A Compilation Process Visualization, Interaction System and Compiler Construction Assistant for Efficient Teaching and Learning

OSSAMA EDBALI, The University of Birmingham, United Kingdom

Compiler construction is a well-researched field, and every computer science student should have at least a good overview on the topic. The traditional method in teaching compiler construction is that of following a textbook and for each section incrementally build the compiler. However many aspects and underlying principles of compilation such as lexing, parsing, abstract machines, automata, garbage collection and many more may need an additional learning method which bridges the gap between theoretical knowledge and practical implementation.

The ongoing project outlined in this article, *CC Tutor*, consists of an extensible platform for teaching and learning compiler construction using a subset of Java. The main feature of the platform is for a program to be visualized and interacted with at various stages of the compilation process. However, the platform is suitable for both students in compiler construction and models of computation modules. This aspect makes the platform tailorable to the teacher and student needs without them being required to follow a specific workflow to visualize and interact with the algorithms. A secondary goal of this project is the development of a mathematical model of algorithm visualizations and interactions in order to evaluate the efficiency of the visualizations.

CCS Concepts: • **Applied computing** → **E-learning**; • **Software and its engineering** → **Compilers**; *Garbage collection*; • **Theory of computation** → *Lambda calculus*; *Abstract machines*; *Formal languages and automata theory*;

1 INTRODUCTION

This article details a novel platform for teaching and learning compiler construction. Many topics in Computer Science have been provided with visualization toolkits, e-learning platforms such as Codecademy. However such comprehensive tools are scarse and not complete in regards to compilers and their underlying principles.

Many compiler construction modules cover the process with a reference book such as Modern Compiler Implementation in Java [Appel and Palsberg 2009] and structure the module segments with the various phases. For instance, the MIT OpenCourseWare Computer Language Engineering course follows Appel and Palsberg [2009] book with detailed attention to code generation optimizations. The approach used in this course is to cover each phase theoretically then an implementation of the phase is carried out in groups. Moreover this course focuses on later stages of compilation leaving the lexical and syntax analysis underlying principles to the lexer and parser generators respectively (ANTLR). Thus, the format of most courses is a hands-on approach rather than an in-depth understanding of each phase and some lateral aspects, such as error detection, which usually are not taught in these courses.

Section 2 of this article will describe the existing problems in teaching and learning compiler construction and what CC Tutor proposes to solve them. It also proposes a novel representation of an algorithm visualization and interaction by means of graph rewriting. Section 3 outlines the the software requirements of the platform. Section 4 outlines the implementation details of the platform and Section 5 gives a roadmap for the software evaluation before, during and after development. Section 6 outlines and compares related work to the focus of the project.

Author's address: Ossama Edbali, The University of Birmingham, Birmingham, United Kingdom, oxe410@student.bham.ac.uk.

2 PROBLEM ANALYSIS AND PROPOSED SOLUTION

2.1 Motivation

The lack of educational tools/toolkits for compiler construction prompted the development of CC Tutor, a platform for teacher and students to visualize, inspect and interact with the algorithms and principles governing the compilation process. CC Tutor is a comprehensive tool which includes in-depth visualization of single algorithms, comparison of algorithms (such as LL and LR parsing) walkthrough of all frontend phases of the compilation process and a compiler frontend implementation assistant.

The phases covered are from the Modern Compiler Implementation in Java book [Appel and Palsberg 2009]:

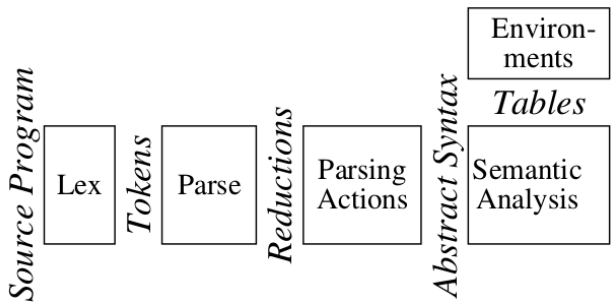


Fig. 1. Phases of a compiler frontend, and interfaces between them.

A variety of compilers courses cover the theory and for practical projects use tools such as ANTLR to generate lexers and parsers. However being able to construct a compiler frontend from scratch is useful to grasp the principles in each phase.

Thus, CC Tutor is implemented in order to fill the gap in educational tools related to compilers and formal languages. The platform is extensible in the sense that it enables software developers and teachers to customize the visualizations of the algorithms to suit their needs.

2.2 Topics Overview

The topics included in CC Tutor are the following:

2.2.1 Lexical Analysis. A lexical analyzer will take as its input a string of individual letters and divide this string into tokens. Additionally, it will filter out whatever separates the tokens (the so-called white-space), i.e., layout characters (spaces, newlines etc…) and comments. This is separate from syntax analysis to allow for efficiency, modularity and easier error recovery [Mogensen 2010]. This phase is vital and sometimes is skipped in compiler modules; it emphasizes the usage of concepts from formal languages and models of computation.

2.2.2 Syntax Analysis. The syntax analysis phase uses the tokens from the lexical analysis phase and forms a syntax tree which reflects the syntax of the input text according to a grammar. This part is central in any compiler construction module and the focus of CC Tutor will also be towards the transition from nondeterministic to deterministic parsing.

2.2.3 Semantic Analysis. The semantic analysis phase of a compiler connects variable definitions to their uses, checks that each expression has a correct type, and translates the abstract syntax into a simpler representation suitable for generating machine code [Appel and Palsberg 2009].

This is the last phase in a compiler frontend and includes concepts such as environments and type checking which are important from an implementation point of view (rather than visualization). CC Tutor incorporates this into its Compiler Construction Assistant (see Section 3.7.10).

2.2.4 Garbage Collection. The memory occupied by garbage (non-reachable heap-allocated records) should be reclaimed for use in allocating new records. This process is called garbage collection, and is performed not by the compiler but by the runtime system [Appel and Palsberg 2009]. This phase has been included in CC Tutor because it is an important concept in object-oriented programming and although a simple concept to grasp when visualized it is usually included in the advanced topics section of compiler textbooks and therefore not covered in traditional compiler construction modules.

2.2.5 Compiler Optimizations. An optimization, generally, is about recognizing instructions that form a specific pattern that can be replaced by a smaller or faster pattern of new instructions [Mogensen 2010]. This phase has been included in CC Tutor since it covers advanced aspects such as data-flow analysis which is used to find contextual requirements of patterns of instructions.

2.2.6 CEK Machine. The CEK machine implements a small step semantics for the lambda calculus by starting with an initial state and taking incremental "steps" to evolve the computation. In essence the CEK machine is an interpreter; it is included in CC Tutor as a machine to implement using the Compiler Construction Assistant (see Section 3.7.10). It includes essential concepts such as lambda calculus, environments and closures. A visualization and interaction with the steps of this process therefore is a first step in building a strong understanding of this abstract machine.

2.3 Proposed Solution

The challenge in developing this platform lies in conveying all the complexity of the compilation process into a two-dimensional time-based setting without sacrificing the details in the name of simple visualizations. Efficient visualizations map data onto visual elements in a way that will help people understand the principles in the algorithms and reason about them effectively. A simple mapping from text/concepts to visual elements matching them goes against the goal of this platform; a too abstract representation of the concepts will remove the efficient learning factor.

Therefore, the main question when developing visualization tools is: what are the visual representations of the data structures and algorithms that will best reveal their semantics? In order to answer this question a visualization and interaction framework has been defined in section 3.7.11. For the latter to be implemented in a way that the user can interpret what they see in an efficient manner, an *algorithm visualization and interaction semantic descriptor* has been defined (see Section below). Algorithm interpretation depends at least on these two main factors: (1) **background knowledge** in the user and (2) **semantic cues** in the graphics. Interpretation is a fluctuation between these two elements: what we have in our head influences what we see in the graphics (this is a very well known fact in vision science) and what we see in the graphics influences what we think.

Background Knowledge. When we look at a visualization, perhaps 95% of what we consciously perceive is not what is "out there" but what is already in our heads in long-term memory [Ware 2011]. Algorithm visualization relies to a certain extent on the mental models the user have. In fact, in this scenario visualization and interaction are used both as a communication and an exploratory tool which requires the user to have a predefined mental model on the topics visualized.

Semantic Cues. The way visualization itself is modeled can support or hinder the semantic association between graphical elements and concepts. The minimum requirement is that the user understands how the graphics works and what it represents.

In order to analyze and compare visualizations an algorithm visualization and interaction semantics descriptor has been developed.

2.3.1 A Graph Rewriting Approach to Modeling Algorithm Visualizations and Interactions. In the literature regarding modeling graphical user interfaces the focus has been towards ontologies and descriptive languages such as XML. A remarkable difference between software engineering models and ontologies is that the former are most often prescriptive models, which are used to specify how a system is supposed to behave, while ontologies are rather descriptive models, which describe how the world is [Heiko and Florian 2011].

In this article an approach that combines the behavior and the state of a user interface (an algorithm visualization and interaction specifically) is defined. An algorithm visualization and interaction semantics descriptor can be modeled as a graph rewriting system. This captures the essence of the algorithm visualizations and interactions in an abstract and general way.

Graph rewriting, concerns the techniques and formalizations of creating a new graph out of an original graph algorithmically. These transformations can be used as a computation abstraction. In fact, as Baresi and Heckel [2002] note, graphs provide a universally adopted data structure, as well as a model for the topology of object-oriented and component-based systems; this makes them suitable for visual-dynamic scenarios.

The basic idea is that an algorithm visualization and interaction can be represented as a graph and changes (either automatic or through user interaction) can be represented as transformation rules on that graph.

A graph rewriting system or graph grammar is a tuple $(N_E, N_V, T_E, T_V, P, G_S)$ where

- $\Sigma_E = N_E \cup T_E$ represents the non-terminals and terminals for edge labels.
- $\Sigma_V = N_V \cup T_V$ represents the non-terminals and terminals for vertex labels.
- G_S is the starting graph (i.e. starting configuration of a visualization).
- P is a finite set of production rules.

A labeled graph $G = (G_E, G_V, s^G, t^G, a^G, n^G$ consists of sets G_E and G_V, called, respectively, sets of edges and vertices, mappings $s^G \colon G_E \to G_V$, $t^G \colon G_E \to G_V$, called, respectively, source and target maps, as well as a pair of additional mappings $a^G \colon G_E \to \Sigma_E$, $n^G \colon G_V \to \Sigma_V$, called, respectively, edge and vertex labeling map.

In an algorithm visualization vertices are the visual elements, control elements (e.g. buttons), the edges are physical connections, vertex labels define properties of the visual elements (e.g. highlighted) and edge labels define the type of physical relationship between the visual/control elements (e.g. node, popup relationship).

Production rules of a graph grammar are in the form:

$$P = (G_L, G_R, H)$$

where

- G_L is a labeled graph corresponding to the left-hand side of P
- G_R is a labeled graph corresponding to the right-hand side of P
- H is a labeled graph with label preserving isomorphisms

The production rule searches for a copy of G_L inside a given graph G and "glues in" a copy of G_R by identifying them along the common subgraph H.

This encoding of visualization into graph rewriting enables us to abstract from the details of the visual elements and to produce a specification for an interactive system. Moreover, through graphs we can analyze the complexity of a visualization by producing various metrics such as alpha index, beta index which measure the connectivity of the graph.

2.3.2 Implementation. In conclusion, in order to relate steps in the algorithms to visualizations and interactions an intermediate inspection layer will be implemented. This resembles the Flux architecture where the algorithm steps are the data, the inspection layer emits this data to the store and the visualization and interaction framework in the controller-view. The platform architecture and the motivation of using the Flux architecture are specified in detail in section 3.7.11.

3 SOFTWARE SPECIFICATION

3.1 Introduction

CC Tutor primary objective is the development of a comprehensive educational platform on compiler principles visualization and compiler construction, therefore a software requirements specification is needed. The IEEE SRS 830 template has been used.

3.2 Naming Conventions

The following naming conventions are used throughout this specification:

Regex	Regular expression
NFA	Nondeterministic Finite Automaton
DFA	Deterministic Finite Automaton
CEK	Control, Environment, Continuation machine [Thielecke 2015]
User	A user is either a student or a teacher

3.3 Purpose and System Scope

The purpose of the CC Tutor platform is to enable students to learn efficiently about the compilation process and its underlying algorithms and principles. It is also a support for the teacher to transfer the knowledge in textbooks and lecture notes in an active environment which has been shown to motivate students to learn, engage and consequently being able to transfer the knowledge themselves [Benware and Deci 1984].

The system tackles the gap in educational tools in computer science (especially related to compilers) by providing a distance learning platform that is focused towards visualizing and interacting with the compilation process algorithms. Both students of compiler construction and models of computation modules can benefit from this tool in their learning process.

3.4 System Perspective

The system is comprised of:

- A platform for students.
- A platform for teachers.
- A generic framework to inspect/debug the compilation process.
- An extensible visualization framework.

3.5 System Functions

In this section an overview of the main features of CC Tutor is defined.

3.5.1 Core Features. The core/general features are common for both the student's and teacher's portal and form the basis of the platform. They include:

- Lexical analysis walkthrough.
- Syntax analysis walkthrough.
- Semantic analysis walkthrough.
- Garbage collection algorithms walkthrough.
- CEK machine walkthrough.

- Compiler optimizations simulations.
- Compiler Construction Assistant.
- Individual algorithm execution.
- Save execution of algorithms (snapshots).
- Save grammars and lexer input.
- Export algorithm visualizations to PDF.

In each compiler phase or topic there are various algorithms that will be run sequentially. However the user might want to inspect only a specific algorithm (e.g. LL(1)). In CC Tutor this is possible by selecting from the list of individual algorithms (see Appendix A for prototypes).

3.5.2 Student Portal. In addition to the core features, the student portal includes the following functionality:

- Complete quizzes assigned by the teacher.
- Complete assignments assigned by the teacher.
- Complete targeted exercises to test the understanding of edge cases in each algorithm.
- Build a compiler frontend using the Compiler Construction Assistant.

3.5.3 Teacher Portal. The teacher portal includes all core features with addition of the following:

- Create general and targeted quizzes.
- Create assignments which can have the following formats:
  - Provide the steps of an algorithm and check the correctness (e.g. LL parsing steps).
  - During a visualization the teacher provides checkpoints where the student should provide an answer (i.e. matching the next step).
  - Provide some stubs to fill in order to execute an algorithm. The teacher can also define test cases or choose from predefined ones.
- Create custom visualizations using the cc-tutor-viz API.
- Create custom test cases for checking the frontend implementation of the students in the Compiler Construction Assistant.

3.5.4 Inspection Framework. The inspection framework is the intermediate layer between the execution of a program and its visualization. It works like a debugging tool where execution units are defined in terms of sections of code. Details of the internal workings of this framework are described in subsection 3.7.11.

3.5.5 Visualization Framework. To visualize the various algorithms an extensible visualization package is to be developed. This is the package that communicates to the underlying algorithms through the inspection framework. It is an extensible package which can be used by developers and teachers to produce their own algorithm visualizations. The visualization framework alongside the inspection framework are described in subsection 3.7.11.

3.6 Documentation

The documentation of the software consists of user tutorials for the student and teacher platforms, a code documentation, an API documentation for the extensibility of the visualization framework. A comprehensive abstract architecture of the software and the design of the database (including ER diagrams and logical mapping) are also provided through the project's documentation repository https://github.com/UoBCS/cc-tutor-docs.

3.7 System Features

In this section the core, relevant system features that form the core of CC Tutor are described in detail. Each system feature starts with a brief description, then a priority relative to the overall software is given. Stimulus/response sequences are the user actions and system responses that stimulate the behavior defined for a specific feature. Finally a list of functional requirements associated with each feature is specified. Each functional requirement is identified by a unique tag which will be used in the development phase to track progress.

For each algorithm feature see subsection 2.2 for an overview.

For each feature a low-fidelity prototype is defined in Appendix A.

The full software specification (with secondary features in terms of visualization and interaction) can be found in https://github.com/UoBCS/cc-tutor-docs.

3.7.1 Regular Expressions to Nondeterministic Finite Automata (NFA).

Description and Priority. This feature allows the user to input several regular expressions (if using the lexical analysis walkthrough) and for each one of them visualize the conversion to NFA. In the overall software perspective this has high priority since it will benefit both compiler construction and models of computation students.

Stimulus/Response Sequences.

- User selects the "Regex to NFA" option from the set of algorithms or selects Lexical Analysis Walkthrough.
- User inputs the regular expression(s) and lexemes associated with them to define tokens (if in the lexical analysis mode).
- System checks for errors in the regular expressions syntax and reports to the user.
- System visualizes the conversion process through the regular expression syntax tree first and then the transformation to NFA.
- System advances to the next regular expression, if any.

Functional Requirements.

LA-REGEX-NFA-1	Submit multiple regular expressions and associated lexemes if necessary.
LA-REGEX-NFA-2	Check for syntax errors.
LA-REGEX-NFA-3	Build and visualize the regex syntax tree to unveil the structure of the regex necessary for the next steps.
LA-REGEX-NFA-4	Enable the user to create nodes in the syntax tree for the next step and check for correctness, while the system is building the tree.
LA-REGEX-NFA-5	Enable the user to create the entire syntax tree and check for correctness.
LA-REGEX-NFA-6	Visualize step by step the creation of NFA from the regex syntax tree by highlighting the nodes being processed.
LA-REGEX-NFA-7	Enable the user to create states and transitions in the NFA corresponding to the next step and check for correctness.
LA-REGEX-NFA-8	Enable the user to create the entire NFA and check for correctness.
LA-REGEX-NFA-9	Enable the user to check the history of the algorithm execution and rewind at any point.
LA-REGEX-NFA-10	Enable the user to save the history for future runs.

3.7.2 *Nondeterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA).*

Description and Priority. This feature allows the user to input a NFA (or as a result from regex to NFA in lexical analysis) and convert it to a DFA through the subset construction algorithm. In the overall software perspective this has high priority since it is one of the core algorithms taught in models of computation courses as well as being the basis for the tokenization process in lexical analysis.

Stimulus/Response Sequences.

- User inputs the NFA either manually or via previous steps (i.e. regex to NFA).
- System checks if the automaton is nondeterministic and highlights where the nondeterminism lies.
- System visualizes the conversion process.

Functional Requirements.

LA-NFA-DFA-1	Dynamically create NFA through user input (file upload or through user interface).
LA-NFA-DFA-2	Check if automaton is nondeterministic and highlight where the nondeterminism lies.
LA-NFA-DFA-3	In each step visualize how the *epsilon closure* and the *move* functions are executed by highlighting the involved states in the NFA and currently generated DFA.
LA-NFA-DFA-4	Enable the user to enter the epsilon closure and/or the move function output for the current DFA state.
LA-NFA-DFA-5	Enable the user to enter a whole DFA state by selecting which NFA states correspond to it.
LA-NFA-DFA-6	Check correctness of the user interactions and inform user.
LA-NFA-DFA-7	Enable the user to check the history of the algorithm execution and rewind at any point.
LA-NFA-DFA-8	Enable the user to save the history for future runs.

3.7.3 *Deterministic Finite Automata (DFA) Minimization.*

Description and Priority. This features enables the user to input a DFA (or as a result from NFA to DFA in lexical analysis) and convert it to a minimal DFA by removing unreachable states and collapsing equivalent states. In the overall software perspective this has medium priority since it is not required by lexical analysis but is a fundamental algorithm in models of computation modules.

Stimulus/Response Sequences.

- User inputs DFA either manually or via previous steps (i.e. NFA to DFA).
- System checks if the automaton is deterministic.
- System visualizes first the elimination of unreachable states, then the collapsing of equivalent states.

Functional Requirements.

DFA-MIN-1 Dynamically create DFA through user input (file upload or through user interface).
DFA-MIN-2 Check if automaton is deterministic.
DFA-MIN-3 Perform depth-first search on the transition graph to remove unreachable states. Visualize the nodes involved.
DFA-MIN-4 Enable the user to input the answer for the next state in the visualization and check for correctness.
DFA-MIN-5 Enable the user to input the entire minimized automaton and check for correctness. The system will highlight the places where the submitted and the actual automata differ.
DFA-MIN-6 Dynamically construct the DISTINCT table which will contain entries if a pair of states is distinct. This is visualized by highlighting each state and transition involved.
DFA-MIN-7 Visualize the merging process by highlighting the empty cells (i.e. not distinct states) and the corresponding states in the automaton.

3.7.4 Tokenization.

Description and Priority. The tokenization process navigates the generated deterministic automaton and advances the input stream pointer. In the overall software perspective this feature has medium priority since it is beneficial only for compiler construction modules students.

Stimulus/Reponse Sequences.

- User provides a DFA representing the acceptance of regular expressions. The DFA can also be generated from the NFA to DFA algorithm in the lexical analysis process.
- System checks the validity of the DFA.
- System visualizes how the tokenization process produces the next token.

Functional Requirements.

TOK-1 Dynamically create DFA through user input (file upload or through user interface).
TOK-2 Perform a visualized depth-first search and store final states in a stack in order to extract the last one (longest match).
TOK-3 Show the priority of each token type and based on that select the state (recall that a DFA state is a set of NFA states) that will be used for the next token.

3.7.5 LL Walkthrough.

Description and Priority. The LL walkthrough will help visualize how a top down parser works. In this setting the algorithm is non-deterministic, therefore no FIRST or FOLLOW sets will be computed to eliminate this non-determinism. Thus this feature relies on the user to input the next step (match characters or predict non terminal). This setting helps the student understand the transition from nondeterminism to the determinism achieved by LL(1). The priority of this feature is set to high since many tools and modules on compiler construction do not cover this part in detail, they instead delve into parsing tables and how they can be computed leaving out an important abstraction.

Stimulus/Response Sequences.

- User provides a grammar or uses the MiniJava grammar as part of the compiler front-end walkthrough.
- User provides the text which needs to be parsed. If using the front-end walkthrough the tokens will be passed from the lexical analysis phase to the parsing phase automatically.

- System builds the stack machine and processes the token stream.
- For each step the user selects the next action to take
- System responds appropriately if the move was valid or produces a stuck state.

Functional Requirements.

SA-LL-1 Create grammar from user input (upload or text).
SA-LL-2 MiniJava editor, standard editor or file upload for input text.
SA-LL-3 Visualize stack, grammar and input stream. At each step highlight all involved entities.
SA-LL-4 Let user input the next step in the process (*match* or *predict*).
SA-LL-5 Save derivation history for later inspection.
SA-LL-6 Save environment (grammar and user input).

Useful visualizations/geometric intuitions for LL are included in the platform: [Thielecke 2016]:

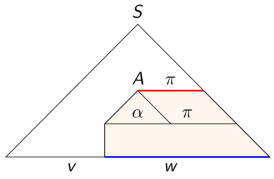


Fig. 2. LL predict step 1

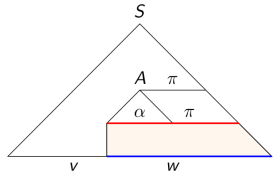


Fig. 3. LL predict step 2

In this setting, we can visualize strings of symbols as horizontal lines in the tree. Derivations are horizontal "slices". A predict step produces a string of symbols from a non-terminal in a production rule (see Figures 2 and 3).

On the other hand, the match action consumes the input string by making the "slices" shrink:

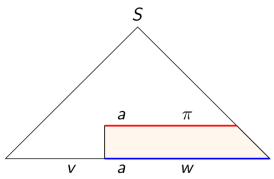


Fig. 4. LL match step 1

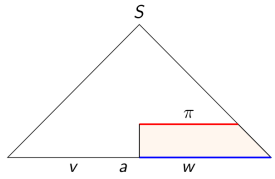


Fig. 5. LL match step 2

3.7.6 *LL(1) Walkthrough.*

Description and Priority. In the LL(1) walktrhough the user can inspect the LL(1) top down predictive parser which uses 1 item of lookahead in the stream (see subsection 2.2 for more details). The user will visualize and interact with the creation of FIRST and FOLLOW sets for each step and see how these help in making a deterministic choice instead of pure guessing. In the overall software perspective this feature has high priority. Beginner students can utilize this tools after the LL walkthrough (non-deterministic) to appreciate the difference.

Stimulus/Response Sequences.

- Input of grammar and text is the same as LL walkthrough.
- System visualizes stack, grammar, input stream and parse tree (dynamically built).
- System selects the next step according to FIRST, FOLLOW and nullable constructions, the symbol on top of the stack and the current token in the input stream.

Functional Requirements.

SA-LL1-1 Create grammar from user input (upload or text).
SA-LL1-2 MiniJava editor, standard editor or file upload for input text.
SA-LL1-3 Visualize stack, grammar, input stream and grammar. At each step highlight all involved entities.
SA-LL1-4 Compute FIRST, FOLLOW and nullable sets in order to decide the next step in parsing.
SA-LL1-5 Enable the user to manipulate the stack and check for correctness in the next step.
SA-LL1-6 Save derivation history with all sets/construction for later inspection.
SA-LL1-7 Save environment (grammar and user input).

3.7.7 LR Walkthrough.

Description and Priority. As with the top-down parsing, LR (bottom-up predictive parser) is represented in the platform as a walkthrough with the user deciding the next step to take (shift or reduce). This setting helps the student understand the transition from non-determinism to the determinism achieved by LR(0). The priority of this feature is set to high since many tools and modules on compiler construction do not emphasize the nondeterminism to determinism transition.

Stimulus/Response Sequences. See paragraph in section 3.7.5.

Functional Requirements. Same as LL functional requirements: 3.7.5. The difference is that the actions in LR are *shift* and *reduce*.

Similar geometric intuitions for LL (see Section 3.7.5 are included for LR bottom-parsing:

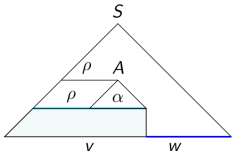


Fig. 6. LR reduce step 1

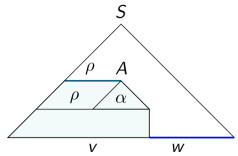


Fig. 7. LR reduce step 2

The reduce step gathers enough symbols on the stack that match the right-hand side of a production and replaces them with the left-hand side. On the other hand the shift action pushes the current input symbol onto the stack, expanding the "slice" in Figures 8 and 9.

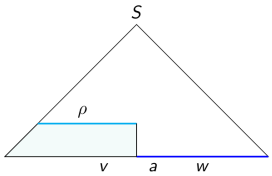


Fig. 8. LR shift step 1

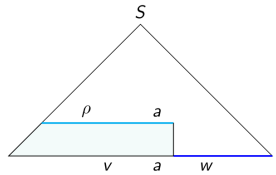


Fig. 9. LR shift step 2

3.7.8 LR(0) Walkthrough.

Description and Priority. In the LR(0) walktrhough the user can inspect the LR(0) bottom up parser which uses items which are production rules with a dot in it to indicate how much of the right hand side has so far been recognized. The user will visualize and interact with the construction of the DFA by using item closures (whose states are sets of items). Once the DFA has been generated the system will visualize how to construct the parsing table from this automaton.

In the overall software perspective this feature has high priority. Beginner students can utilize this tools after the LR walkthrough (non-deterministic) to appreciate the difference.

Stimulus/Response Sequences.

- User inputs the grammar and text to parse.
- System optionally visualizes the construction of the powerset automaton for items.
- System visualizes the stack of items, the automaton, the input stream and the parse tree.

Functional Requirements.

SA-LR0-1 Create grammar from user input (upload or text).

SA-LR0-2 MiniJava editor, standard editor or file upload for input text.

SA-LR0-3 Visualize grammar and how it is used to build the DFA through the computation of closures. Each state will have a set of "dotted items" which indicate how much of the right hand side has been consumed so far.

SA-LR0-4 Enable the user to visualize and/or build the parsing table and check for correctness.

SA-LR0-5 Visualize the items stack, input and the DFA of items and how the parsing works according to the latter (without explicitly showing the parsing table).

3.7.9 Garbage Collection.

Description and Priority. In CC Tutor, garbage collection is visualized in 3 different scenarios: (1) **Mark and Sweep**, (2) **Mark-Sweep-Compact** and (3) **Mark and Copy**. This allows the student to visualize the difference between the different methods of removing unused objects. In the overall software perspective this feature has medium priority since it is a runtime concept.

Stimulus/Response Sequences.

- User inputs the heap graph either directly or indirectly (i.e. through Java code).
- System visualizes step by step the marking of used objects (depth-first search).
- System proceeds with the removal of unused objects.

Functional Requirements.

GC-1 Create heap graph manually, from file or from code

GC-2 System visualizes heap graph, abstract contiguous heap memory (to emphasize the difference between the removal methods) and the free-list if applicable.

GC-3 Enable the user to select the removal method to use (or execute all of them in parallel).

GC-4 Enable the user to input the abstract contiguous heap memory entries which the system will check and highlight the errors.

3.7.10 Compiler Construction Assistant.

Description and Priority. The Compiler Construction Assistant is an advanced feature which aids the student in the implementation of a compiler frontend. This is a step by step tool that accompanies the user through the development of a complete frontend. The job of CCA is to give hints, pointers and defined requirements; most of the development work is done by the student.

In the overall software perspective this feature has high priority since it serves as the next step in the learning curve, after the visualization of algorithms and principles underlying the compiler frontend. In fact, concepts such as environments in semantic analysis are better understood if implemented rather than visualized.

Stimulus/Response Sequences.

- System initializes the environment (either from saved state or new state).
- System guides user through the various sections defined in each phase.
- User implements each feature.
- System checks the solution against predefined tests.

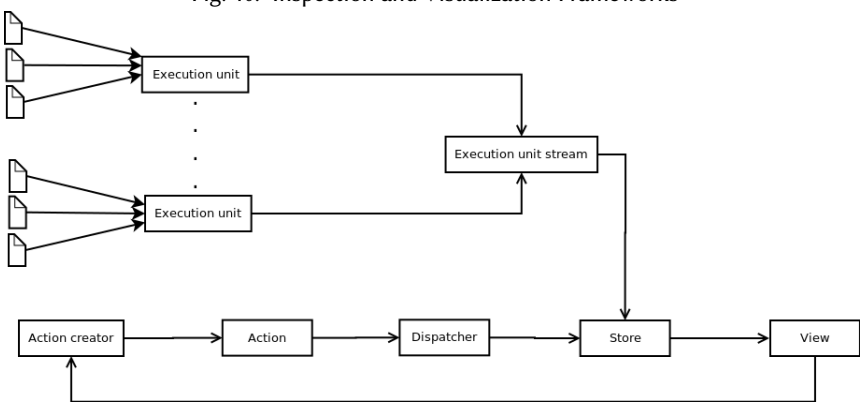
Functional Requirements.

CCA-1 System prepares the environment; a space for storing folders and files and a Java editor.
CCA-2 Enable the user to save the state of the assistant.
CCA-3 Enable the user to build some or all parts of the frontend.
CCA-4 Enable the user to check the validity of their implementation against pre-defined test cases or ones defined by the teacher.
CCA-5 System gives hints to the user on algorithms.
CCA-6 Enable the user to create custom test cases to check the correctness of algorithms.

3.7.11 Inspection and Visualization Frameworks.

Description and Priority. The inspection framework is the layer that splits the algorithm execution into chunks called *execution units*. The latter form an *execution unit stream* which is then supplied to the visualization framework. The relationship between the two is better visualized in a diagram in Figure 10:

Fig. 10. Inspection and Visualization Frameworks



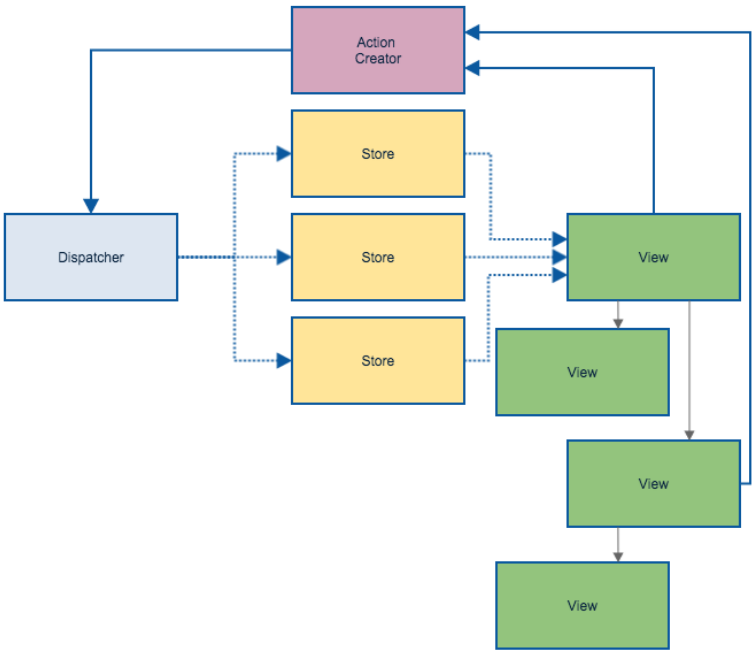
Since this is the vital aspect of the platform, careful design needs to be put in place. The most obvious approach to follow when building graphical user interfaces that respond to changes in the data is the *Model-View-Controller (MVC)* pattern. In a MVC application, a user interaction triggers code in a controller. The controller knows how to coordinate changes to one or more models by calling methods on the models. When the models change, they notify one or more views, which in turn read the new data from the models and update themselves accordingly so that the user can

see that new data. As an MVC application grows the dependencies between views, controllers and models become more intricate leading to complex architectures.

For this reason CC Tutor uses Flux which is the application architecture that Facebook employs for building client-side web applications. This architecture solves part of the problems raised by MVC by using a unidirectional data-flow.

Flux applications have three major parts: *the dispatcher, the stores,* and *the views.* All user interactions within a view call an action creator, which causes an action event to be emitted from a singleton dispatcher. The dispatcher is a single-point-of-emission (central hub) for all actions in a flux application. The action is sent from the dispatcher to stores, which update themselves in response to the action. When a store updates, it emits a change event. In many client-side applications, special views (known sometimes as "controller-views") are responsible for watching for this change event, reading the stores' new data, and passing that data through properties to child views.

Fig. 11. Flux and complexity



In Figure 11 we can see how the addition of multiple stores and views does not create an intricate dependency graph, since the data-flow is unidirectional.

In this setting, in order to change the visualization of a certain algorithm one needs only to use custom views for a specific execution unit stream (which is then fed into the store). Therefore by using the flux pattern it is easier to extend the algorithm visualizations and interactions.

4 IMPLEMENTATION

The most suitable software implementation of this project is as a web platform using PHP Laravel 5.5 for the backend and React JS for the frontend. This separation allows developers to use their own portals/GUIs by calling the API.

Authentication between the API and the React JS client is implemented through OAuth 2.0. This is defined in the API documentation (see Section 3.6).

The implementation of the frameworks defined in this article (in the Laravel backend) uses the *controller-service-repository* pattern. This pattern is suitable for this kind of project which involves a considerable amount of data being passed between client and server. The benefits of such approach are separation of concerns and one-way data-flow.

5 EVALUATION

In order to evaluate the efficacy of the system two evaluation phases are necessary: against the high-fidelity prototype which assesses the visualization and interaction semantics and against the increments (milestones) of the system during the development phase. The latter is preferred to the evaluation against the final product because users are able to guide the development process.

Therefore a mixture of user and heuristic evaluation [Nielsen and Molich 1990] is adopted. User evaluation will be conducted on computer science students using 3 groups: (1) compilers and languages students, (2) students that do not have experience with compilers and (3) students that built a small-medium programming language as an educational experience. User evaluation will be conducted using the "think-aloud" method as well as post-questionnaires.

6 RELATED WORK

The literature regarding algorithm visualization is usually limited to traditional algorithms and data structures such as sorting, trees, graph searching, graph algorithms etc… There is very little effort in tools that assist students in visualizing compiler principles let alone building their own compiler from scratch (frontend or full). Moreover compiler construction modules usually focus on the theoretical aspects then apply these using tools to aid in the implementation rather than learning. In the compiler construction course, students learn how to write a compiler by hand and how to generate a compiler using tools like lex and yacc. However, these tools usually have little or no didactical value [Mernik and Zumer 2003]. In fact these are hands-on approaches useful to study later stages of the compilation process as they do not cover the details in the early parts. This however is not ideal for a full compiler construction module.

Following is an outline of the work that has been done in visualizing and interacting with compiler phases.

The tools described in Vegdahl [2000] merely show the correspondence between the program and an AST which is later annotated with semantic values. Lexical and syntax analysis are absent. Fundamental concepts for a computer science student such as formal languages and models of computation are not covered.

Interactive and visual instructional tools such as LLparse and LRparse [Blythe et al. 1994] enable constructing LL(1) and LR(1) parse tables from appropriate grammars, and for using these constructed tables to parse strings. For LR(1) as an example, the process works by letting the user input the grammar, then they calculate FIRST and FOLLOW sets, graphically constructs a deterministic finite automaton of item sets, and finally constructs the LR(1) parsing table. Upon completion of the constructed table, the user can observe a visualization of the parsing of input strings. The issue with this approach is that it maps the complexity of the algorithms as it is and it does not show why FIRST and FOLLOW sets are useful to decide the next step in the state machine, apart from using these sets to build the parse table. Although it adds a interactivity element, it is not dynamic with respect to the algorithm. Another aspect to note in this article and in the literature in general is the lack of clear incremental implementation of deterministic parsers; students need to understand the transition from nondeterminism (e.g. LL no lookahead - no FIRST and FOLLOW set construction) to determinism (LL1). Thus, CC Tutor provides visualizations and interactions of

both forms in order for the student to get the full picture of why certain constructions are needed (e.g. FIRST and FOLLOW).

Work has been built on top of LLparse and LRparse [Rodger et al. 1997]. It includes algorithms and principles from formal languages in addition to LL(1) and LR(1) construction. As far as parsing concerned, similar visualizations are used with addition of parsing LL(2) grammars, displaying parse trees, and parsing any context-free grammar with conflict resolution.

Khuri and Sugono [1998] developed an algorithm animator [Ross et al. 1997] which visualizes the internal workings of parsing algorithms (specifically LL(1) AND SLR(1)). It visualizes, in the same window, the parsing stack, the input string being processed, the current action and the parse tree being constructed. The layout helps with readability and flow of information, however it lacks the association with the grammar rules and the FIRST, FOLLOW and nullable computations. This hides the complexity of the parsing procedure which is what CC Tutor is tackling.

Chen et al. [2016] developed a compiler teaching assistant where "all procedures in compiling can be dynamically showed in a step-by-step way". Lexical analysis visualization in this project uses pointers to extract the tokens. However the visualization does not add any learning outcome to compiler construction or models of computations students. In fact there it does not cover any algorithm related to abstract machines such as finite automaton and how these are used in the lexical analysis phase. Thus, the lexical analysis phase needs more visualization and interaction points in relation to central concepts in compilers and languages such as regular expressions and finite automata. In terms of syntax analysis, the application developed in [Chen et al. 2016] dynamically shows the parse tree being built from the abstract machine stack. This is an efficient way to visualize the relation between the two. On the other hand since the parsing table is being implemented from FIRST and FOLLOW sets, this does not convey the necessary information to the student as to why certain production rules have been chosen. Moreover, for beginners, didactical value is absent if the process does not show the transition from nondeterminism to determinism during parsing. The software described in this article can be viewed as a simulation of the compilation process instead of an educational tool.

7 CONCLUSION

In this article, a work-in-progress educational platform for compiler construction has been presented. The main output of this study will be a compilation process visualization, interaction system and compiler construction assistant platform, CC Tutor. A secondary deliverable of this study is a novel mathematical framework for formalizing algorithm visualizations and interactions using graph rewriting in order to implement successful user experiences in educational tools as well as determining the efficacy of existing interfaces through the usage of metrics in graph theory.

There are many tools and packages for visualizing and interacting with traditional algorithms, however there has been little effort towards compiler construction educational tools. Moreover a formalization/model of such interactive systems and in general graphical user interfaces has not been properly defined (apart from semantics description languages such as XML).

The CC Tutor project will therefore tackle these issues by delivering a platform which will follow the requirements defined in Section 3 (primary objective) and by formalizing a framework for modeling algorithm visualizations through graph rewriting defined in Section 2.3.1 (secondary objective).

A PROTOTYPES

Low-fidelity prototypes have been developed in order to perform early user and heuristic evaluation. The platform will be based on the evaluations results as well as expert analysis.

The prototypes can be found in the project's documentation repository: https://github.com/UoBCS/cc-tutor-docs/blob/master/software-design/prototypes.pdf

REFERENCES

Andrew W. Appel and Jens Palsberg. 2009. *Modern compiler implementation in Java.* Univ. Press.

Luciano Baresi and Reiko Heckel. 2002. Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. *Graph Transformation Lecture Notes in Computer Science* (2002), 402âĂŞ429. https://doi.org/10.1007/3-540-45832-8_30

C. A Benware and E. L Deci. 1984. Quality of Learning With an Active Versus Passive Motivational Set. *American Educational Research Journal* 21, 4 (Jan 1984), 755âĂŞ765. https://doi.org/10.3102/00028312021004755

Stephen A. Blythe, Michael C. James, and Susan H. Rodger. 1994. LLparse and LRparse. *Proceedings of the twenty-fifth SIGCSE symposium on Computer science education - SIGCSE 94* (1994). https://doi.org/10.1145/191029.191121

Xiwen Chen, Hanfei Lin, Yufei Liang, and Xiaoming Ju. 2016. A Process-Visible Compiler Aimed for Teaching Assistant. *Proceedings of the 2015 Conference on Education and Teaching in Colleges and Universities* (2016). https://doi.org/10.2991/cetcu-15.2016.12

P. Heiko and P. Florian. 2011. A Formal Ontology on User Interfaces - Yet Another User Interface Description Language? (2011).

Sami Khuri and Yanti Sugono. 1998. Animating parsing algorithms. *ACM SIGCSE Bulletin* 30, 1 (Jan 1998), 232âĂŞ236. https://doi.org/10.1145/274790.274303

M. Mernik and V. Zumer. 2003. An educational tool for teaching compiler construction. *IEEE Transactions on Education* 46, 1 (2003), 61âĂŞ68. https://doi.org/10.1109/te.2002.808277

Torben ÃĘ. Mogensen. 2010. *Basics of compiler design.* Department of Computer Science, University of Copenhagen.

Jakob Nielsen and Rolf Molich. 1990. Heuristic Evaluation of User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90).* ACM, New York, NY, USA, 249–256. https://doi.org/10.1145/97243.97281

Susan H. Rodger, Anna O. Bilska, Kenneth H. Leider, Magdalena Procopiuc, Octavian Procopiuc, Jason R. Salemme, and Edwin Tsang. 1997. A Collection of Tools for Making Automata Theory and Formal Languages Come Alive. *SIGCSE Bull.* 29, 1 (March 1997), 15–19. https://doi.org/10.1145/268085.268089

Rockford J. Ross, Christopher M. Boroni, Frances W. Goosey, Michael Grinder, and Paul Wissenbach. 1997. WebLab! A Universal and Interactive Teaching, Learning, and Laboratory Environment for the World Wide Web. *SIGCSE Bull.* 29, 1 (March 1997), 199–203. https://doi.org/10.1145/268085.268160

Hayo Thielecke. 2015. Implementing functional languages with abstract machines. (December 2015). Retrieved November 23, 2017 from https://www.cs.bham.ac.uk/~hxt/2015/compilers/compiling-functional.pdf

Hayo Thielecke. 2016. LL and LR parsing. (October 2016).

Steven R. Vegdahl. 2000. Using Visualization Tools to Teach Compiler Design. In *Proceedings of the Fourteenth Annual Consortium on Small Colleges Southeastern Conference (CCSC '00).* Consortium for Computing Sciences in Colleges, USA, 72–83. http://dl.acm.org/citation.cfm?id=369340.369325

Colin Ware. 2011. *Visual thinking for design.* Elsevier/Morgan Kaufmann.