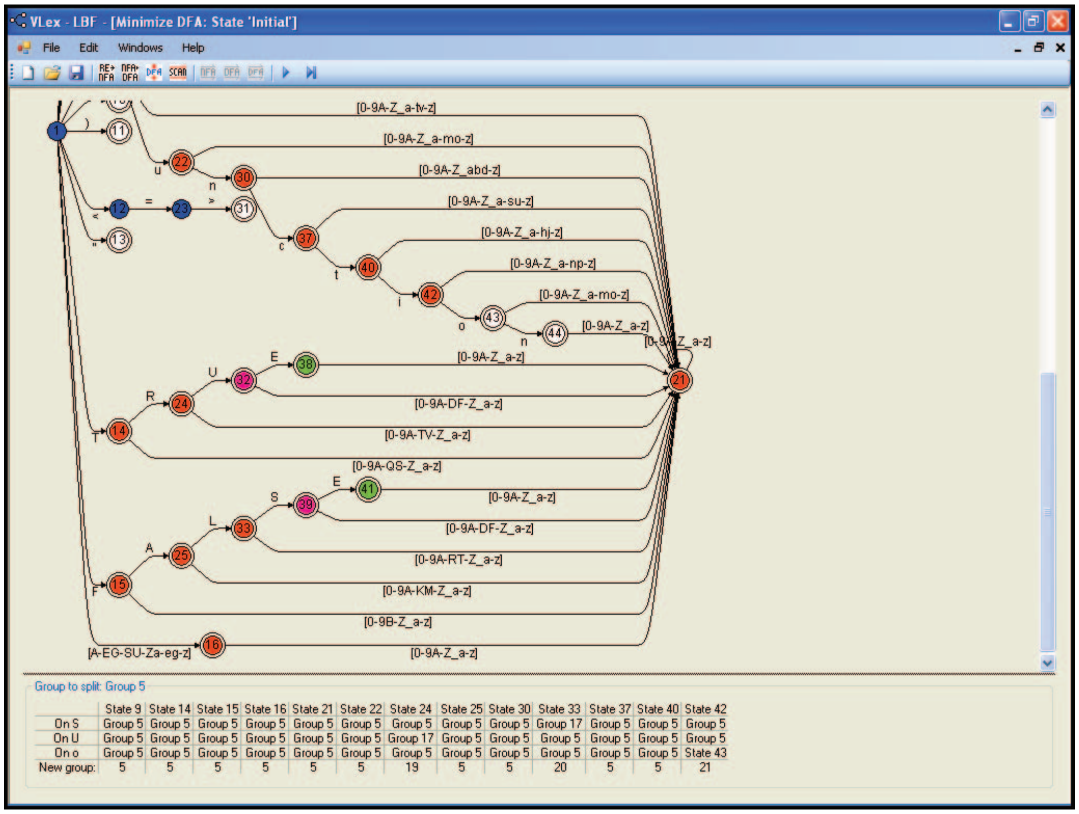


Figure 2: Partially expanded NFA during construction



DFA State	NFA States	on "	on (	on)	on *	on ,	on /	on ;	on <	on =	on F	on T	on [0-9_]	on [A-EG-SU-Za-eg-tv-z]	on [\n\r]	c
State 1	1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,42,44	<todo>	30 (10)	31 (11)		24 (5)	53,80 (8)	23 (4)	<todo>	55,57 (6)	<todo>	<todo>		<todo>	27 (7)	5
State 2	54						<todo>									
State 3	22															
State 4	23															
State 5	24															
State 6	55,57									<todo>						
State 7	27															
State 8	53,80				<todo>											<t
State 9	35,81,82,29										<todo>	<todo>	<todo>	<todo>		

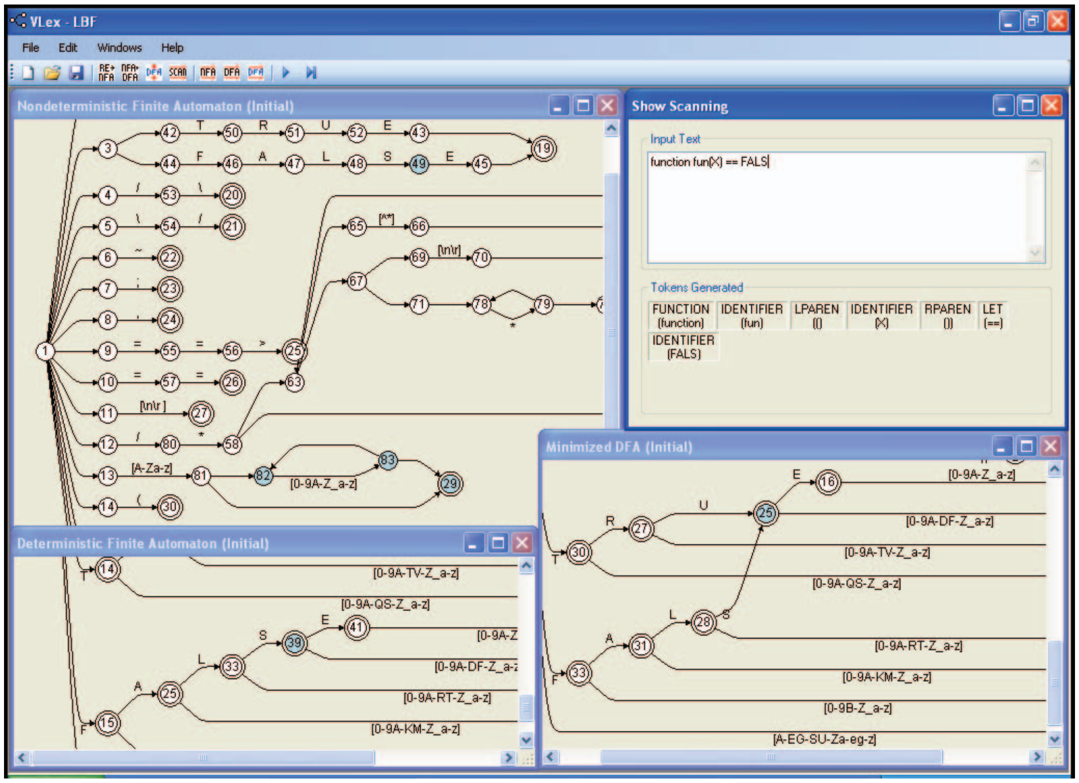
Figure 3: NFA-to-DFA conversion



Group to split: Group 5

	State 9	State 14	State 15	State 16	State 21	State 22	State 24	State 25	State 30	State 33	State 37	State 40	State 42
On S	Group 5	Group 5	Group 5	Group 5	Group 5	Group 5	Group 5	Group 5	Group 5	Group 17	Group 5	Group 5	Group 5
On U	Group 5	Group 5	Group 5	Group 5	Group 5	Group 5	Group 17	Group 5	Group 5	Group 5	Group 5	Group 5	Group 5
On o	Group 5	Group 5	Group 5	Group 5	Group 5	Group 5	Group 5	Group 5	Group 5	Group 5	Group 5	Group 5	State 43
New group:	5	5	5	5	5	5	19	5	5	20	5	5	21

Figure 4: DFA minimization


VLex - LBF
File
Edit
Windows
Help
Nondeterministic Finite Automaton (Initial)
Deterministic Finite Automaton (Initial)
Minimized DFA (Initial)
Show Scanning

Input Text

function fun(X) == FALS

Tokens Generated

FUNCTION (function)	IDENTIFIER (fun)	LPAREN (()	IDENTIFIER (X)	RPAREN ())	LET (==)
IDENTIFIER (FALS)					

Figure 5: Animation of scanning process

represent the epsilon closure of the NFA states that can be reached via a given symbol, or character class, and the final bracketed number is the corresponding DFA state. Clicking on an entry highlights the corresponding NFA and DFA states in the automata. Transitions that have not been expanded yet are marked by a table entry with a `<todo>` value.

The user can step through the algorithm by double clicking on any `<todo>` entry. VLex then adds, if necessary, the new DFA state, labels the transition with the character class that enables it, and replaces all `<todo>` entries from the selected source state that transition into the same new target state. If the newly created state transitions differently on different input characters, VLex also splits the character classes and adds more columns to the transition table. Finally, VLex updates the display of the DFA, minimizing the part of the automaton that is redrawn. Note that different expansion order can thus lead to different DFA layouts. The expansion process can again be completed automatically by clicking on the *run-to-completion* button in the toolbar, which displays the final DFA.

2.4 DFA Minimization

VLex animates the standard DFA state minimization algorithm described in [2, p181], which maintains and iteratively refines an equivalence relation on the DFA states (see Fig. 4). The main problem here is to display these equivalence classes comprehensively. VLex uses color: states in the same equivalence class are filled in the same color; to minimize the number of colors required, classes comprising a single state remain white. An incremental color picking algorithm based on recursively dividing the available hue range by the Golden Ratio improves the contrast, without requiring to know the maximal number of non-degenerate classes up-front. Although this approach will eventually lead to similar colors being used for different classes, we have found that this only occurs for very large DFAs, and overall, the coloring scheme has proven to be an effective way of representing the equivalence classes. VLex can also display information about each state which is essential when trying to understand the minimization algorithm. By clicking on a state node, a table is displayed with all states in the class and the classes that can be reached on transitions from each state. The user can step through the algorithm by double-clicking on any colored state node. If two or more states transition on the same character into different equivalence classes, these states become distinguishable, and the equivalence class is split. VLex then updates the state coloring (picking a new color if required) and the transition table. The minimization process can again be completed automatically by clicking on the *run-to-completion* button in the toolbar.

2.5 Scanning Process

Fig. 5 shows how VLex animates the scanning of a code fragment. As characters are input and recognized the corresponding tokens get generated and are output. If any of the three automata (NFA, DFA and minimized DFA) are displayed then the transitions through them are animated by highlighting the state reached during the recognition of the current token. If a character is input that is not recognized, then an ERROR token is output and the start state of each automaton is highlighted. The error can then be corrected by deleting characters from the input.

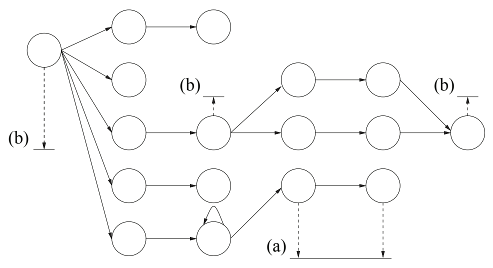


Figure 6: Vertical adjustment of nodes

3. IMPLEMENTATION AND AVAILABILITY

VLex is implemented in C# and uses MS Visual Studio 2008. It is freely available from `www.vlex-tool.net`. Its central data structure is the automaton, which is shared between all three types (i.e., NFA, DFA, and minimized DFA). Each of the major algorithms (NFA and DFA construction as well as DFA minimization) is implemented as a separate module. The implementations themselves are more complicated than the textbook versions, or the HaLeX variants, because they need to produce incremental visualizations. They are thus not exposed to the students.

VLex uses a customized layout algorithm so that the automata are drawn in a similar style to that used in textbooks. The states are traversed breadth-first and the state nodes are placed on a grid, starting at the top-left corner, with minimal vertical spacing and the horizontal spacing determined by the breadth-first order and length of the transition labels. If a state is placed to the right of the target of one of its outgoing transitions, then the target state (and all its successors) are shifted to the right, unless the transition represents a backwards edge in the graph. This way, all forward edges in the graph are laid out in left-to-right direction. The state nodes are then vertically adjusted to keep a compact layout. Fig. 6 shows this process; here, the nodes are moved to their new positions as indicated by the dashed arrows and level lines. The vertical adjustment (a) aligns transitions and states that can form a simple straight line, and (b) moves each node with multiple outgoing transitions towards the centre of its neighbors, similar to Sugiyama's algorithm [9]. The transitions are drawn as arcs after the adjustment. They are composed from Bezier curves and straight lines to achieve the usual appearance. Transition labels are placed horizontally at the mid-point of the transition, and vertically on the opposite side from the average vertical position of the two end points. Finally, if the gradient of the transition is very steep on one side, the label is shifted slightly in the opposite direction. This reduces the problem of labels being drawn on top of neighbouring transitions in cases where a state has a large number of transitions in or out. The algorithm is "incrementally stable" and minimizes redrawing, e.g., after expanding an RE.

4. CONCLUSIONS

VLex is an incrementally visualizing lexical analyzer generator that allows user to determine at which locations (and thus with which steps) the underlying generator algorithms proceed. Early feedback on VLex was encouraging, and we are now planning a class-room evaluation. A variety of extensions are possible, although not necessarily straightforward, due to the incremental nature of the tool. The layout algorithm could be extended by a graph planarization and by manual control over node placement, and the transitions could be animated, as in jFAST. We also plan to add arbitrary user-defined semantic actions as in JFlex; the automata animation could then be integrated with a traditional debugger to allow students to step through the actions as well.

5. REFERENCES

[1] JFlex homepage. `http://jflex.de`.

[2] A. A. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, second edition, 2007.

[3] B. Braune, S. Diehl, A. Kerren, and R. Wilhelm. Animation of the generation and computation of finite automata for learning software. In *Proc. 4th Intl. Workshop Implementing Automata*, LNCS 2214, pp. 39–47. Springer, 2001.

[4] R. Cavalcante, T. Finley, and S. H. Rodger. A visual and interactive automata theory course with JFLAP 4.0. In *Proc. SIGCSE'04*, pp. 140–144. ACM Press, 2004.

[5] S. Diehl, A. Kerren, and T. Weller. Visual exploration of generation algorithms for finite automata on the web. In *Proc. 5th Intl. Conf. Implementation and Application of Automata*, LNCS 2088, pp. 327–328. Springer, 2000.

[6] M. T. Grinder. Animating automata: A cross-platform program for teaching finite automata. In *Proc. SIGCSE'02*, pp. 63–67. ACM Press, 2002.

[7] M. E. Lesk and E. Schmidt. Lex – a lexical analyzer generator. Bell Labs, 1975.

[8] J. Saraiva. HaLeX: A Haskell Library to Model, Manipulate and Animate Regular Languages. In *Proc. ACM Workshop on Functional and Declarative Programming in Education*, pp. 133–140. University of Kiel Technical Report 0210, 2002.

[9] K. Sugiyama, S. Tagawa, S. and M. Toda. Methods for Visual Understanding of Hierarchical System Structures. IEEE Transactions on Systems, Man and Cybernetics, SMC-11(2), pp. 109-125, 1981.

[10] T. M. White and T. P. Way. jFAST: A Java finite automata simulator. In *Proc. SIGCSE'06*, pp. 384–388. ACM Press, 2006.