An Interactive Learning Platform for Compiler Data-Flow Analysis

Ayrton Massey

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2016

Abstract

Data-flow analysis is one of the cornerstones of modern compiler optimisation. A thorough understanding of the processes involved is essential to further exploration of the subject. A tool which allows exploration of data-flow analysis in an interactive environment would prove invaluable to students encountering the topic for the first time.

This report describes the design and implementation of an interactive system to simulate and visualise forms of data-flow analysis on simple assembly-like programs. The system is evaluated by in terms of user experience and the achievement of learning outcomes, through self-assessment and by examining usage data collected during the evaluation period.

The software was wildly successful in increasing the learning capacity of the evaluation subjects. The results collected present strong evidence of the need for more interactive learning tools, particularly in engineering subjects. Responses to the feedback survey were overwhelmingly positive, achievement of learning outcomes was well above average, users were kept engaged by the platform, and all but one participant stated that they would use the system alongside other methods of study.

Acknowledgements

First and foremost I would like to thank the project's supervisor, Hugh Leather, for his continued support throughout the project and for volunteering his time and effort during particularly unfortunate circumstances.

I am extremely grateful for the assistance provided by Drs. Alan Smaill and Iain Murray, and the UG4 project co-ordinator Prof. Don Sannella.

My sincere gratitude goes to Dr. Steve Gregory of the University of Bristol for distributing the application to his students, and to the participants of the evaluation without whom this project would not have been possible.

Finally, I would like to thank my family and friends for their support and encouragement throughout my academic career.

Table of Contents

Chapter 1

Introduction

This chapter gives a short introduction to the topic of data-flow analysis, describes motivations and desired outcomes for the project and provides a brief summary of contributions.

1.1 Data-Flow Analysis

Data-flow analysis is a tool for analysing the flow of data through a program at various points in its execution. Analysis is performed over a control-flow graph, computing the properties of values flowing *in* and *out* of each node. Many forms of data-flow analysis exist to compute various properties, for example *liveness analysis* identifies variables which will be used in future instructions and *available expressions* identifies those expressions whose value has been previously computed at some point in the control-flow graph.

This analysis is used to inform optimisations which can be performed on a given program. Using the example of *liveness analysis*, the values computed can be used to optimise register allocation: a variable which is not live at a given point does not need to be allocated to a register, enabling more efficient use of available resources.

Data-flow analysis is not only useful in compiler optimisation. The information gathered can be used in other ways, such as identifying unsafe operations in PHP web applications[1] by monitoring sanitization[1] of variables which have been assigned to user input.

[1] To *sanitize* a user input is to remove any potentially dangerous elements from said input; for example, if a user input string is to be inserted into the HTML of a webpage it could be sanitized by replacing instances of $<$ and $>$ with < and >, respectively. This would prevent that input being misinterpreted as HTML and thus avoid malicious scripts contained within that input from being executed.

1.2 Motivations

This project was inspired by the project's supervisor, Hugh Leather. The original concept was an online tutor for data-flow analysis which would allow users to simulate an analysis on simple programs. The user could vary parameters, such as the data-flow in question or the order in which nodes are evaluated, and examine the resulting solution.

As lecturer of the Compiler Optimisations (COPT) course at the University of Edinburgh, Dr. Leather desired a system which could teach students the foundations of the course in a more interactive format than standard lectures. The system should be suitable for hosting on the course web page in order to provide access to all students.

My personal interest in this project stemmed from a desire to use my practical skills to increase my capacity for understanding theoretical content. As noted in our early discussions, many students find it difficult and time consuming to read and understand material from the course textbook. Presenting this information in such a way that it could be easily digested by even a novice to Computer Science proved an exciting challenge.

1.3 Objectives

The main aim of this project was to create an interactive system to teach students the basic principles of data-flow analysis in compilers.

This would take the form of a web application using visual components which could be combined in different ways, for example to present a series of tutorials on data-flow analysis or to provide a sandbox environment to explore. The content of the application would cover a range of topics from the basics of data-flow analysis to algorithms and frameworks for solving generic data-flow problems.

The application would be aimed at students of the COPT course and as such would be based on material from the course textbook *Engineering a Compiler, 2nd ed.*[2] by Keith D. Cooper and Linda Torczon. The application could then be extended to cover the topic in more depth using content from *Compilers: Principles, Techniques and Tools, 1st ed.*[3] by Alfred V. Aho, Ravi Sethi and Jeffery D. Ullman (commonly referred to as the *Dragon Book*).

Users would be able to interact with the system by providing simple assembly-like programs, altering parameters of the analysis and stepping through a simulation. Elements of the simulation such as the current state and the control-flow graph of the program would be visualised on-screen and update as the simulation progressed. Each of these elements would be linked visually to show how the concepts relate.

The application would be tested on real users. It would be evaluated in terms of user experience by analysing interactions with the system and conducting a user experience survey; achievement of learning outcomes would be assessed by examining responses to questions built into the software and self-assessment by the user.

1.4 Summary of Contributions

The final version of the software is an online learning platform featuring a sandbox simulation for data-flow analysis and a series of interactive tutorials which provide the background knowledge required in order to use it.

A summary of contributions is as follows:

- Simulation of pre-defined data-flows using generic framework models. (p. 24)
- Simulation of analysis on user-defined programs using the ILOC [2, appx. A] language from *Engineering a Compiler*. (p. 27)
- A parser for the ILOC language, with extended grammar rules. (p. 27)
- A built in code editor with the ability to share programs (p. 36)
- Simulation using the round-robin iterative algorithm allowing step-by-step, instant or automated playback. (p. 26)
- Visualisation of the following simulation elements:
    - Control-flow graph (p. 34)
    - Simulator state incl. currently evaluated node, framework etc.
    - Table of results displaying data flowing in / out of each node. (p. 35)
    - Hasse diagram of meet semi-lattice (p. 35)
- Tutorials covering basics of the topic with interactive elements (p. 41)
- Interactive test to assess achievement of learning outcomes (p. 43)
- An API to record user interaction events modelled on the Google Analytics event tracking system (p. 49).

Detailed explanations of the terminology mentioned above can be found in chapter 2. A quick summary of terms can be found in the glossary in appendix A.

Chapter 2

Background

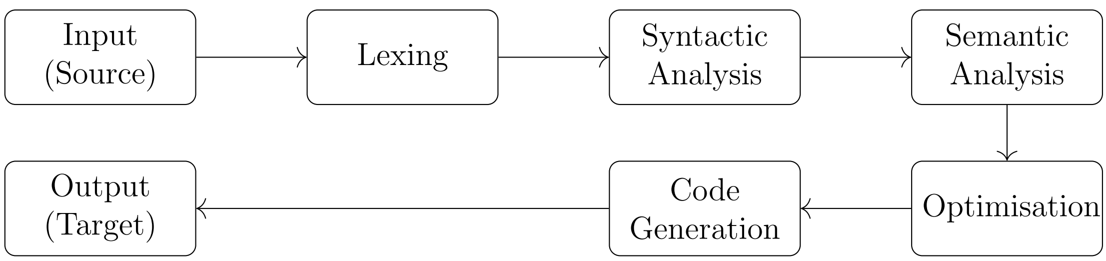
This chapter briefly covers the necessary background information required to understand the project, both to inform the reader and to demonstrate understanding of the topic.

2.1 Modern Compiler Design

In Computer Science, a compiler is a tool for translating source code written in one language into another target language. In general, modern compilers are divided into 5 distinct stages[1]:

- *Lexing*, which converts input into a stream of tokens.
- *Syntactic analysis*, which parses these tokens into a symbolic representation known as a parse tree.
- *Semantic analysis*, which transforms the parse tree into an intermediate representation (IR), often in the form of an abstract syntax tree (AST).
- *Optimisation*, which applies transformations to the IR in order to increase the program's size or run-time performance.
- *Code generation*, which translates the IR into the target language.

Whilst the parse tree generated by syntactic analysis represents the exact structure of the input, semantic analysis strips this down into the IR which contains only the information required to understand the program's meaning.



[1] This is an over-simplification, but one which is suitable for purpose of this report.

2.1.1 Control-flow Graph

The IR may be used to form other representations; for example a control-flow graph (CFG) models the possible execution paths for a given program. Each node in the graph represents an instruction or block of instructions, each edge represents an execution path leading from one instruction to another. A node can have multiple outward edges if it is a branching instruction, and branches may point backward in the control-flow. An example of a simple control-flow graph can be seen in fig. 2.1.

A *point* in the control-flow graph refers to some point along the edges of the graph. In data-flow analysis we usually deal with sets of values at the *in* and *out* points of each node, i.e. the point where the *in-edges* meet and the *out-edges* originate, respectively (shown in magenta on the right).

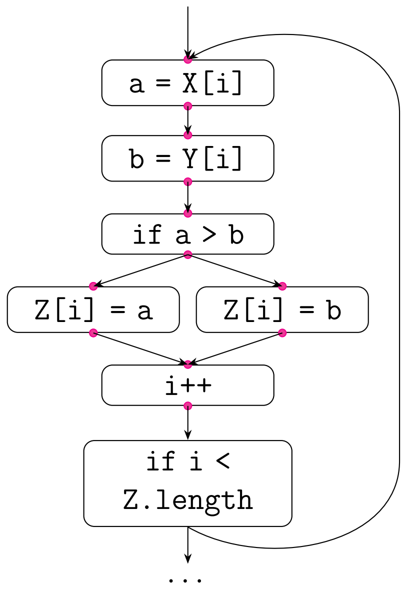


Fig. 2.1: A control-flow graph.

2.2 Data-Flow Analysis

During construction, the IR is often annotated to assist in optimisation and code generation. Data-flow analysis is one of the techniques used to perform this annotation. In general, it is performed in the semantic analysis or optimisation stage and is used to compute information about data flowing *in* and *out* of each node in the program's CFG. Many types of analysis can be performed and the information gathered is used to inform the decisions of optimising compilers. A brief list of analyses and their purposes can be found in appendix B.

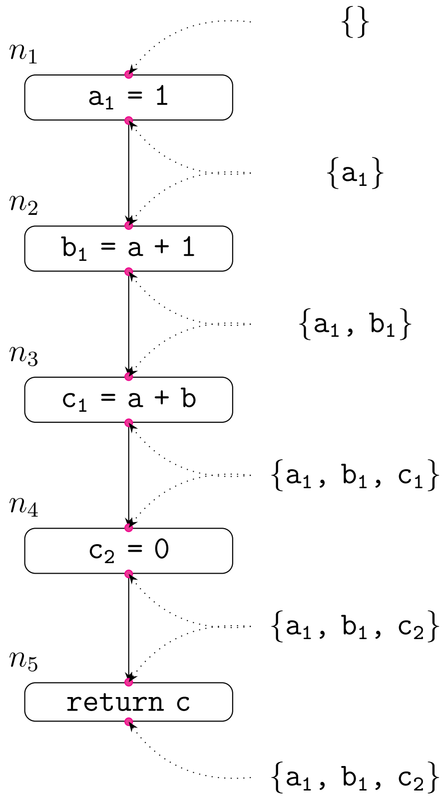
2.2.1 A Simple Example

An oft-used example of data-flow analysis is that of *reaching definitions*, which we will demonstrate here due to its simplicity. Reaching definitions computes the set of assigned values, or definitions, which are available for use at a given point in the CFG. For example, the statement `a = b + c` defines a value of `a`. A variable may have multiple definitions, each referring to a value assigned to that variable, so we label each definition with an index to distinguish it from other definitions of the same variable.

In reaching definitions, a definition d is said to *reach* a point p if the variable holding d is not reassigned along at least one path from the d to p.

Let us take the example in fig. 2.2. The first node defines the variable `a`, generating the definition $\mathtt{a}_1$. This definition reaches every subsequent point in the control-flow graph since `a` is not reassigned. The definition $\mathtt{c}_1$, however, does not reach very far at all – `c` is reassigned in n_4, replacing $\mathtt{c}_1$ with $\mathtt{c}_2$. Fig. 2.2 has been annotated with the set of reaching definitions at each point in the program. We have combined the *in* and *out* points to save space.

Fig. 2.2: A control-flow graph.



2.2.2 Properties of Data-Flows

Data-flows have direction. Reaching definitions is a *forward flow problem*; values flow from the entry node of the CFG to the exit node.

Values at each point are determined using data-flow equations. For example, the equations for reaching definitions (defined in terms of *in* and *out*) at a given node n are:

$$\mathrm{In}(n) = \bigcup_{p \in preds} \mathrm{Out}(n)$$

$$\mathrm{Out}(n) = \mathrm{DefGen}(n) \cup (\mathrm{In}(n) \setminus \mathrm{DefKill}(n))$$

The equations for $\mathrm{In}(n)$ and $\mathrm{Out}(n)$ are often referred to as *meet* and *transfer* functions. Other symbols such as $\mathrm{DefGen}(n)$ and $\mathrm{DefKill}(n)$ are referred to as local information, as they are constant values containing information about the current node. In a forward flow problem the meet function calculates the In set from the Out sets of the node's predecessors; the transfer function computes a node's Out set from its In set and local information, thereby *transferring* values through a node. In *backward* data-flow problems, this is the opposite – entry instead of exit, In instead of Out, successors in place of predecessors.

2.2.3 Lattices

Sets of values in a data-flow problem have a partial order; this is necessary for the data-flow to terminate (see §2.4.2). The meet function imposes this partial order, which may be be expressed using a structure known as a *meet semi-lattice*. A meet semi-lattice consists of a set of possible values L, the meet operator $\wedge$, and a *bottom element* $\perp$. The semi-lattice imposes an order on values in L such that:

$$\begin{array}{ll} a \geq b & \text{if and only if } a \wedge b = b \\ a > b & \text{if and only if } a \wedge b = b \text{ and } a \neq b \\ a \geq \bot & \forall a \in L \end{array}$$

Partial orders can be visualised using a Hasse diagram. Each node represents a value in L, whilst each edge represents an order between nodes: an edge from a to b indicates that $a \geq b$. To calculate the meet of two sets a and b, we simply find the greatest (with respect to the order) common descendent of a and b in the diagram.

The meet semi-lattice for a data-flow can be expressed using a *Hasse diagram*, shown in fig. 2.3. In order to reduce clutter, a transitive reduction has been performed on the graph; for instance, the edge between {a} and {a,b,c} is represented by the path {a} $\rightarrow$ {a,b} $\rightarrow$ {a,b,c}.

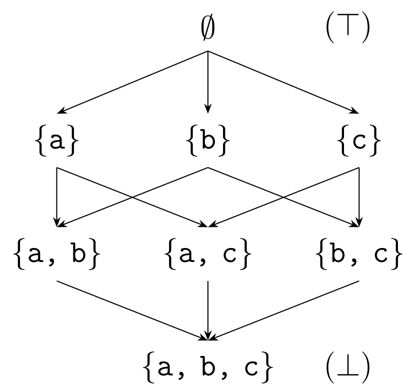


Fig. 2.3: A Hasse diagram for the meet function $a \wedge b = a \cup b$.

2.2.4 Bit-Vector Data-Flows

The data-flows present in this project are commonly known as bit-vector data-flows. In these problems values are single items, such as a variable or definition. A set of values can be represented as a vector of bits, each bit representing the presence or absence of a particular value. The reaching definitions example from the previous section is a bit-vector data-flow problem. Fig. 2.4 shows how this can be applied to the example in §2.2.1.

$$\begin{array}{ccccccc} & & & \mathtt{a}_1 & \mathtt{b}_1 & \mathtt{c}_1 & \mathtt{c}_2 & \\ \mathrm{Out}(n_5) & = & \{ & 1 & 1 & 0 & 1 & \} \end{array}$$

Fig. 2.4: A bit-vector for the CFG in fig. 2.2

Tuple-valued data-flows consider values as tuples instead of single items. One such example is constant propagation, in which variables are paired with one of three elements: *undef* ($\top$), *nonconst* ($\bot$) and *const*. A variable is initially paired with *undef*. When it is assigned a constant value, we assign it that particular value. If it is later assigned another value, we assign it *nonconst*. This can be expressed as the meet function, $\wedge$, seen in fig. 2.5. The values form the semi-lattice in fig. 2.6.

$$
\begin{aligned}
\mathit{nonconst} \wedge c &= \mathit{nonconst} && \text{for any constant } c \\
c \wedge d &= \mathit{nonconst} && \text{for any constants } c \neq d \\
c \wedge \mathit{undef} &= c && \text{for any constant } c \\
\mathit{nonconst} \wedge \mathit{undef} &= \mathit{nonconst} && \\
x \wedge x &= x && \text{for any value } x
\end{aligned}
$$

Fig. 2.5: Equations describing the constant propagation meet function.

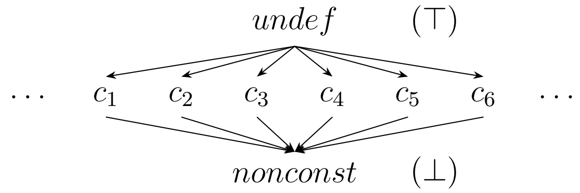


Fig. 2.6: A Hasse diagram for the meet function described in fig 2.5.

2.3 Algorithms for Analysis

There exist a number of algorithms for performing data-flow analysis. This section contains a brief overview of the methods proposed in the related reading[2] [3].

2.3.1 Round-Robin Iterative Algorithm

The simplest algorithm used to solve data-flow problems is the round-robin iterative method. We consider each node of the CFG in turn, calculating the In and Out sets using our data-flow equations. This is a fixed-point computation: we iterate until our value sets stop changing between iterations. Algorithm 2.1 shows the round-robin iterative method for solving reaching definitions.

Algorithm 2.1: Iterative Round-Robin Method for Reaching Definitions

```
for each (node n in the CFG)
    In(n) = ∅;

while (changes to any sets occur)
    for each (node n in the CFG)
        In(n)  = ∪_{predecessors p of n} Out(p);
        Out(n) = DefGen(n) ∪ (In(n) \ DefKill(n));
```

2.3.2 Worklist Algorithm

The above solution holds value in its simplicity, but it is trivial to find a more efficient solution. A node's sets will only change if the Out sets of its predecessors change. Thus, we may use a worklist: starting with the initial node, we calculate the Out set for that node; if it changes we add the node's successors to the list. We continue this process until the worklist is empty. Algorithm 2.2 shows the worklist method for solving reaching definitions.

Algorithm 2.2: Worklist Method for Reaching Definitions

```
worklist = [ n0 ];

while (worklist is not empty)
    n = pop(worklist);
    In(n)  = ∪_{predecessors p of n} Out(p);
    Out(n) = DefGen(n) ∪ (In(n) \ DefKill(n));
    if (Out(n) has changed) append n to worklist;
```

2.3.3 Structural Algorithm

A third algorithm takes an entirely different approach. Instead of iterating over each node, we perform a number of simple transformations on the graph in order to reduce it to a single region. We then expand each node, calculating the In and Out sets of our reduced nodes as we expand them. A more detailed explanation of this concept may be found in the *Dragon Book*[3, p. 673].

2.4 Data-Flow Frameworks

It is possible to model data-flow problems using a generic framework. This allows us to use the same algorithm for multiple problems by specifying the following constraints[3, p. 680]:

- The *domain* of values on which to operate;
- The *direction* in which data flows;
- A set of *data-flow equations* including the *meet operator* $\wedge$ and the set of *transfer functions* F[2];
- The *boundary* value $v_{BOUNDARY}$ specifying the value at the entry or exit to the CFG; and
- The *initial value*, $\top$, at each point in the graph.

[2] The function corresponding to a particular node/block B is denoted F_B

2.4.1 Algorithm for General Frameworks

Algorithm 2.3, adapted from the one in the *Dragon Book*[3, p. 691], computes the value sets at each node using the elements of our general framework.

Algorithm 2.3: Data-Flow Analysis of General Frameworks

```
Meet_BOUNDARY = v_BOUNDARY;

for each (block B in the CFG)
    Meet_B = ⊤;

while (changes to any Transfer occur)
    for each (block B in the CFG)
        Meet_B     = ∧_{priors P of B} Transfer_P;
        Transfer_B = F_B(In_B);
```

Instead of referring to the value sets as In and Out as the *Dragon Book* does, we may call them Meet and Transfer. This allows us to generalise our algorithm to both forward and backward analyses; in the forward direction Meet is In whereas in the backward direction it is Out (and vice-versa for Transfer).

We first initialise the Transfer set using the boundary condition, then initialise each node's transfer set to our initial value $\top$.

Next, we perform a fixed-point computation on the CFG, evaluating each node's Meet and Transfer sets using our data-flow equations until the sets stop changing.

The meet is taken over a node's *priors*: in the forward direction, the node's predecessors; in the backward direction, the node's successors.

This algorithm can be applied to any framework. In fact, all of the data-flow problems in appendix B may be solved using this process.

2.4.2 Conditions for Termination

We must be careful when constructing our general frameworks. If our value sets continuously change we may never reach a fixed-point and thus our computation will never halt.

To avoid this, our frameworks must satisfy the following conditions[3, p. 684]:

- The set of transfer functions, F, contains the identity function[3];
- F is closed under composition: that is, for any two functions f and g, $f(g(x))$ is also in F;
- F is monotone; and
- The domain and the meet operator, $\wedge$, must form a meet semi-lattice.

[3] The identity function maps its input to its output, i.e. $F(x) = x$

These conditions ensure that during every iteration of the algorithm the values at each point will either become *smaller* (with respect to the partial ordering) or stay the same. Since F is monotone, all of the sets must eventually stop changing **or** reach the bottom element of the lattice, $\perp$, at which point they cannot change any further.

Chapter 3

Related Work

This chapter will discuss the merits and drawbacks of related work and identify aspects of said work which may be applied to this project.

3.1 Introduction

Although there exists a wide range of literature dealing with data-flow analysis in compilers *independent* of interactive tutoring and vice-versa, the combination of the two has yet to be explored. Therefore, research has been widened to two main areas: the topic of data-flow analysis, and advancements in interactive tutoring software.

Chapter 2 provided a brief introduction to data-flow analysis with reference to two major textbooks; the first, *Engineering a Compiler, 2nd ed.*[2] by Keith D. Cooper and Linda Torczon, serves as an excellent introduction to the topic and an overview of some of the more complex elements. The second, *Compilers: Principles, Techniques and Tools, 1st ed.*[3] (often referred to as the *Dragon Book*) provides a more theoretical discussion of the material.

This chapter discusses the second area of research, advancements in interactive tutoring software.

3.2 Online Learning Platforms

In recent years there has been a surge in online learning platforms such as Khan Academy and Coursera which deliver a series of lectures or tutorials over the internet.

The videos are very similar in format to a traditional lecture, consisting of a voice-over accompanied by related imagery or text-based slides. However, the advantage over the traditional format is two-fold: the ability to view the content at the student's convenience, rather than in a specific location and time slot, allows students to organise their learning around their own schedule. Users are also able to replay portions of the class they have missed or struggled to understand. In this way, learning may be tailored to a student's specific needs.

These platforms often integrate an interactive or practical element into their teaching; whilst services such as Coursera offer feedback through the more traditional peer-assessed hand-in[4], others such as Khan Academy and HackerRank provide live demos and code interpreters to develop understanding.

For example, in its series on linked lists[5], HackerRank presents the user with a series of coding challenges. The user inputs code to solve a given problem (fig. 3.1), such as reversing a linked list, and the site verifies their solution. Khan Academy's introduction to binary search[6] demonstrates the increase in efficiency with a visual representation of both linear and binary search (fig. 3.2), allowing the user to step through each and compare the number of steps taken for themselves.

These demonstrations enable a different kind of learning than that which can be found in the classroom. Students are able to use their intuition to form their own understanding of the material and take a much more active role in their own development. This kind of learning, referred to as *kinaesthetic* learning, can be incredibly effective (see §3.4.2).

There are some drawbacks to this model: whilst students may learn at their own pace, the only information available is that on the page or in the video they are watching. If they have any questions or struggle to understand the material as presented, there is no lecturer or teaching assistant to provide alternate explanations or answer queries. Some platforms provide discussion forums for peer support, but this can result in "the blind leading the blind" – without an expert hand to guide them, students may cement incorrect knowledge, defeating the purpose of the learning platform.

Fig. 3.1: Code editor from HackerRank[5]

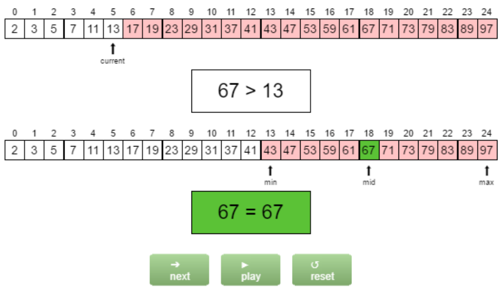


Fig. 3.2: Binary search visualisation from Khan Academy[6]

3.3 Simulators as a Teaching Aid

In his article *Evaluating A System Simulator For Computer Architecture Teaching And Learning Support*[7], Mustafa discusses the design & implementation of a system which integrates the simulation of a compiler, CPU and operating system to aid in the teaching of undergraduate computer architecture and operating systems modules.

Although brief, the article provides valuable insight into the design and evaluation of such teaching software. The primary source of feedback came from an opinion survey using a 5-point Likert scale for quantitative analysis and open-ended questions for qualitative feedback. In addition, students were administered a test to assess their knowledge pre- and post- use of the system.

The evaluation results highlight an important point to consider when designing teaching software; over 20% of respondents indicated that they spent more time learning how to use the software than they did completing the given exercises. An equal number of students reported that the simulator left them more confused than before. In addition, 7.1% of respondents said that the simulator was too complicated to use effectively in their tutorials.

However, the converse of these results gives a positive outlook for this project: 72.4% of respondents disagreed with the above statements and 79.3% believed that the system improved their understanding of the topics covered in lectures. Over 95% of respondents agreed that the simulator was more useful than reading textbooks or searching the internet in helping them understand the material.

3.4 Learning and Teaching Styles

A 1988 report by Richard M. Felder and Linda K. Silverman, entitled *Learning and Teaching Styles in Engineering Education*[8], categorises students' learning methods and describes ways in which professors may target specific categories in their teaching strategy.

Felder refers to the learning modalities (or VAK) model proposed by Walter Burke Barbe, which defines three modalities:

- **visual** – those who learn best by viewing images and diagrams;
- **auditory** – those who learn best by listening or speaking aloud; and
- **kinaesthetic** – those who learn best by actively experiencing things, learning by doing.

The report claims that most people of college age and above, the intended audience of this project, identify as *visual* learners – those who benefit from charts, diagrams and such. In contrast, the presentation of content in university courses is primarily *auditory* (via lectures) or a visual representation of auditory information (i.e. words or mathematical formulae).

The survey results from Mustafa[7, p. 103] support this observation: although only a small sample (N=54) responded, 31% of respondents identified as visual learners, while a mere 6% identified as auditory learners. The majority of students (52%) identified as kinaesthetic learners, indicating that taking a hands-on approach is a valuable tool for learning.

It should be noted that Felder discounts kinaesthetic learning from his report as he considers the *learning by doing* to be a separate category, perceiving the remaining attributes of kinaesthetic learning to have little value in engineering. A later observation by Felder agrees with Mustafa, stating that engineers are "more likely to be active than reflective learners"[8, p. 678].

3.4.1 Ability to Identify Own Learning Style

In her PhD dissertation *"Individual Differences in Learning: Predicting One's More Effective Learning Modality"*[9], Beatrice J. Farr claims that students are able to accurately predict the learning style in which they perform best:

> "An experiment with 72 college students confirmed that individuals could accurately predict the modality in which they could demonstrate superior learning performance. The data also revealed that it is advantageous to learn and be tested in the same modality and that such an advantage is reduced when learning and testing are both conducted in an individual's preferred modality."[10, p. 242]

Coffield[11, p. 120] disagrees with this, based on an observation by Merrill[12] that "most students are unaware of their learning styles and so, if they are left to their own devices, they are most unlikely to start learning in new ways". This would indicate that basing teaching methods on a student's preferences is damaging to their education. This is, however, a misinterpretation; the actual text of Merrill reads:

> "... a student must engage in those activities ... that are required for them to acquire a particular kind of knowledge or skill ... Most students are unaware of these fundamental instructional (learning) strategies and hence left to their own are unlikely to engage in learning activities most appropriate for acquiring a particular kind of knowledge or skill."[12, p. 4]

Merrill later argues that the optimal strategy for teaching is decided first and foremost by the content being taught, then fine-tuned to the learner's preferred style[12, p. 4]. By not knowing the most effective strategy for learning the content they are studying, a student limits his or her potential. However, the appropriate strategy *should* in fact be tailored to the student's preferred learning style to obtain the best results.

3.4.2 Criticism of Learning Modalities

Some of the criticism levied at the concept of learning styles is that the idea of *matching* – exclusively teaching a student based on his or her preferred learning style – is harmful to a student's education. Although Coffield's[11, p. 120] reasoning is flawed, the conclusion drawn by him and Merrill[12, p. 4] is sound: by allowing students to exclusively use their preferred way of learning, students are at risk of missing out on more effective methods of study.

Felder's report mentions a study carried out by the Socony-Vacuum Oil Company, which concludes that:

> "...students retain 10 percent of what they read, 26 percent of what they hear, 30 percent of what they see, 50 percent of what they see and hear, 70 percent of what they say, and 90 percent of what they say as they do something."[8, p. 677]

This indicates that by relying solely upon auditory methods, professors can only hope to convey as little as 26% of the desired material to their students. Felder advocates using a mixture of teaching methods in order to appeal to all students' preferred learning styles.

3.5 Summary

While there may be some disagreement over the validity of learning styles or modalities, even its critics seem to agree that there is value in varying the methods of teaching in use. At present, the only available resources for learning data-flow analysis are lecture slides, videos and textbooks. These are mostly auditory and sometimes visual methods of teaching, with little kinaesthetic learning involved. There is a clear need for more active study; although 52% of students identified as learning best through kinaesthetic learning[7, p. 103], there is almost no support for this method of study.

Given the relative success of the examples discussed in this chapter and the apparent lack of any such resources for data-flow analysis, there is a strong precedent for this project and as such a system will be developed to fulfil this role.

However, it is important to note the mistakes made by the examples discussed here. The simulator designed by Mustafa was deemed too complex[7, p. 103] and a detriment to their learning by over 20% of respondents. This project will seek to ensure that that content is clear and concise on order to produce an effective learning resource.

To evaluate the success of the system this project will build upon the methods presented by Mustafa, aggregating opinion using a Likert scale and examining this data to judge overall satisfaction with the software and identify specific areas for improvement.

Chapter 4

Design

This chapter discusses the high-level design and architecture of the system, including motivations for design choices and solutions to conceptual problems.

4.1 Introduction

The original goal of this project was to produce an online simulation of data-flow analysis to show students how data-flow works. As discovered in the related reading, however, a large portion of users of a similar system found it to be too complex and claimed to have spent more time studying the software itself than gaining useful knowledge. The literature also raised another key point; a wide variety of learning techniques are necessary to gain a true understanding of a topic.

For these reasons the proposal has been extended to create a comprehensive learning platform. In addition to the simulator the software will provide supporting lectures to explain the basic concepts and gradually introduce each element of the simulation. These lectures will include visual and interactive elements to engage the user through a range of learning styles. It is hoped that the system will prove a valuable tool for learning alongside existing resources such as textbooks and lectures.

4.2 Design Constraints

In this section, design constraints are identified and potential solutions are suggested.

4.2.1 Technical Constraints

As the intention is to make such a system available to COPT students via the University, the following technical considerations must be made:

- The system must be distributed to all students in some format;
- This format must be functional on and compatible with a wide range of devices owned by said students;
- The system must be secure and require little maintenance; and
- The system must be hosted on some platform available to the University.

In order to meet these criteria the system must rely on as little technology as possible. The easier the platform is to host and distribute, the more likely it is to be made available to students – a system which uses new technology and requires its own dedicated hosting would be more difficult to set up and maintain than one which can be deployed to existing hardware. The application could be developed using purely client-side technologies such as JavaScript, HTML and CSS. These static files may be distributed over HTTP using any standard web server such as the one already used to host the course webpage.

Efforts must be made to keep the performance of the system consistent across a range of web browsers and devices; although all modern web browsers are capable of interpreting these types of content there are subtle differences which may break functionality on one platform but not others.

It is also necessary to ensure that the system is secure to avoid damaging University or student property. Provided that user input is sanitized and scripts are only included from trusted sources, the sandboxed environment inherent to modern web browsers should meet the remaining security requirements.

Use of popular libraries and frameworks such as jQuery and Bootstrap will be encouraged as this provides a number of benefits. The user will experience reduced load times since they have likely already downloaded the required files, and such libraries will provide security and robustness due to their wide use and active development. Likewise, the more popular a library is the more documentation and resources will be available to aid in developing the best possible system.

The specific technologies used in each are detailed in the implementation (chapters 5 & 6).

4.2.2 Content Constraints

To be viable as a learning platform, the system must:

- Appeal to a range of learning styles;
- Provide comprehensive coverage of the topic at hand;

- Ensure the content included is correct, clear and concise; and
- Maintain a shallow learning curve, gradually introducing students to each topic or element of the simulation.

Extending the proposed system into a comprehensive learning platform will provide a range of ways in which to study data-flow analysis. Students will have the choice to use the areas of the software which most appeal to them and content will be presented using a mixture of interactive, visual and textual formats.

As the aim is to assist students of the COPT course, coverage of topics will be prioritised based on their inclusion in the course syllabus and whether they are required to understand the simulation:

- Basic principles of data-flow analysis;
- A few simple data-flows, including:
    - Reaching Definitions
    - Liveness Analysis
    - Available Expressions
- Round-robin fixed-point algorithm;
- Effect of orderings on analysis efficiency;
- Generic frameworks for data-flow analysis;
- Hasse diagrams and lattice representation of values; and
- Conditions for analysis termination.

If the content delivered by the platform is incomplete, incorrect or too difficult to understand, the system will not present a viable alternative method of study. When evaluating the system it will be important to assess the clarity and presentation of the material, and the ease of use of the platform as a whole.

4.3 System Architecture

The system will be divided into two main areas of development. The back-end of the system controls the logic and state of a data-flow analysis simulation. The front-end components will hook into and expose elements of the back-end, visualising the state of the simulation and allowing the user to interact with it. See fig. 4.1 for a visual representation.

Object-oriented design will be paramount, allowing functionality to be extended and shared between similar elements. In addition to speeding up development of the system by reducing the need to duplicate code, this is generally good design practice. Components will be written as re-usable modules which can be combined in different ways, for example to present the user with a series of lectures or an interactive simulation. See §?? for more details on the user interface.

Each component needs to be be self contained; it should be possible to instantiate more than one of each type of component at once, including the simulator itself. This would allow, say, displaying the annotated control-flow graphs of two simulations side-by-side. The simulation will also be independent of the user interface so that it may be adapted for re-use in other projects.

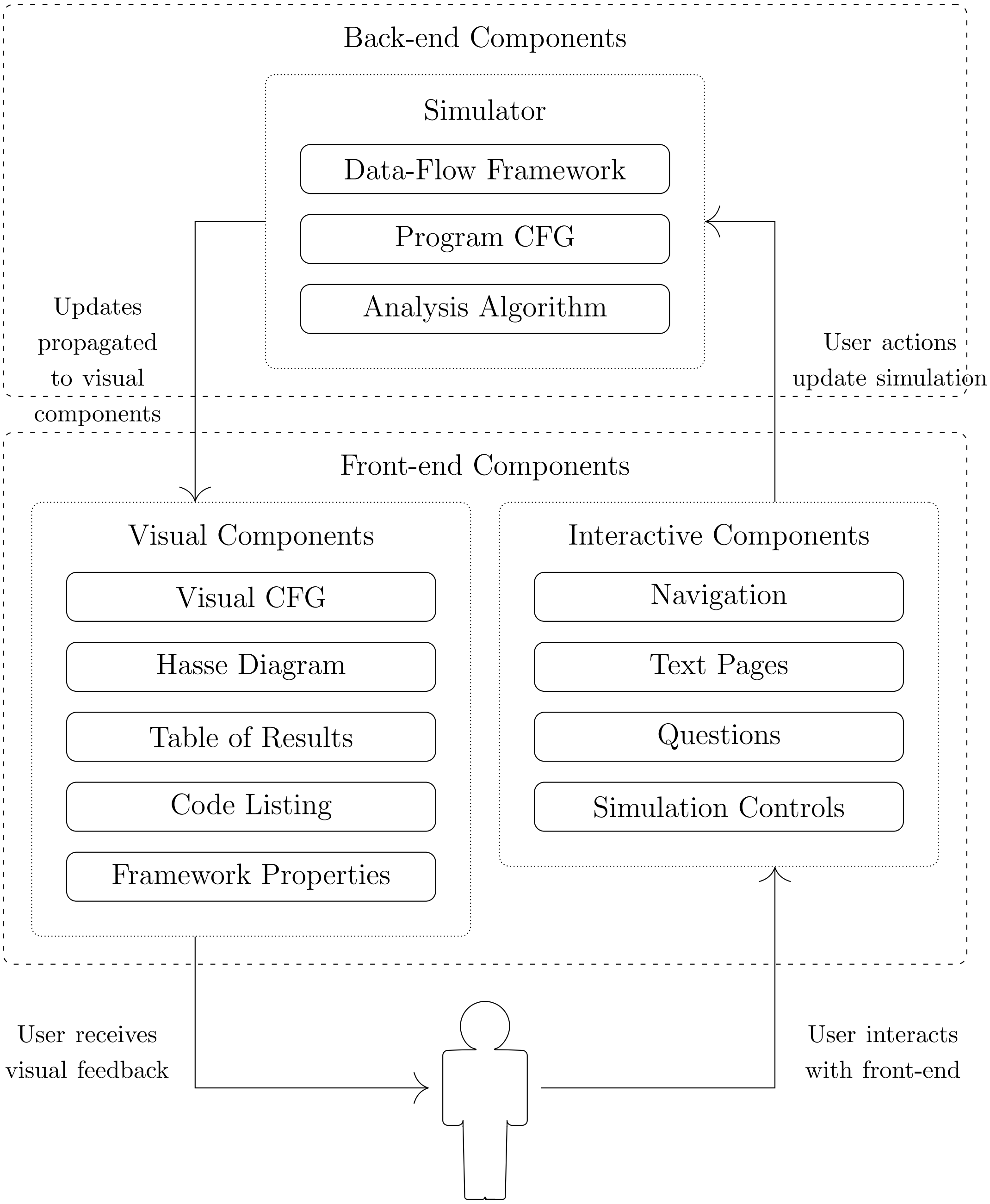


Fig. 4.1: An overview of the proposed system architecture.

Chapter 5

Back-End Implementation

This chapter discusses the implementation of the data-flow analysis simulation – the back-end elements which control the logic and state of the analysis, which is then visualised by the user interface components described in chapter 6.

5.1 Overview

The implementation of the simulator is based upon the concept of a general framework as described in §2.4 of this report and §10.11 of the *Dragon Book*[3]. The simulator takes as input a program written in the ILOC (see §5.5) language, a data-flow framework and an order in which to evaluate nodes. It parses the program and produces an AST, then uses this to build a CFG. The data-flow may then be simulated on said graph using the round-robin fixed-point algorithm for general frameworks (algorithm 2.3).

Playback of the simulation can be controlled through the simulator's API. Function calls which update the state of the simulation trigger an event handling system to update the visual components. However, the simulation is entirely independent of the rest of the system; re-using it in another project is as simple as copying the directory containing the simulator code and including the required libraries.

The simulator is implemented entirely in JavaScript. Snippets of code taken from external sources are explicitly commented as such in the source code and will be mentioned here.

5.2 Value Sets

JavaScript includes a native `Set` data structure for representing collections of unique objects. Unfortunately, two objects are only seen as equal if they refer to the exact same instance. For the purposes of this simulation it would be useful to compare objects which share some attributes but refer to different instances; for example, when operating on sets of operands it would be useful to consider two operands of different instructions as the same if they refer to the same variable.

For this reason the simulator uses a new `ValueSet` data structure, backed by a native JavaScript `array`. The `ValueSet` stores objects which inherit from a `ValueMixin`[1] and must define a `compare` function. Objects are checked for equality using this function so that two objects may be considered equal based on selected attributes.

`ValueSet`s support the following operations, listed here along with their estimated worst-case time complexities:

`s.size()`	`s.add(v)`	`s.delete(v)`	`s.has(v)`
$O(1)$	$O(\|s\|)$	$O(\|s\|)$	$O(\|s\|)$

`s.union(t)`	`s.intersect(t)`	`s.difference(t)`
$O(\|s\| \cdot \|t\|)$	$O(\|s\| \cdot \|t\|)$	$O(\|s\| \cdot \|t\|)$

Using an alternate backing structure such as a hash table would provide constant-time implementations of `has`, `add` and `delete`, and thus also improve the run-time of operations which make use of these functions (`union`, `intersect` and `difference`). Given that the expected user input will only operate on small `ValueSet`s this is not a major concern, but a potential improvement that should be noted for future development.

5.3 Data-Flow Frameworks

Each data-flow framework defines the elements outlined in §2.4, namely:

- The *domain* of values on which to operate;
- The *direction* in which data flows;
- A set of *data-flow equations* including the *meet operator* $\wedge$ and the set of *transfer functions* F.
- The *boundary* value specifying the initial value at the entry or exit to the CFG; and
- The *initial value*, $\top$, at each point in the graph.

The meet and transfer functions operate on `ValueSets`. Frameworks must also specify the following additional information:

- A name and identifier;
- LaTeX representations of the meet and transfer functions; and

[1] A mixin class contains methods which may be inherited by another class, but is not necessarily the parent of that class.

- JavaScript functions to compute local information independent from the transfer function.

This allows properties of the framework to be displayed by the visual components in addition to the values calculated at each point.

5.3.1 Included Data-Flows

As stated in section 4.2.2, the system includes three pre-defined data-flows: reaching definitions, liveness analysis and available expressions. These data-flows may be swapped in and out at any time, as the simulation has been developed independent of any framework. The system is capable of supporting bit-vector data-flows and could easily be extended to support tuple-valued data-flows, but none are available at present.

The three data-flow frameworks operate on programs written in ILOC (see §5.5). ILOC programs are able to write to three types of storage: registers, main memory and the comparison register. To allow users to translate examples in other languages or pseudocode into ILOC and obtain the same solution, these data-flows have been defined to operate solely on registers.

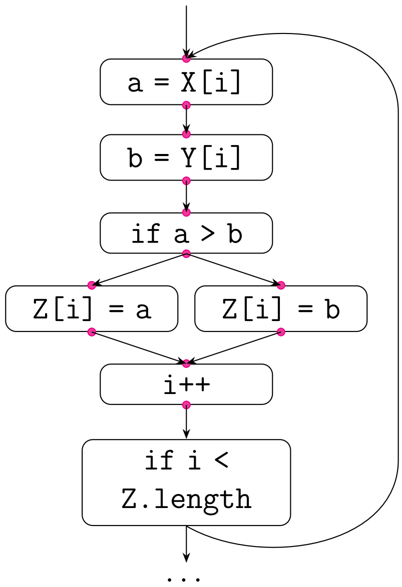


Fig. 5.1: A control-flow graph.

For example, consider the CFG in fig. 5.1. The branching statement, `if a > b`, only uses variables – there are no assignments involved. However, if we were to represent this in ILOC:

```
cmp_GT ra, rb => rc
cbr    rc     -> LEFT, RIGHT
```

The comparison and branch must be written as two separate statements. In the ILOC program above, the comparison is an assignment to `rc` – but, taking the example of reaching definitions, we would obtain a different solution than expected in fig. 5.1 because that program does not assign the result of the comparison. However, in the following program:

```
comp   ra, rb => cc
cbr_GT cc     -> LEFT, RIGHT
```

The comparison assigns a value to `cc`, the comparison register. By ignoring this memory type in our data-flows, we have solved the problem and made our simulation much more flexible.

5.4 Simulation Algorithm

Algorithm 5.1 shows a disassembled version of algorithm 2.3. The functions can be combined to enable a step-by-step evaluation or automatic playback.

Algorithm 5.1: Implementation of General Framework Algorithm

```
function reset()
    B = entry node of CFG
    B.meet = framework.boundary
    for each (node in the cfg)
        node.meet = framework.top
    step    = MEET
    changed = false

function iterate()
    if (step is MEET)
        B.meet, changed = framework.meet(b, this.cfg)
        step = TRANSFER
    else
        B.transfer, changed = framework.transfer(b)
        step = MEET
        B    = next node
```

To demonstrate, listings and 5.1 and 5.2 show the implementation of some of the playback functions. In the fast forward function, the simulation is advanced until it is complete before triggering an update event which is propagated to other components of the system. In the automatic playback function, a function is called at set time intervals which advances the simulation by one iteration, then triggers the update. Controlling when update events are triggered events helps prevent slowdown from unnecessary re-rendering of components.

Listing 5.1: JavaScript Implementation of Fast Forward

```
this.fast_forward = function() {
  while(!this.state.finished) {
    this.iterate();
  }
  this.events.trigger('update');
}
      \vspace{-5mm}
```

Listing 5.2: JavaScript Implementation of Automatic Playback

```
this.play = function() {
  this.state.paused = false;
  var _this = this;
  (function foo() {
    if (!_this.state.paused && !_this.state.finished) {
      // If the user hasn't pressed pause and we're not finished
      _this.iterate();                       // Step forward
      _this.events.trigger('update');       // Update components
      setTimeout(foo, _this.play_speed); // Repeat at interval
    }
  })();
}
```

5.5 Understanding Programs

The simulation operates on programs written in ILOC, as described in *Engineering a Compiler*[2, appx. A]. ILOC is an assembly-like language designed for toy compilers, in that it is both simple to parse and human-readable. This makes it the perfect input language for the learning platform, with the added benefit that students will be able to step through examples from the textbook using the simulator.

The simulator supports the following ILOC features:

- The entire set of opcodes listed in *Engineering a Compiler*;
- Three types of literal: registers, labels, and integers;
- Simple programs with a single operation per instruction; and
- Distinction between three types of memory: registers, main memory, and the comparison register.

There is also limited support for multi-operation instructions, which may be used to emulate data-flow analysis on basic blocks. This functionality remains untested as priority was assigned to simulation on nodes as single instructions.

In order to understand ILOC programs, they must be to converted from text into a CFG. The first stage in this process is parsing the text. The PEG.js[13] was used to generate a parser (written in JavaScript) from an input grammar. This parser produces an abstract syntax tree (AST), which is then transformed into a CFG representing the program. The following sections describe this process in more detail.

5.5.1 Parsing Expression Grammar

A Parsing Expression Grammar (PEG) is very similar to a context-free grammar in that it describes a formal language using a set of rules for recognising strings in that language. The description of ILOC in *Engineering a Compiler* describes a simple PEG for the language, which has been adapted for use in this system.

Some rules in the grammar have been extended as shown in fig. 5.2. The reason for these changes is that ILOC allows the user to write to three types of storage: registers, main memory and comparison flags. Changing the grammar allows disambiguation in the abstract syntax tree between operations which write to or read from different types of storage. This is an important distinction to make because the built-in data-flows only consider register accesses when calculating value sets at each point (see §5.3.1 for an explanation).

$$
\begin{array}{rcllcl}
Operation & \rightarrow & NormalOp & & & \\
 & | & ControlFlowOp & & & \\
 & | & MemoryLoadOp & & & \\
 & | & MemoryStoreOp & & & \\
 & & & & & \\
MemoryLoadOp & \rightarrow & LoadOpcode & OperandList & \Rightarrow & OperandList \\
MemoryStoreOp & \rightarrow & StoreOpcode & OperandList & \Rightarrow & OperandList \\
NormalOp & \rightarrow & NormalOpcode & OperandList & \Rightarrow & OperandList \\
ControlFlowOp & \rightarrow & ControlFlowOpcode & OperandList & \Rightarrow & OperandList \\
 & & & & & \\
LoadOpcode & \rightarrow & \texttt{loadI} & & & \\
 & | & \texttt{loadAO} & & & \\
 & | & \ldots & & & \\
 & & & & & \\
StoreOpcode & \rightarrow & \texttt{storeI} & & & \\
 & | & \texttt{storeAO} & & & \\
 & | & \ldots & & & \\
 & & & & & \\
NormalOpcode & \rightarrow & \texttt{addI} & & & \\
 & | & \texttt{rshift} & & & \\
 & | & \ldots & & & \\
 & & & & & \\
ControlFlowOpcode & \rightarrow & \texttt{jumpI} & & & \\
 & | & \texttt{cbr\_GE} & & & \\
 & | & \ldots & & & \\
 & & & & & \\
Operand & \rightarrow & register & & & \\
 & | & num & & & \\
 & | & label & & & \\
 & | & cc & & & \\
 & & & & & \\
cc & \rightarrow & \texttt{cc} & & &
\end{array}
$$

Fig. 5.2: Extensions to ILOC Parsing Expression Grammar

5.5.2 Abstract Syntax Tree

A parse tree represents the exact structure of parsed data. In contrast, an AST represents the structure of data independent of its original representation; this may involve adding or removing nodes or re-structuring the tree entirely.

To construct an AST one would traditionally obtain a parse tree and then transform it into the desired structure. In contrast, parsers generated by the PEG.js library are capable of constructing the AST as they parse the input program. The developer may supply JavaScript code to be executed when a rule matches and this code is used to create an abstract representation of the data matched by that rule. This feature can also be used to annotate the tree as it is built.

Listing 5.3 shows a snippet of the PEG.js grammar for ILOC. Groups of symbols may be assigned an identifier and referred to in the JavaScript code; on line 9

Listing 5.3: Example of PEG.js Grammar Rules

```
1  Operand
2      = _ r:register _ { return r; }
3      / _ n:num _ { return n; }
4      / _ c:cc _ { return c; }
5      / _ l:label _ { return l; }
6
7
8  register
9      = _ "r" n:([0-9a-z_]i)+ _ {
10         return new ILOC.Operand({
11             type: ILOC.OPERAND_TYPES.register,
12             name: n.join("")
13         });
14     }
```

the identifier `n` is given to the register name so it can be set as the name of the returned `Operand`. The return value of the JavaScript code is passed to the above context; the identifier `r` on line 2 refers to the `Operand` returned by the `register` rule. More complex code may collect information and use it to annotate the AST; for example, the grammar used in this project annotates assignments to a given register with a unique identifier for use in data-flows such as reaching definitions.

Fig. 5.3 shows an example AST for the simple ILOC program in listing 5.4.

Listing 5.4: Simple ILOC Program

```
Start: addI    ra, 1  => rb
       comp    ra, rb => cc
       cbr_LE  cc     -> Start, Store
Store: storeAO ra     => rx, 32
```

5.5.3 Control-Flow Graph

The simulation constructs a CFG from the AST. Internally, the edges of the CFG are represented using an adjacency matrix. As the input is expected to be small the choice of internal representation has minimal effect on the efficiency of the simulation. If larger input was expected then a strong case could be made for switching to a different structure such as an adjacency list in order to reduce memory requirements; however, the CFG shares much of its underlying code with the Hasse diagrams (see §5.5.4) for which an adjacency matrix is better suited.

Algorithm 5.2 describes how the CFG is constructed from the AST. The algorithm first adds all of the instructions to the graph, identifying which labels refer to which instructions and storing this information in a map. Next, the algorithm populates the adjacency matrix. If the node is a ControlFlowOperation an edge to the target instructions (looked up using the label map), otherwise we add an edge to the next instruction in the sequence.

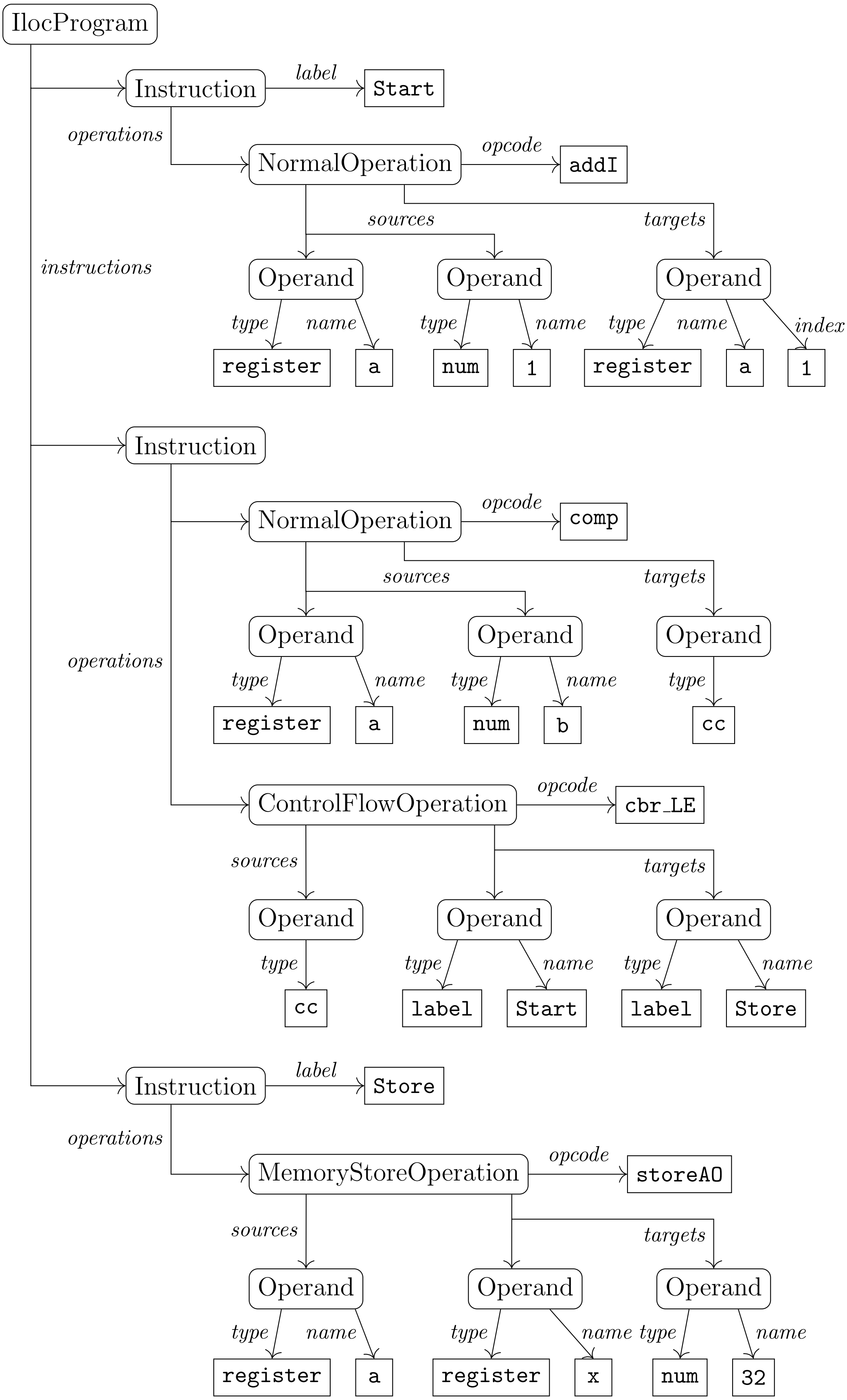


Fig. 5.3: The AST for the ILOC program in fig. 5.4.

Algorithm 5.2: Constructing a CFG from an AST for an ILOC program.

```
labels = Map (Label → Instruction)

for each (instruction in the AST)
    add instruction to the CFG as a node
    if (instruction has label)
        add (label → instruction) to labels

for each (node in the CFG)
    if (node is a ControlFlowOperation)
        for each (target in node)
            add an edge from node to labels[target]
    else
        add an edge from node to next node
```

5.5.4 Lattices

The simulation uses the CFG to construct the meet semi-lattice for a given data-flow framework. The lattices in this simulation only provide support for bit-vector data-flows since these are the only type included with the system (see §5.3.1).

Due to this constraint, the process of generating the lattice is quite simple. The set of all combinations of values is a powerset, and our lattice must include each element of this set. This is easy to see when we consider the meaning of the term bit-vector: the number of possible sets represented by n bits is 2^n, and each possible set is present in the lattice.

To generate the lattice for a simulation, the domain of values is first identified. Then, all possible sets of values must be generated (the code for which was taken from a snippet online[14]). Each value set becomes a node in the lattice. Next, the algorithm finds the meet of every pair of sets; an edge is added from the sets to the result of the meet operation. Finally, a transitive reduction is performed on the graph: if an edge exists from $x \to y$ and from $y \to z$, any edge from $x \to z$ may be removed as it is represented by the path $x \to y \to z$. This constructs a Hasse diagram representing the meet semi-lattice, as described in algorithm 5.3.

The simulator only generates a lattice for programs with a small set of values due to the exponential growth of the size of the lattice. The graph is backed by an adjacency matrix since this structure has constant-time operations for adding and removing edges, resulting in a huge performance increase over other structures such as an adjacency list.

Algorithm 5.3: Constructing a Hasse diagram for the meet semi-lattice of a CFG.

```
values = collect all values from the CFG
sets = generate all possible subsets of values

graph = Graph

for each (set in sets):
    add set to graph as a node

// Find all the edges between the nodes
for i from 0 to length(sets) − 1
    for j from i + 1 to length(sets)
        temp = meet(sets[i], sets[j])
        add edge to graph from sets[i] to temp
        add edge to graph from sets[j] to temp

// Perform a transitive reduction of the graph
for i from 0 to length(sets)
    for j from 0 to length(sets)
        if (there exists an edge i → j)
            for k from 0 to length(sets)
                if (there exists an edge j → k)
                    remove the edge i → k
```

5.6 Summary

In summary, the implementation of the simulation is sound. Whilst better choices could have been made regarding the data structures used, the modular system simplifies the addition of data-flows and enables the back-end to be extracted for use in other projects. The simulation makes relatively little use of external libraries[2] making it fairly lightweight and simple to understand. In addition, the extensions to the ILOC grammar make the language flexible enough to model programs from other languages; students will hopefully be able to use this feature to check their solutions to worked problems.

Reliance on techniques described in the *Dragon Book*[3] has deepened my understanding of the material, as the book proved invaluable for guidance throughout the implementation of the back-end. In future, I would be interested to see extensions to the simulation to support different methods for solving data-flow problems described by the book such as the structural or worklist algorithms, and perhaps adapt and improve my implementation for use in similar projects.

[2]Beside PEG.js, necessary for the parser.

Chapter 6

Front-End Implementation

This chapter discusses the implementation of the front-end elements which make up the user interface, including the visualisation and interactive teaching components.

6.1 Overview

The implementation of the user interface components uses a modular design. Each element is a `View`, a component which renders content to a HTML element referred to as that component's canvas. An inheritance model is used so that functionality is shared between `Views`, for example, each of the visual components inherits from `SimulatorView`, which handles registering callback functions with a simulator's event handler.

The front-end is implemented using JavaScript, HTML and CSS. A number of libraries provide additional functionality, including:

- d3.js and the extension dagre-d3 to draw directed graphs;
- jQuery and extensions such as tipsy for navigation and visual elements;
- MathJax for rendering of LaTeX math formulae;
- Handlebars.js for HTML templating; and
- Bootstrap v4 alpha for page layout and theming.

6.2 View Model

Each user interface component is implemented as its own class, or `View`. A `View` component renders HTML content inside another HTML element referred to as its canvas. Multiple `Views` may exist at one time, and a `View` may contain other views.

Beside from the individual view classes, there are multiple types of `View`; for example, all of the visual components inherit from the `SimulatorView` class. Each `SimulatorView` is required to implement callback functions for the `update` and `reset` events. These functions are called by the simulator's event handler when an event is triggered, which may cause the component to be re-rendered or the information contained within to be modified.

In addition to specifying the interface for various classes, the `View` model allows functionality to be shared between components. The `TutorialView` class implements navigation through and display an interactive tutorial, so that each one need only implement functions to control the content shown at each step.

This model is a very simple implementation of what is known as a single page app: a web application in which the page is only loaded once, after which JavaScript is used to update the page content and asynchronous calls request more data from the server. Some existing libraries are based around this concept; for example AngularJS[15] provides a similar framework for creating single page apps using re-usable components.

6.3 Visual Components

This section describes the implementation of the components which visualise the simulation, including the choice of any libraries and frameworks and the justification of design decisions.

6.3.1 Control-Flow Graphs

Control-flow graphs are rendered using the dagre-d3 extension for the graph visualisation library d3.js. An SVG component is created and the nodes and edges added to the dagre-d3 graph. The layout of the graph is handled entirely by the graphing library.

The dagre-d3 library was chosen due to the strength of the demonstrations on the project's website[16]. The available examples showed graphs very similar to the desired result, with only a small set of API calls required to draw a simple control-flow graph. Given more time, perhaps using a more fully-featured graphing library such as Cytoscape.js[17] or vis.js[18] would improve the aesthetics and performance of the component. However, dagre-d3 is more than adequate for a proof of concept.

To reduce clutter in the graph, the user has three options for displaying nodes: *current*, in which only the points relating to the current step are shown; *all*, in which every point is displayed; or *none*, in which only the nodes of the CFG are shown. Fig. 6.1 shows the CFG in the *current* setting.

A system was devised to allow dynamic insertion and removal of points from the CFG. In addition to automatically displaying nodes based on the settings described above, the component's API can be used to add or remove specific points. This allows the tutorial examples to display the exact information required rather than relying on the simulation to process the desired nodes.

As the user steps through a simulation, nodes and points in the CFG are highlighted to visually relate them to actions occurring in other components. This is shown by fig. 6.1; the transfer function is reading information, shown in light blue, from In(n_7) and modifying Out(n_7), shown in dark blue.

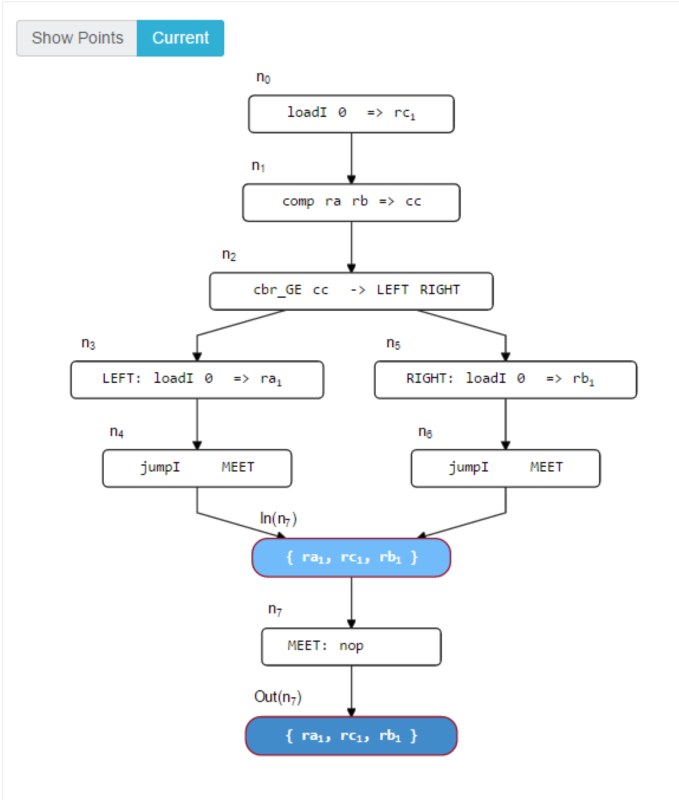


Fig. 6.1: Control-Flow Graph Visualisation

6.3.2 Hasse Diagrams

The Hasse diagram component is implemented using much the same system as the control-flow graphs. An SVG component is created, the nodes and edges are added to a dagre-d3 graph and the result is rendered to the screen.

As the user steps through a simulation, nodes in the diagram are highlighted to visually relate them to actions occurring in other components – during the meet phase, the sets being considered are highlighted in light green and the resulting set in dark green.

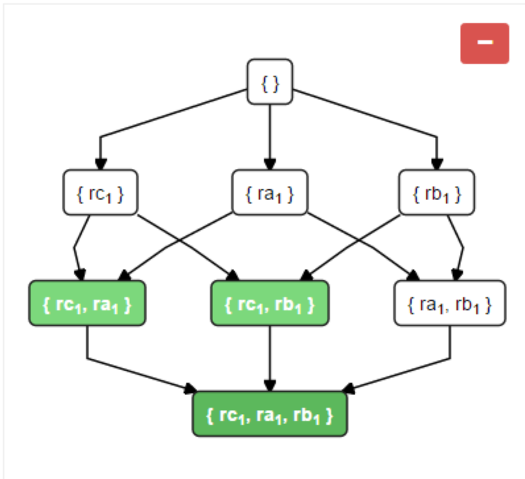


Fig. 6.2: Hasse Diagram Visualisation

6.3.3 Table of Results

The table of results, which displays the set of values at each point, is simply a HTML table containing the required information. Various layouts were considered for this table, with the final design shown in fig. 6.3.

	Local Information		Global Information				
Instruction	defgen	defkill	Round	1		2	
			Set	In	Out	In	Out
0	$\{ra_1\}$	$\{ra_1\}$		$\{\}$	$\{ra_1\}$	$\{\}$	$\{ra_1\}$
1	$\{rb_1\}$	$\{rb_2, rb_1\}$		$\{\}$	$\{rb_1\}$	$\{\}$	$\{rb_1\}$
2	$\{rc_1\}$	$\{rc_2, rc_1\}$		$\{\}$	$\{rc_1\}$	$\{\}$	$\{rc_1\}$
3	$\{rc_2\}$	$\{rc_1, rc_2\}$		$\{\}$	$\{rc_2\}$	$\{\}$	$\{rc_2\}$
4	$\{rb_2\}$	$\{rb_1, rb_2\}$		$\{\}$	$\{rb_2\}$	$\{rc_2\}$	$\{rb_2\}$
5	$\{rd_1\}$	$\{rd_1\}$		$\{\}$	$\{rd_1\}$	$\{rb_2\}$	$\{rd_1, rb_2\}$
6	$\{\}$	$\{\}$		$\{\}$	$\{\}$	$\{rd_1\}$	$\{rd_1\}$
7	$\{\}$	$\{\}$		$\{\}$	$\{\}$	$\{\}$	$\{\}$

Fig. 6.3: Current Table Layout

Cells in the table are highlighted to link them to other visual components. Any sets are read or modified, including local information, are highlighted in the table in each step of the simulation.

This design could be improved by changing the layout of the table when viewed on small screen sizes. In the current layout the entire table becomes scrollable if it is too large for its canvas. Perhaps changing the table so that only the rounds become scrollable (see fig. 6.4) would improve its readability; however, the amount of time required to implement this was deemed better spent on implementing other functionality as the cases in which it is required are uncommon.

	Local Information								
Instruction	defgen	defkill	Round	3		2		1	
			Set	In	Out	In	Out	In	Out
0	$\{ra_1\}$	$\{ra_1\}$		$\{\}$	$\{ra_1\}$	$\{\}$	$\{ra_1\}$	$\{\}$	$\{ra_1\}$
1	$\{rb_1\}$	$\{rb_2, rb_1\}$		$\{ra_1\}$	$\{rb_1, ra_1\}$	$\{ra_1\}$	$\{rb_1, ra_1\}$	$\{\}$	$\{rb_1\}$
2	$\{rc_1\}$	$\{rc_2, rc_1\}$		$\{rb_1, ra_1\}$	$\{rc_1, rb_1, ra_1\}$	$\{rb_1\}$	$\{rc_1, rb_1\}$	$\{\}$	$\{rc_1\}$
3	$\{rc_2\}$	$\{rc_1, rc_2\}$		$\{rc_1, rb_1, rd_1, rb_2\}$	$\{rc_2, rb_1, rd_1, rb_2\}$	$\{rc_1, rd_1\}$	$\{rc_2, rd_1\}$	$\{\}$	$\{rc_2\}$
4	$\{rb_2\}$	$\{rb_1, rb_2\}$		$\{rc_2, rd_1\}$	$\{rb_2, rc_2, rd_1\}$	$\{rc_2\}$	$\{rb_2, rc_2\}$	$\{\}$	$\{rb_2\}$
5	$\{rd_1\}$	$\{rd_1\}$		$\{rb_2, rc_2\}$	$\{rd_1, rb_2, rc_2\}$	$\{rb_2\}$	$\{rd_1, rb_2\}$	$\{\}$	$\{rd_1\}$
6	$\{\}$	$\{\}$		$\{rd_1, rb_2\}$	$\{rd_1, rb_2\}$	$\{rd_1\}$	$\{rd_1\}$	$\{\}$	$\{\}$
7	$\{\}$	$\{\}$		$\{rd_1\}$	$\{rd_1\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$

Fig. 6.4: Alternate Table Layout with Scrollable Rounds

6.3.4 Code Display

The code display handles displaying ILOC programs, with visual links to other components sharing the simulation. This is handled by a simple HTML table, as each ILOC instruction has a maximum of 7 fields: label, opcode, two sources, the $\rightarrow$ or $\Rightarrow$ symbol, and two targets. Aligning each token in the instruction allows the code to be read much more easily than if the raw text were displayed on screen. Fig. 6.5 shows the code display component.

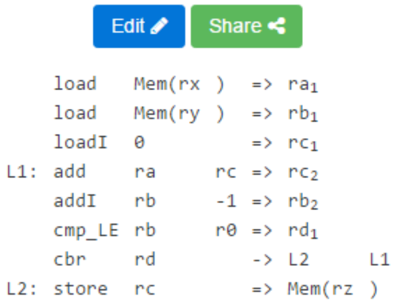


Fig. 6.5: Default Code Display

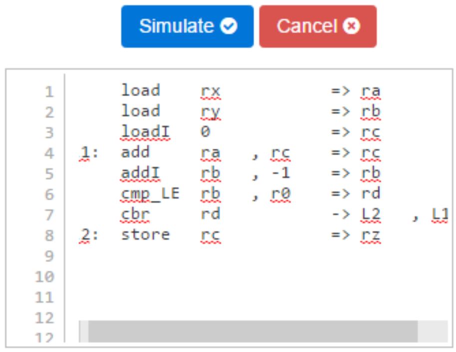


Fig. 6.6: Editing Code in the Simulator

Fig. 6.7: Share Code Dialog

In the simulator interface this view also handles editing or sharing code with others. Figures 6.6 & 6.7 show this in action. If the user enters invalid code an error message is displayed using line and column information obtained from PEG.js. Line numbers are displayed in the text area to help the user identify the source of the error; this functionality is provided by the jQuery Lined TextArea[19] plugin. The additional whitespace is added by the simulator to improve readability, but is not necessary.

6.4 Interactive Components

This section describes the implementation of the components which enable the user to interact with the system, including the choice of any libraries and frameworks and the justification of design decisions.

6.4.1 Simulator Controls

The simulator controls simply activate the playback functions in the simulator through its API. The controls can be embedded in any other view, so whilst their main use is in the simulator interface it is also possible to include them in interactive tutorials or tests.

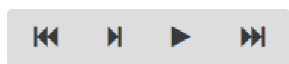


Fig. 6.8: Simulator Controls

6.4.2 Questions

The `QuestionView` class was designed to be adapted to as many situations as possible. Each question displays a number of possible answers, and has a variety of configuration options:

- The order in which answers should be presented (or shuffled).
- Whether a question allows multiple answer choices.
- Which answers should be revealed upon selecting an answer, if any.
- Which answers should be disabled upon selecting an answer, if any.

The question and its answers are rendered using the MathJax library to display mathematical formulae. Submitting a question reveals the nature of each answer (correct or incorrect) and marks those selected by the user. Answers may be assigned flavour text which is displayed upon selection to provide the user with instant, detailed feedback. Two example questions are shown in figures 6.9 & 6.10

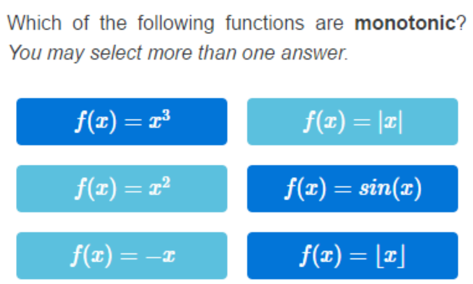


Fig. 6.9: A multiple-choice `QuestionView`.

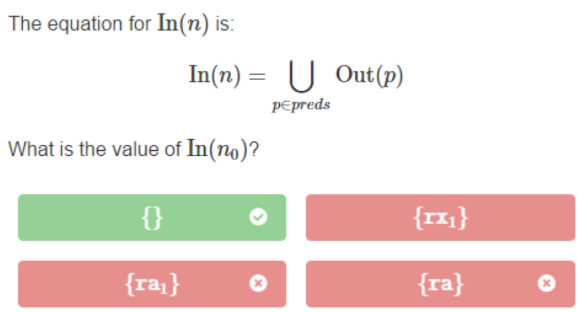


Fig. 6.10: A `QuestionView` after it has been submitted.

The highly configurable nature of the questions allows them to be re-used across the app. In `TutorialViews` the answers are revealed immediately, whereas in `TestViews` the test must be submitted before the user is given feedback. `QuestionViews` are also used in the evaluation to collect categorization information from the user.

Two callback functions can be passed to the `QuestionView` to control the application's behaviour upon selecting an answer. This function can perform such actions as allowing the user to proceed by enabling navigation buttons, or altering neighbouring components to provide visual feedback.

6.5 User Interface

This section describes the implementation of the user interface, including the choice of any libraries and frameworks and the justification of design decisions.

6.5.1 Overview

The user interface, much like the visual components, is implemented using the `View` model. The modular nature of the `View` system means that it is easy to add or remove components or sections of the user interface (UI). In the same way that the simulator and its visual components may be extracted from the rest of the system, the interactive learning components are independent of the data-flow analysis content. This would allow others to implement a similar system for other topics with only a small amount of work.

A combination of the Bootstrap design framework, jQuery, Handlebars and MathJax libraries were used to produce the overall look and feel of the UI. The latest development version of Bootstrap (v4-alpha-2) was used due to the addition of CSS `flexbox` support – whereas the current version (3.5.6) uses `float`ing elements, `flexbox` allows more control over the layout including more precise vertical alignment and scaling elements to fill their containers. Whilst similar layouts are possible in 3.5.6, they are often difficult or painful to achieve consistent behaviour across different web browsers.

However, this increase in flexibility comes with some drawbacks. Load times of the application are reduced; though most of the JavaScript and CSS libraries are hosted using a content delivery network (CDN), less common libraries such as this project's build of the Bootstrap alpha must be downloaded from the application's server. A CDN would provide faster download speeds and enable the use of cached files if the user has visited other sites using the same libraries, whereas hosting them on the application server requires those libraries to be downloaded regardless of whether the user has already received them from another source.

6.5.2 Menu

The main menu is the first screen the user sees upon loading the app. The final design consists of a list of buttons which take the user to the corresponding view. The buttons contain small icons to indicate whether a user has completed a lesson or test.

One improvement would be to include a description panel in order to summarise the content of the lessons or simulation. Re-ordering the menu items such that the introductory lessons are at the top of the list would draw more attention to them and hopefully provide some guidance upon opening the application for the first time. Finally, adding some visual cues such as images or icons relating to the

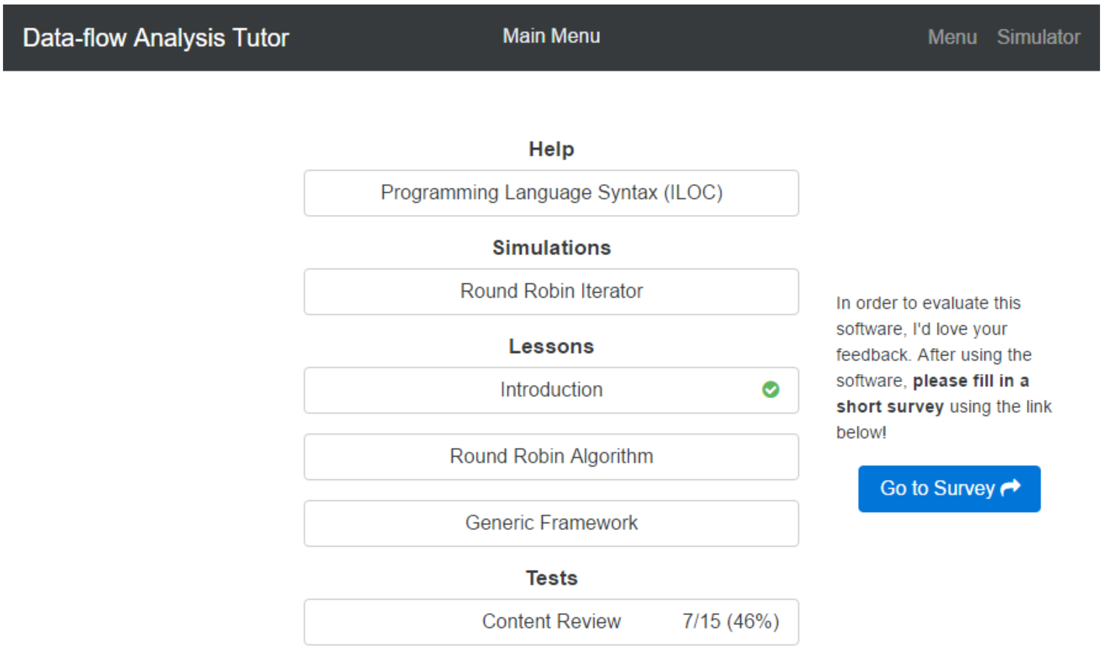


Fig. 6.11: The main menu of the application.

content of each item would help the user quickly navigate the menu and break the monotony of the current design.

6.5.3 Simulation Interface

The final simulation interface is shown in fig. 6.12. The design is very similar to early concepts, but changes had to be made to accommodate different screen sizes:

- The CFG and table of results have been moved into separate tabs, as there was not enough room to display both simultaneously on small displays. This saves screen real-estate where it is needed, but wastes it where it is not – the change could be improved by changing the layout to a side-by-side view on larger displays using JavaScript.
- The Hasse diagram component is collapsible, which increases the area available to the code display when simulating programs with many instructions. In cases where it is too large to be displayed it is replaced by a warning message.
- The simulation settings have been moved to a separate window. When the user clicks the "Settings" button is clicked a modal dialog appears, prompting them to change the configuration of the simulation. These settings were previously beside the data-flow framework properties.

These features increase the usability of the design for users of laptops or similarly sized devices. Other features of the interface include the "Share" button (detailed in §6.3.4) to allow students to share their programs and simulations with classmates or their professor, and the framework details which show the list of nodes in addition to the ordering in use. The use of colour and familiar icons draw the user to important features and clearly indicate the function of buttons and menus.

An alternate design considered during early development proposed the use of a window system, which would allow the components to be dynamically rearranged or resized. However, the simulation interface lays out all of the available visual components in such a way that the maximum amount of information is conveyed without being too confusing or intimidating. The time which would have been spent developing the window system would likely not have been worth the effort (even with the availability of external libraries) given the strength of the implemented design.

6.5.4 Tutorial Interface

In the tutorial interface, the user follows a series of steps which present them with text descriptions and sometimes visual components. These steps are written in JavaScript and so are able to hook into the component or simulator APIs in order to manipulate them; for example, in fig. 6.14 when the user selects the correct answer they are shown visual confirmation in the CFG and table of results.

The steps-as-functions system gives a lot of power to the developer at the cost of added complexity. Whilst this is great for complicated steps involving questions which trigger simulator events or manipulate visual components, many of the steps are simply a passage of text with some formulae. If the project were to be developed further or extended to other topics it would be useful to implement some kind of high-level description language for tutorials. Simple operations such as displaying text or a `QuestionView` or advancing the simulation, could be written quite simply in this format. This follows the tried-and-true software engineering principle Don't Repeat Yourself (DRY).

The current step implementation also has an unfortunate side-effect – each step is dependent on the ones before it, so to navigate to a previous step the system must execute each of the preceding step functions (for example, to go from step 38 to step 37 the system will reset and then execute steps 1 to 37). When navigating forward through a tutorial, some steps taking slightly longer to load is barely noticeable. However, when navigating backward it becomes glaringly obvious, due to the cumulative effect of executing every step at once.

The high-level description language mentioned above would enable some kind of state history system. Changes could be popped on and off a stack in order to navigate forward or backward, increasing the efficiency of navigation. This functionality would be abstracted away from the developer by the description language, removing any complexity faced when implementing this in the current system.

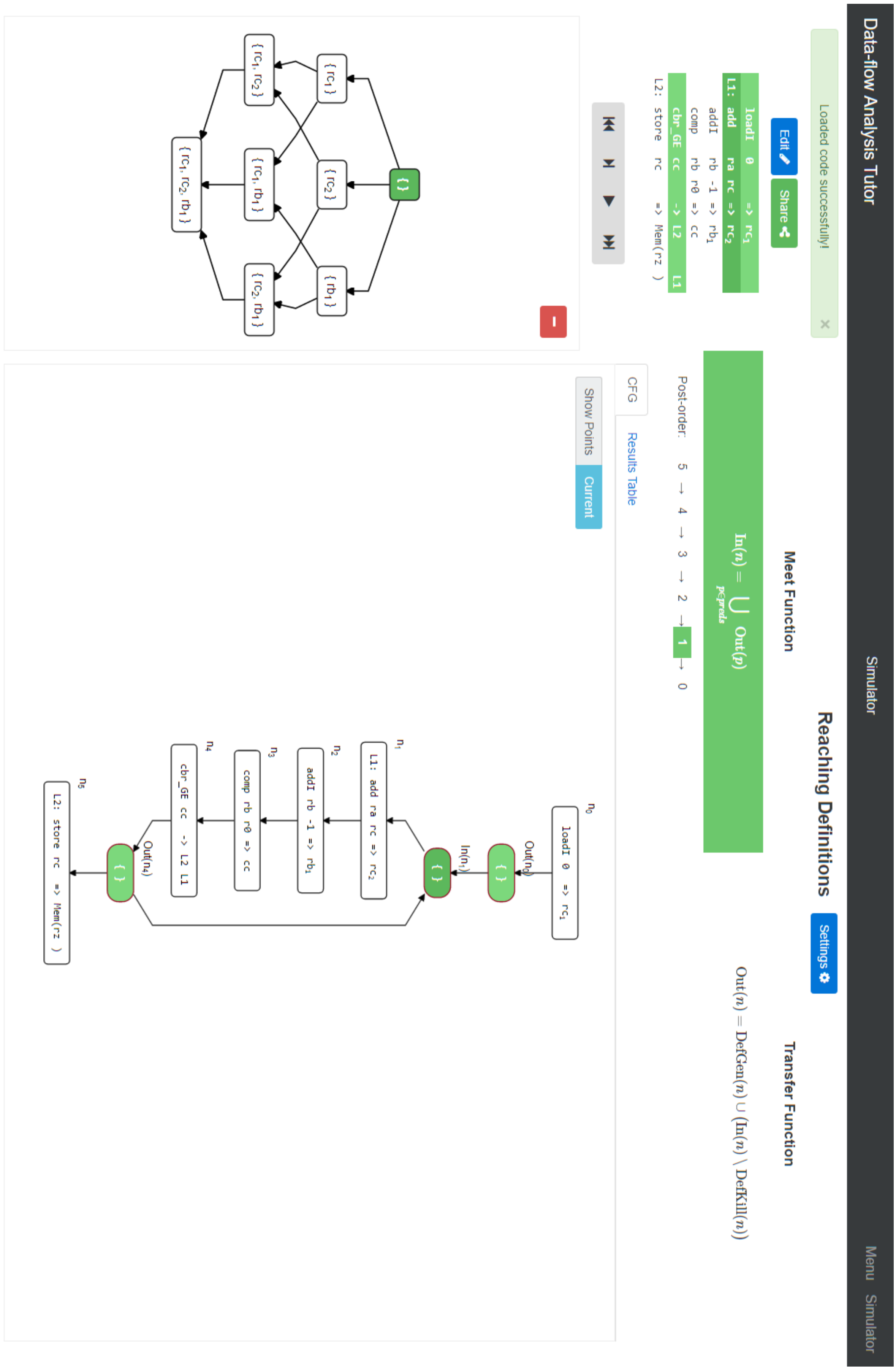


Fig. 6.12: The simulator interface.

6.5.5 Testing Interface

The testing interface was created for two reasons: first, to provide useful data in order to evaluate the project; second, to provide the user with a means of obtaining feedback about their learning progress.

The design and implementation is an extension of the interactive tutorials. However, instead of supplying the code for each step of the program, a developer may supply text questions and answers with an optional function display any other components (such as a CFG, as shown in fig 6.13). Unlike in the `TutorialViews`, navigating to the previous question does not require every question to be re-initialized as each question is self-contained.

Upon submission the user is shown some feedback and the correct answers are revealed, with icons to indicate which answers they picked to help them identify any mistakes they made. The feedback given is a simple overall score – perhaps it would be more useful to provide a score breakdown by topic, to indicate which areas they need to improve upon. Some simple usability tweaks such as a progress indicator or a dialog to confirm whether the user would like to submit the test would not go amiss.

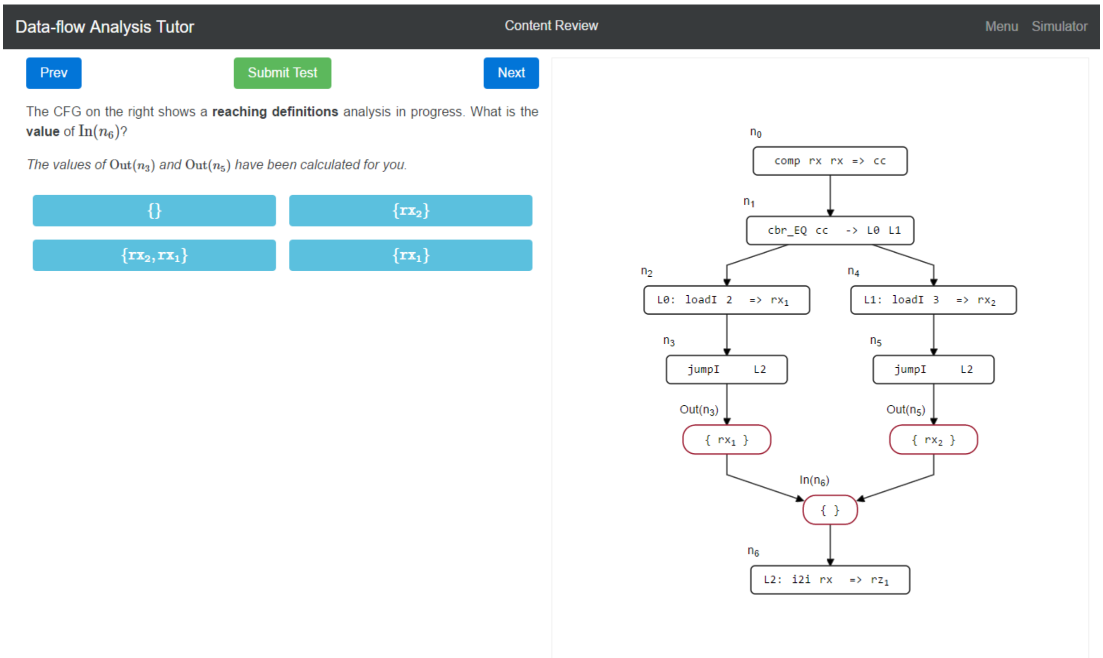


Fig. 6.13: An interactive test.

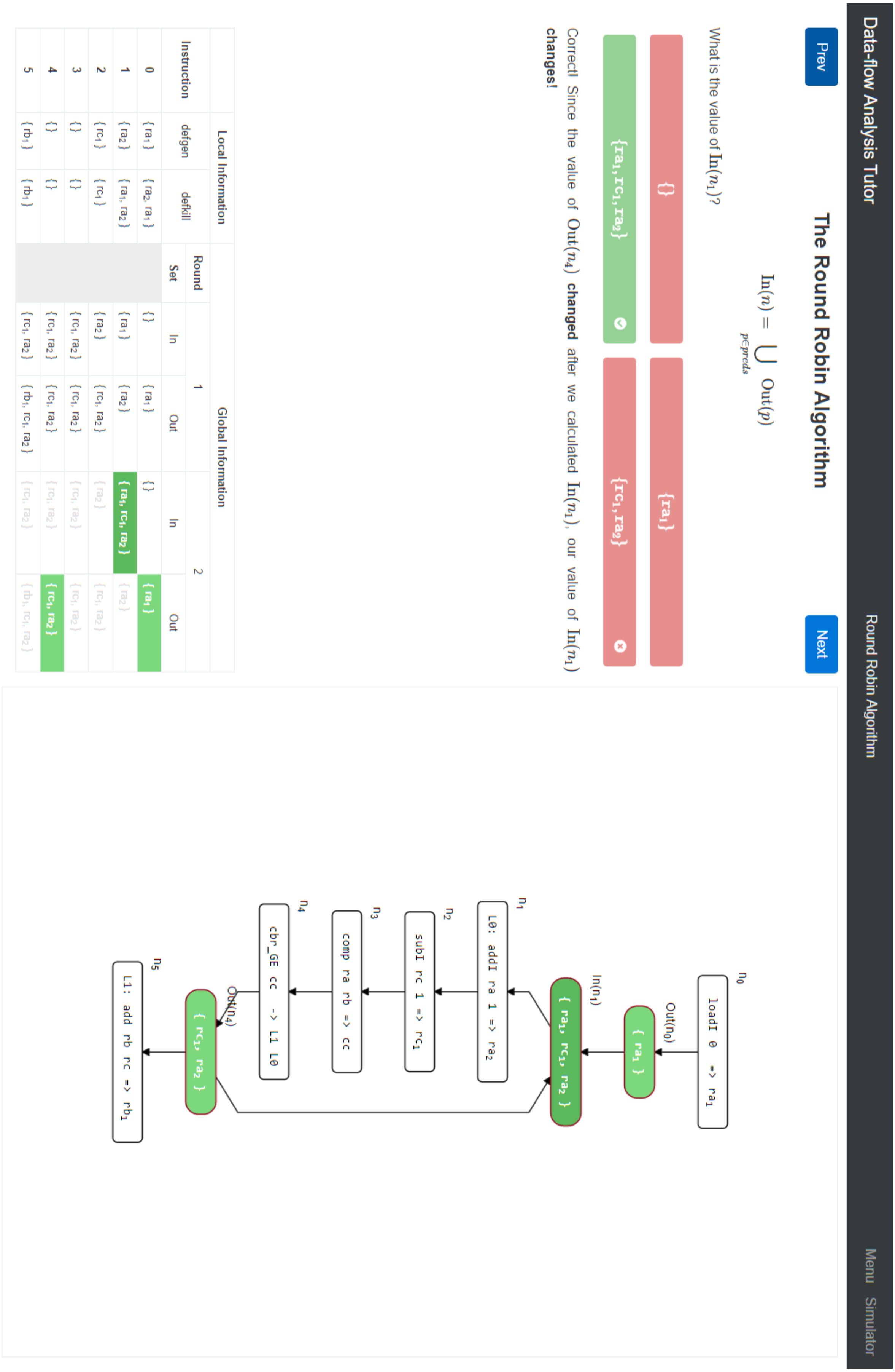


Fig. 6.14: An interactive tutorial.

6.6 Summary

In summary, the user interface is fairly well-designed but lacks polish. Some small features such as progress indicators were overlooked during development; the loss of these elements slightly reduces the usability of the interface.

However, the design has many positive aspects. The interface was optimised for display on smaller screens to make the application accessible to all students, and the added code sharing functionality should help students share examples with their classmates or professors in order to develop their learning through discussion.

The simulation interface visualises the content clearly and is certainly the highlight of the application's front-end. The choice of popular external libraries allows the system to be easily modified and extended by most JavaScript or web developers, in addition to increasing performance and security as stated in 4.2.2. The system's modular design allows its components to be re-used across the application, reducing the need to duplicate code and providing consistency throughout.

Even without the improvements mentioned throughout this chapter, the platform is fully-functional and should prove itself a valuable tool for learning. I hope that the techniques I have employed in developing a truly interactive and visual learning experience have set a good example for similar projects in future.

Chapter 7

Evaluation

This chapter describes the evaluation methodology for the project, followed by a critical analysis of the design and implementation of the system and suggestions for future development. The chapter concludes with an assessment of the success or failure of the project as a whole.

7.1 Introduction

The goal of this project was to create an interactive system to teach students the basic principles of data-flow analysis in compilers. The final system took the form the form of an online learning platform, containing a data-flow analysis simulator and a series of interactive tutorials.

To determine whether the final product meets the original aim of the project, there are three main points against which it should be evaluated:

- Interactions with the system, to analyse the users' engagement with the platform;
- Achievement of learning outcomes by the participants, in order to assess its validity as a learning platform; and
- Feedback from the participants, to determine whether it is a useful tool for the intended audience.

The following sections present a thorough evaluation of the system against these three objectives.

7.2 Method

The intended audience of the project is primarily students of the COPT course. However, the number of students available is limited; the course has around 30 registered students and anecdotal evidence shows that around 10 of those regularly attend class. In order to collect a suitable amount of data to perform an evaluation the net must be widened to include participants from outside the course, perhaps including those with prior experience in STEM subjects or students in general. The aim was to obtain around 30 responses, as this is generally seen as a reasonable sample size in statistical analysis.

Evaluating the system requires the collection of data and supporting evidence from which conclusions may be drawn. There are a number of ways in which this data could be collected; for example, in-person evaluations may be performed and the sessions recorded. These recordings would then be analysed to obtain feedback. However, this method presents some issues: firstly, the participant may feel pressured to give a positive response in order to please the person conducting the evaluation. Second, whilst observing the participant would provide valuable insight into how participants use the application, it is very time consuming and difficult to find willing participants given the required time investment. To obtain the desired 30 responses, a more efficient method of data collection was required.

Two possible methods which would enable remote participation involve collecting usage information or distributing an online survey to gather feedback. These methods are useful but can be unreliable; when collecting usage information if the user has JavaScript disabled or their internet connection drops then loss of data could occur. It is also difficult to determine whether respondents to an online survey have actually used the software without witnessing them do so.

The most effective solution would therefore involve a mixture of all three methods: performing a few evaluation sessions in person would ensure that the methodology was sound and that the automated data collection functioned as intended, whilst enabling participants to complete the evaluation in their own time and removing the need for an assessor to be present. This also alleviates any pressure the participant may feel to provide a positive response.

This was the method chosen to evaluate the project: participants were given a link to the application and usage information was collected with their consent. Upon accessing the application for the first time, users were asked two questions in order to categorize them (see §7.3). A link in the application gave users the option to participate in a feedback survey. Three in-person evaluation sessions were held: two one-to-one sessions in order to validate the evaluation method itself, and one in-class session with the COPT students.

These sessions went well and thus an invitation was extended to the following sources:

- The remaining COPT students who did not attend the in-class session;
- Friends and family through my personal Facebook feed;
- Members of the "Informatics – Class of 2016 – University of Edinburgh" and "Informatics – University of Edinburgh" Facebook pages;
- Students of the Compilers courses at the Universities of Bristol, Birming-

ham, Durham, Oxford, Manchester, Warwick, Imperial College London and UCL, whose organisers' e-mail addresses were publicly available; and

- Members of the /r/compilers, /r/computerscience and /r/webdev communities on Reddit [20].

This should encourage participants from a wide range of backgrounds, including those with prior experience with or an interest in studying data-flow analysis, those with knowledge of similar material such as Computer Science and STEM students, developers of web applications who should be familiar with user experience (UX) and UI design, and those completely unrelated to any technical fields.

All participants were presented with the same version of the software and were encouraged to be truthful and honest with their feedback.

7.2.1 Usage Data

To collect usage information, the use of some tracking system was required. A variety of web platforms exist for this purpose including Google Analytics and Piwik. However, these platforms are intended for large-scale data collection and as such are not suited for the fine-grained analysis necessary in the evaluation of this project. For example, it is not possible to link tracked events to specific users or view a complete list of triggered events using Google Analytics. Furthermore, extended use of these platforms often comes at significant financial cost. It is also important to consider the ethical issues that may arise from sharing usage data with companies like Google; using such a platform would require a much deeper discussion around user privacy and use of the data for financial gain.

Developing a custom event-tracking API would allow full control of the data collected both in terms of accessing the raw data and maintaining user privacy. The data model used in this API must be able to store the following on a per-user basis:

- Page views;
- Button clicks and interactions with components;
- Page load times; and
- Question attempts and overall scores.

This data can be collected using an API with a single endpoint[1], based on the Google Analytics event model. Each `event` record has the following fields:

- A `tracking token` used to group events by user;
- A `type` such as page view or button click;

[1] An endpoint is a URL at which a particular API service may be accessed, e.g. `api.com/users/` may provide access to user data

- The `datetime` at which the event occurred; and
- Four levels of categorization: `category`, `action`, `label`, and `value`. Each field can be used to group similar events and store values such as question scores or which element was clicked. The actual content of these fields does not matter as long as the grouping is consistent.

The tracking token is generated when the user first visits the page and persists across visits to the site. Storing this token enables analysis of how a user's actions affect their achievement of learning outcomes, such as whether completing a given tutorial or using a simulator component increases question scores in the related topic. It would be possible to determine which elements of the software users find difficult to understand based on their interactions with each component.

This design was implemented in Python using Django REST Framework (DRF)[21], due to the ease with which one may develop a simple web API. This application was hosted on an Apache server using the `mod_wsgi` module. The API receives events as JSON data and stores them in a PostgreSQL database hosted on the same server. A small JavaScript library was written to generate identification tokens and send user events to the API.

All of the data sent to the API is completely anonymous; the tracking token is randomly generated and exists only to tie records to a given session. In addition to the events outlined above, the application sends the user's categorization answers to the API to provide context to the data collected.

7.2.2 Feedback Survey

After using the software, participants had the option to fill out a short evaluation survey. This survey was produced using Google Forms[22] because of its flexibility and low risk regarding loss of data.

The first page of the form, which requests the same categorization information as the application, is automatically filled if the user follows the link to the survey from the application's main menu. This reduces the amount of time users must spend filling out the survey and increases consistency between the data collected via the API and the survey.

Following the example set by Mustafa[7] (see §3.3), the survey consists of a number of Likert scale questions ranging from "Strongly Agree" to "Strongly Disagree". Participants were required to select a response to each Likert item, but were given a "No Response" option instead of the usual "Neutral" response which allowed them to explicitly indicate that they did not wish to respond or that they had no strong feelings one way or another. This provided context as to why participants did not respond to a particular item, ensuring that they did not accidentally miss a question.

More information about the content of the survey can be found throughout the remainder of this chapter.

7.3 Demographics

During the two-week evaluation period 184 unique users visited the application, 37 of whom used the software for more than 5 minutes. The data used throughout the rest of this chapter has been culled to those 37 users who spent more than 5 minutes with the software.

Upon accessing the application for the first time, users were presented with two questions in order to categorize them based on their interest and experience in data-flow analysis. The first question asked them to rate their level of experience given the following options:

- None at all
- Basic
- Advanced

A summary of the response to the this question is shown in fig. 7.1.

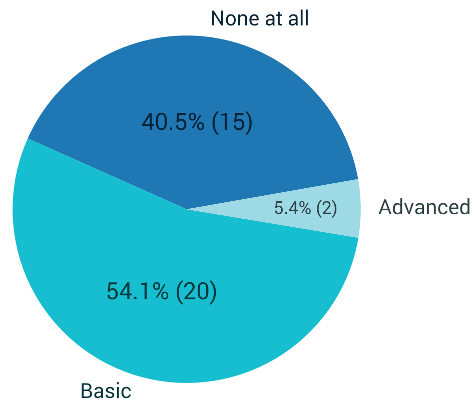


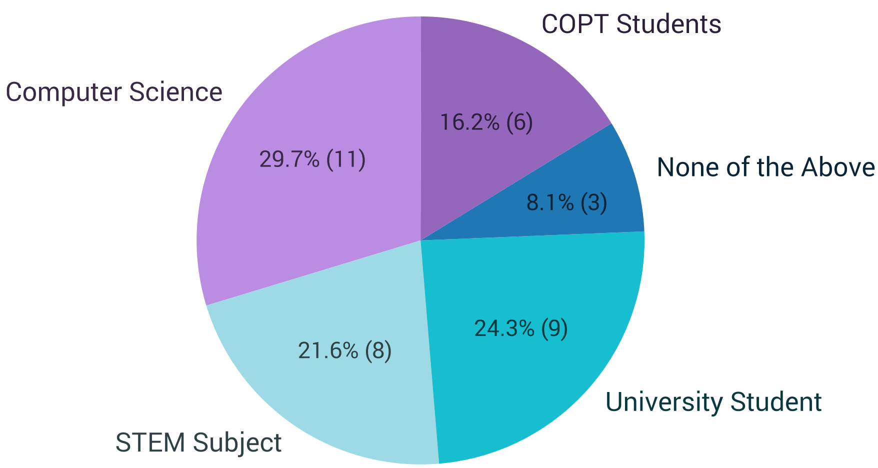
Fig. 7.1: Distribution of User Subject Experience

The second question asked them to describe their academic background by selecting from a number of choices. These options were then ranked in the following order, by their relation to data-flow analysis:

1. I am a member of the COPT course (UoE only)
2. I have studied Computer Science at degree level.
3. I have studied a STEM subject at degree level.
4. I am a student.
5. (None of the above)

Each user has been grouped by the highest-ranked option they selected; the distribution of these groups is shown in fig. 7.2.

Fig. 7.2: Distribution of User Academic Experience



These figures are quite positive. The participants come from a wide range of academic backgrounds and have a good mixture of prior experience with the topic. Although the number of participants is only 20% of the total number of unique visitors, the low retention rate may be attributed to a number of factors:

- Users who visited the application but did not have time to invest in evaluating it;
- Users who visited the site but did not want to participate in the evaluation;
- Users who were not interested in the content; and
- Web crawlers visiting the page.

For an application targeting a relatively niche subject, a 20% retention rate seems quite reasonable and does not indicate any particular flaws with the system.

7.4 Critical Analysis

In this section I will attempt to draw conclusions about the system by analysing usage data collected over the evaluation period.

7.4.1 Time Spent

As mentioned in §7.3, a total of 37 participants used the application for more than 5 minutes. This information was calculated by examining blocks of events triggered by each user. A *block* is defined as a sequence of events in which there is less than a 10-minute gap between consecutive records; the duration of these blocks is summed to produce an estimate of time spent during the two-week evaluation period. Fig. 7.3 shows the distribution of time spent by those 37 participants, categorised by their experience with the subject.

From this data, we can draw a number of conclusions: first, that a significant amount of users found the software engaging or at least of some use – more than half of the participants used the software for more than 15 minutes, with at least one individual using it for over two hours. Of the users who stayed for more than 15 minutes, most tended to invest a significant amount of time into using the application. This is promising – anecdotally, I have noticed that project evaluation participants tend to quickly scan through the available features and then fill in the survey without spending enough time to thoroughly evaluate the application; however, the majority of users in this evaluation seem to have shown genuine interest in the system and its content.

Second, it appears that the more experience a user has, the less time they are likely to spend using the application. Users who considered themselves "Advanced" in the topic spent the least time with the software, whilst those with no experience spent the most. This could indicate that the content presented by the system is too simple or that it re-treads too much ground covered by other sources such as lectures or textbooks.

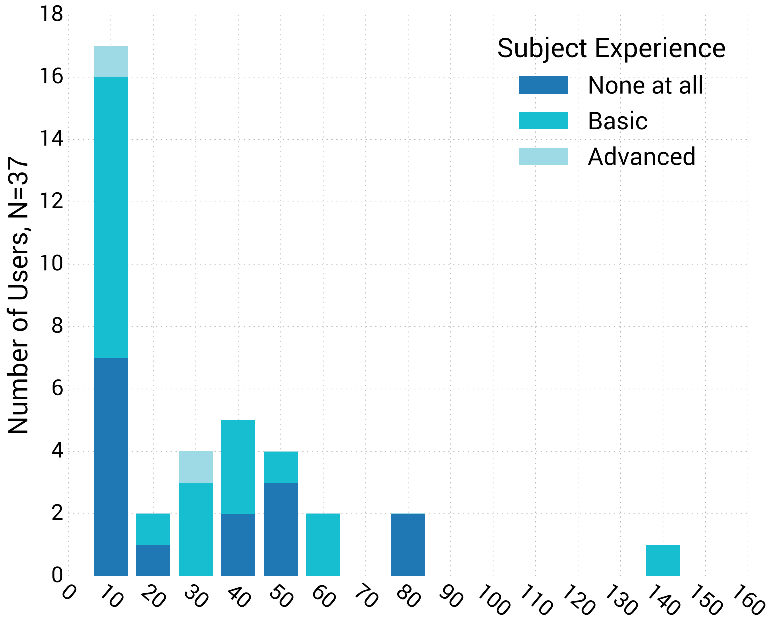
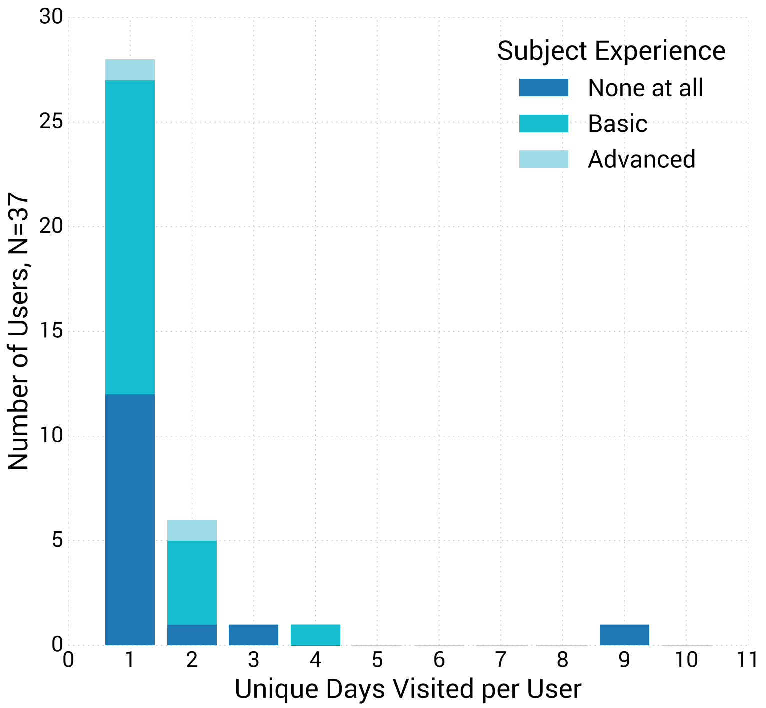


Fig. 7.3: Distribution of Time Spent Using the Application per User

Users were encouraged to continue using the software after they had participated in the evaluation and completed the feedback survey. Fig 7.4 shows how often users returned to the site. Although the majority of users only visited on the day they evaluated the application, 24% returned to the application days later, one user in particular visiting on 9 separate days.

Fig. 7.4: Distribution of Returning Users



7.4.2 Lessons Completed

Fig. 7.5 shows the number of lessons attempted versus the number of lessons completed. The pale bars represent the percentage of people who attempted each lesson, whilst the darker bars represent the percentage of people who completed them. The numbers in each bar represent the proportion of those who started the lesson and subsequently completed it.

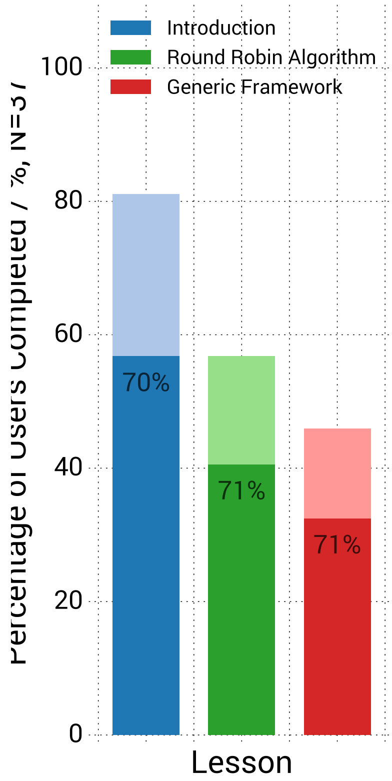


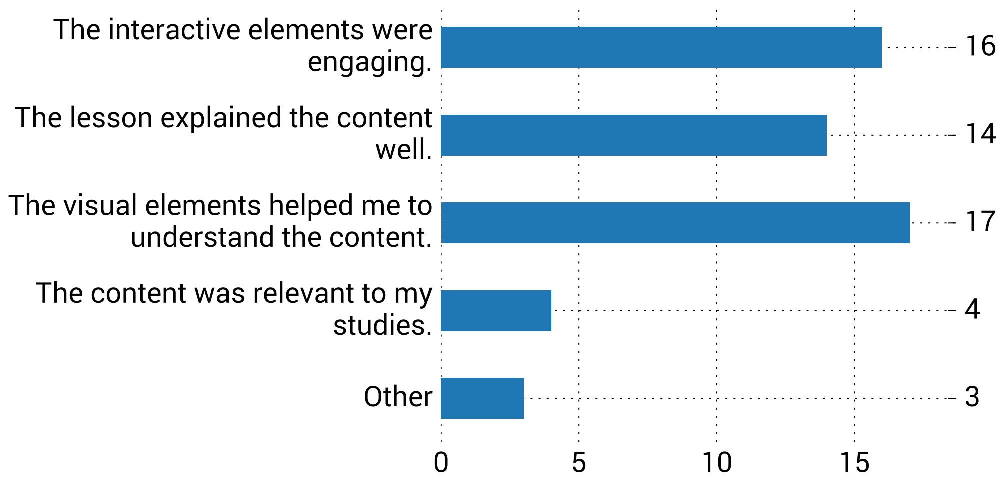
Fig. 7.5: Lessons Completed by Participants

The results show that most users attempted the introductory lesson, with participation decreasing moderately for subsequent lessons. This is as expected; during the in-person evaluations participants were recommended to at least complete the introductory lesson. However, the participation rate for subsequent lessons is excellent – above 50% of the participants voluntarily attempted the second lesson, and 45% attempted the third.

The completion rate is fairly consistent, at around 70%. Each lesson requires some form of active participation in order to continue, indicating that the lessons kept users engaged throughout.

In total, 28 users responded to the feedback survey, 25 of whom said that they had used the tutorial system. These users were asked to provide reasons why they did or did not complete a lesson after starting it. The feedback from these questions is shown in figures 7.6 & 7.7.

Fig. 7.6: Reasons Provided for Completing Lessons (N=25)



The majority of users credited the interactive and visual elements as reasons for completing lessons. Half of the responses to the survey said that the lessons explained the content well, but only 14% said that it was relevant to their studies. This may be due to the low participation by COPT students, but in any case implies that perhaps more care should be taken when considering what material to cover.

Of the three "Other" responses, two were joke comments, but the third was particularly revealing: a user responded that being able to select from existing answers, rather than having to type in formulae, was "a win". Online assessment platforms often require users to type in a textual representation of some formulae, but marking of these answers via a computer is difficult and is prone to false negatives. I had considered including some text-answer questions, but decided against it for this reason.

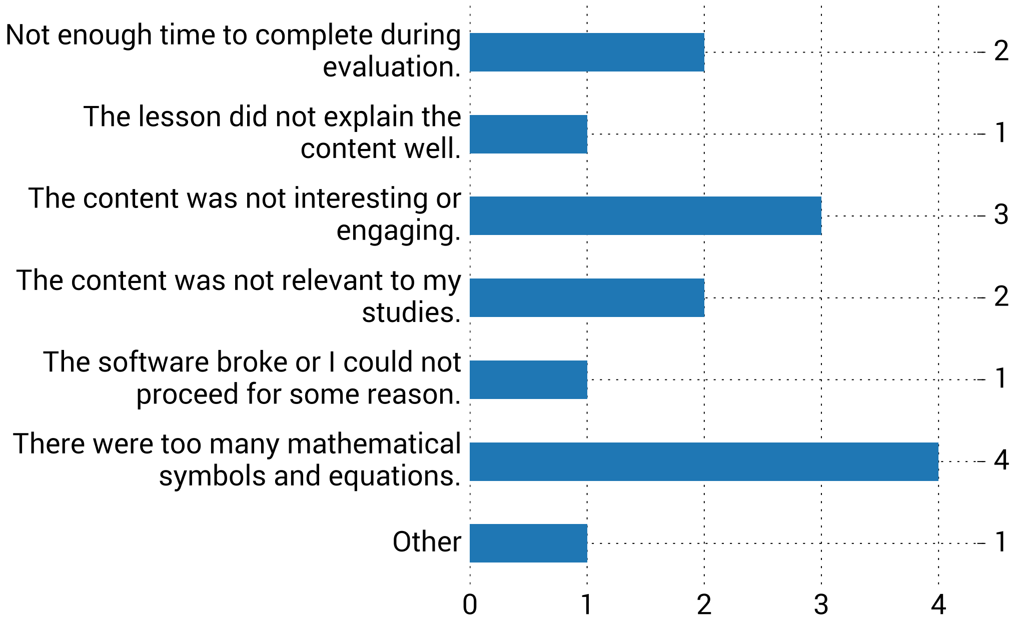


Fig. 7.7: Reasons Provided for Abandoning Lessons (N=25)

Of those who abandoned a lesson, the majority opinion was that there were too many mathematical formulae involved. This is true of the last lesson in the series, but unfortunately this is unavoidable as a large portion of the material is based in mathematical proof. However, this is certainly a point to consider when designing the content for learning platforms in the future, and efforts could be made to simplify the information presented by this system.

7.4.3 Achievement of Learning Outcomes

Participants were given the option to take a short content review test (consisting of 10 questions) during their use of the app. Of the 37 participants, 16 of them made a total of 20 attempts. I have analysed the scores achieved in each attempt

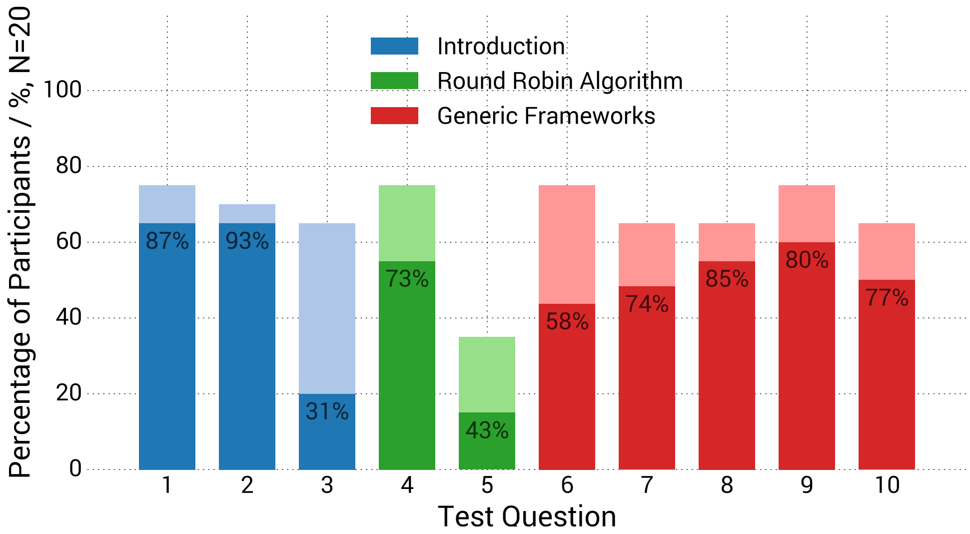


Fig. 7.8: Attempts & Answers to Test Questions

in order to assess the participants' achievement of learning outcomes; this graph is shown in fig. 7.8.

The pale bars represent the number of participants who attempted each question, whilst the darker bars represent the average score achieved by all participants. The numbers in each bar represent the score achieved by those who attempted that question. Questions are color-coded by the lesson they are meant to assess.

Most questions presented the user with 4 possible answers and allowed them to select only one. Questions 6 & 7 allow multiple choices, but users are penalised for incorrect answers. The expected score obtained by random guessing is therefore approximately 25%, or 1-in-4.

During the in-person evaluations, some participants noted that they were confused by the submission button - they had expected it to submit only one question, when in fact it submitted the whole test. This could explain why all questions have an attempt rate below 80%, the missing 20% attributed to those who accidentally submitted the whole test.

In addition, participants noted that questions 3 and 5 were misleading: the answer to question 3 appeared to be displayed in the accompanying diagram, so many participants did not realise they were required to perform a calculation and thus answered incorrectly. In addition, many participants felt that question 5 was not covered by the lesson's content.

Disregarding those two factors, the results are very positive. More than 85% of participants understood the basic content, with only a slight drop in score as the complexity of the content increases. This implies that the application does, in fact, effectively teach the course content.

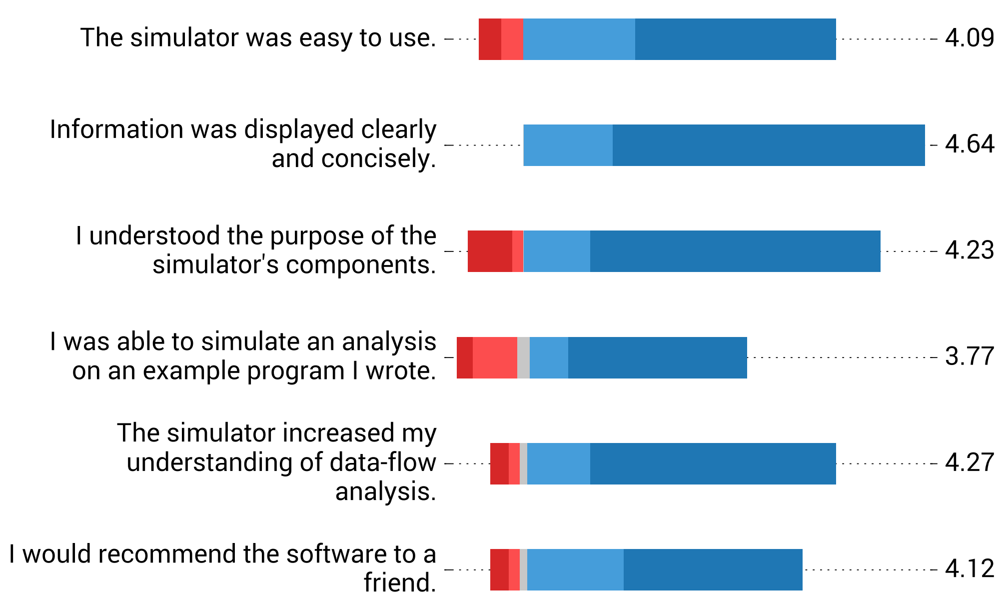
Of course, these results are only an indication. It is highly possible that only academically-gifted participants took the test or that the questions covered content with which they were already familiar. However, the results are well above the expected average and optimistically show that the system achieved its goal.

7.4.4 User Opinion

In the feedback survey users were asked a number of Likert scale questions, rating their opinion on a scale from "Strongly Agree" to "Strongly Disagree". A response was mandatory for those who used each aspect of the system; the number of respondents can be found in the figure captions.

Figures 7.9 & 7.10 show the response to each item. The size of the "Strongly Agree" and "Strongly Disagree" bars are doubled to visualise the weighted response to each item; the scores on the right-hand side show the actual average (5 being the most positive, 0 being the least).

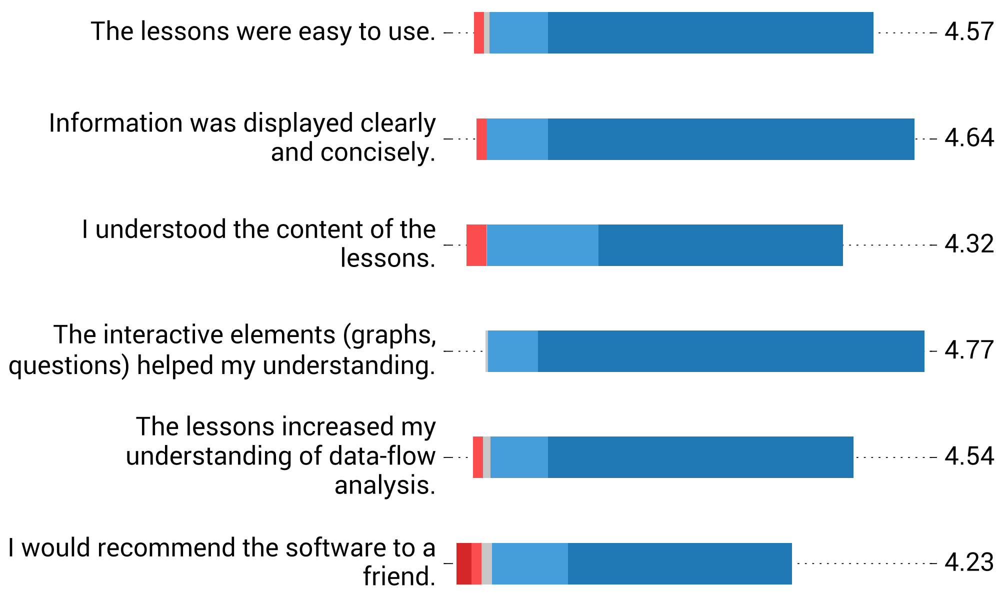
Fig. 7.9: Opinion of the Round-Robin Simulator, N=22



Reactions to the simulator were quite positive, with participants praising the way it displayed components of the analysis. Most users found the simulation easy to use, but there was some concern over the complexity of the interface and the content involved.

One interesting point to note is that many users were unable to simulate an analysis on their own programs, or declined to respond to that item (perhaps for the same reason). Initially, the platform did not have any instruction as to how to write ILOC programs, so users were forced to rely on the examples in lessons and their own intuition for guidance. This oversight was corrected shortly after the evaluation period began by adding a help screen, which seems to have improved the response somewhat.

Fig. 7.10: Opinion of the Interactive Lessons, N=25



Opinion of the interactive tutorials was more positive – most users agreed that the interface was easy to use and that the information was conveyed well. Users generally understood the content and all agreed that the interactive components helped increased their learning.

Whilst the organisation of the content could use some improvement, it appears that overall opinion is that the system works well. Users were asked to select the methods of study they used most often, and how they would use the learning platform alongside those methods of study. The response to these questions is shown in figures 7.13 & **??**.

The response to the first question was extremely positive – only one person said that they would not use the platform to study over the other methods they chose. When asked if they planned to continue using the software after the evaluation period was over, exactly 50% said that they would do so.

Fig. 7.11: Overall User Opinion, N=27

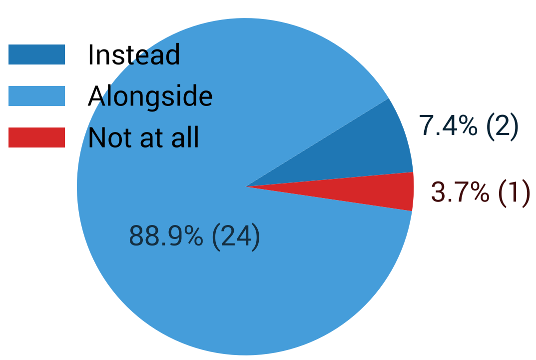
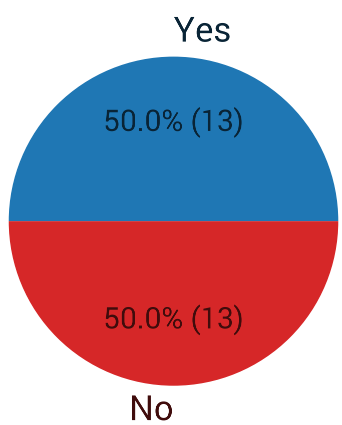


Fig. 7.12: Opinion on Future Use, N=27



Given the information about which methods participants use to study, this is hardly surprising – the interactive tutorials are very similar to the most popular option ("Reading lecture slides / notes") only with more active participation involved.

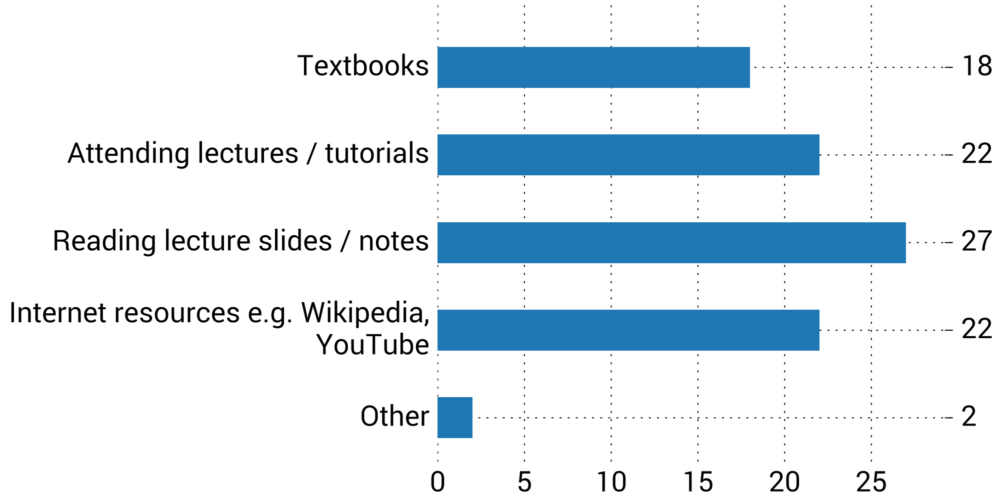


Fig. 7.13: Methods of Study, N=27

7.4.5 Written Feedback

Users were asked to provide written feedback both in the online invitations and throughout the survey, in order to gather insight into user opinion and potential improvements which could be made. A fair amount of the written feedback was incredibly positive; what follows is a selection of those comments:

> "The lessons were quite interesting. I liked the concise and short amounts of information in each step - never felt overwhelmed."

> "I wish the uni would use things like this for teaching - it's far easier to understand than parsing through lecture notes."

> "This is superb! Thank you for this. I am just learning about dataflow analysis in my compilers course."

> "... great job! I'm in my third year of PhD in compiler optimizations and found your tool really nice and illustratory (*thumbs-up emoji*)."

> "I like how upon giving a correct answer, it is still explained! When giving answers in first lesson, I was still not 100% sure about them, so having them explained even when I was correct was nice."

Of the negative feedback, most was constructive criticism. Users suggested a range of improvements to the site, primarily relating to the lessons and how they could be used to improve understanding of the simulation. Many users complained about the organisation of the main menu (§**??**), due to the confusing order in which the menu items were laid out. Users also desired a summary of each lesson and the system as a whole; the menu does not really explain the purpose of the software, which put some users off exploring it further.

Regarding the simulation, users were confused by the colour co-ordination and felt that it could have been improved upon. In addition, there was some concern about the lack of guidance. I had assumed that most people would complete the introductory lesson before playing with the simulator, but it appears that this was not the case – or that perhaps it was difficult to relate the lesson content to simulator's components. One user in particular left the following comment:

> "Few bugs here and there but simulation is engaging... more could be done to link between description and simulation."

The lessons also received some minor criticism. The most common opinion was that the interface lacked some basic features, such as an indicator of progress or the ability to navigate directly to each section of the lesson. The segmentation of content was met with a mixed response; some users complained that they had to click the "Next" button too many times whilst others appreciated that the information was split into bite-sized chunks.

A small number of bugs were brought to my attention. Firstly, in Firefox browsers the navigation buttons do not always function as expected. Unfortunately this is due to an issue with the dagre-d3 library, but it could be possible to patch it with further testing. Users also reported that the back button in their browser did not work. This was an oversight relating to the implementation of the `View` system and could be fixed with some minor tweaks to the way pages are loaded.

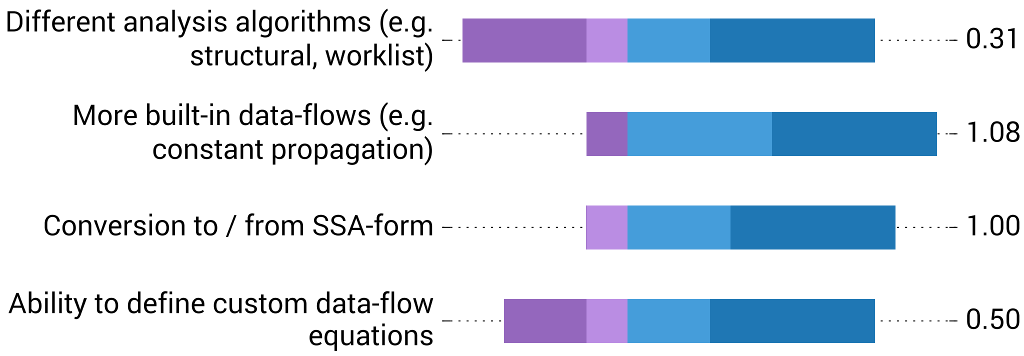
7.5 Future Developments

Throughout this report suggestions have been made for potential improvements to the system, ranging from small usability changes such as adding progress indicators and navigation elements to overhauling the content in order to strengthen the link between the tutorials and the simulation. However, there has not yet been any discussion around long-term future developments or content to be covered.

Participants in the survey who stated that they had used the simulator were asked to rank a list of potential future features in order from most to least important. The response to this question is shown in fig. 7.14.

The choices listed were based on early discussions between myself and the project's supervisor, Hugh Leather. His original proposal for the project suggested a possible extension which would allow users to define their own data-flow frameworks and simulate an analysis using them. Although I chose to focus on other aspects of the system, this idea was one that I was keen to see realised if time permitted. Examining the responses it seems that focusing elsewhere was the right choice

Fig. 7.14: Opinion on Future Simulator Improvements, N=22



after all; users largely preferred that data-flows be written for them rather than by them, perhaps because they were not aware of how or why they would use this feature. Some of the written responses to the survey would agree – participants wrote that although they had learned how to perform data-flow analysis, they did not know what they would use it for. To rectify this the content could be expanded to demonstrate the applications of the topic both in compilers and in other areas such as web security[1]; maybe then users would find value in being able to write their own data-flows.

The data-flow framework system would need to be improved in order to add interesting new data-flows. Whilst it is certainly possible to add more bit-vector data-flows, tuple-valued data-flows would need to be implemented to support data-flows such as constant propagation. This, in turn, would require an overhaul of the Hasse diagram visualisation to make it capable of displaying lattices for more complex data-flows.

The second most popular improvement was conversion to and from SSA-form[2]. The lecture slides cover this topic quite heavily, but I decided against including it because its applications lie closer to performing compiler optimisations than to data-flow analysis itself. If the content were to be expanded, SSA-form would be one of the more simple features to include and would open the door to visualising optimising transformations such as code motion.

Further down the line, the control-flow graph visualisation could be adapted to allow users to edit it directly. This improved visualisation would allow users to add, delete or edit nodes, making connections or removing them in order to explore how changing the CFG's structure or local information changes the problem's solution.

Finally, the interactive tutorials have huge potential for enhancement. Currently, the main form of interaction is through multiple-choice questions. A more advanced system could allow users to write ILOC programs to solve problems, or if the custom data-flow framework system were implemented users could be asked

[2]Single Static Assignment (SSA) form is a transformation applied to control-flow graphs in which variables are copied and renamed so that every variable is defined only once.

to write frameworks to solve a given data-flow problem. The presentation of the content could be diversified using audio or video formats or automated demonstrations which control the user's mouse and keyboard.

7.6 Conclusion

The main objective of this project was to develop a learning platform for compiler data-flow analysis which provided a more interactive alternative to traditional teaching formats, and to demonstrate that such a system would be an effective tool for learning.

The results collected throughout this evaluation are strong evidence that the project has achieved this goal. Response to the feedback survey was overwhelmingly positive; the visual and interactive elements received particularly high praise, as did the presentation of the material. Achievement of learning outcomes was well above average, users were kept engaged by the platform, and all but one participant stated that they would use the system alongside other methods of study.

However, there are some lessons to be learned. Much time was spent rectifying mistakes made early in development; whether caused by lack of experience or rushed decision-making, some forethought would not have gone amiss. Perhaps a different development process would have revealed these issues before they became significant problems, or more time should have been spent researching technologies before diving into the implementation.

In the near future, I hope to see others extend the system I have developed or apply the techniques I have used to enhance teaching in other subject areas. The system has proven that online platforms are a valuable and effective tool for learning, one which must be explored further in order to truly realise their potential.

Applications like the one developed in this project are only the beginning. As more and more industries move online, it is only natural that teaching move with them. It is time for higher education institutions to diversify the ways in which they teach their students and make efforts to appeal to a range of learning styles. By developing their own tools such as this one or pooling their resources with an entity like Coursera, Universities can truly bring teaching into the digital age.

Bibliography

[1] A. Rimsa, M. d'Amorim, and F. Pereira. Tainted Flow Analysis on e-SSA-form Programs. *International Conference on Compiler Construction*, 2011.

[2] K. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers, 2nd edition, 2011.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1st edition, 1986.

[4] Coursera. About Coursera. `https://www.coursera.org/about/`, 2015. Accessed: 2015-09-20.

[5] HackerRank. Reverse a linked list: Challenges. `https://www.hackerrank.com/challenges/reverse-a-linked-list/`, 2015. Accessed: 2015-09-21.

[6] Khan Academy. Implementing binary search of an array. `https://www.khanacademy.org/computing/computer-science/algorithms/binary-search/a/implementing-binary-search-of-an-array`, 2015. Accessed: 2015-09-21.

[7] B. Mustafa. Evaluating A System Simulator For Computer Architecture Teaching And Learning Support. *Innovation in Teaching and Learning in Information and Computer Sciences*, 9(1):100–104, 2010.

[8] R. M. Felder and L. K. Silverman. Learning and Teaching Styles In Engineering Education. *Engineering Education*, 78(7):674–681, 1988.

[9] B.J. Farr. *Individual Differences in Learning: Predicting One's More Effective Learning Modality*. PhD thesis, Catholic University of America, 1970.

[10] R. S. Dunn and K. J. Dunn. Learning Styles / Teaching Styles: Should They... Can They... Be Matched? *Educational Leadership*, 36:238 – 244, 1979.

[11] F. Coffield, D. Moseley, E. Hall, and K. Ecclestone. *Learning styles and pedagogy in post-16 learning*. The Learning and Skills Research Centre, 2004.

[12] M. D. Merrill. Instructional Strategies and Learning Styles: Which takes Precedence? *Trends and Issues in Instructional Technology*, 2000.

[13] David Majda. PEG.js – Parser Generator for JavaScript. `http://pegjs.org/`, 2016. Accessed: 2016-03-25.

[14] Troels Knak-Nielsen. Calculate All Combinations (permutations). `https://dzone.com/articles/calculate-all-combinations`, 2016. Accessed: 2016-03-25.

[15] Google Inc. AngularJS – Superheroic JavaScript MVW Framework. `https://angularjs.org/`, 2016. Accessed: 2016-03-26.

[16] Chris Pettitt. cpettitt/dagre-d3 Wiki. `https://github.com/cpettitt/dagre-d3/wiki#demos`, 2016. Accessed: 2016-03-26.

[17] Cytoscape Consortium. Cytoscape.js. `http://js.cytoscape.org/`, 2015. Accessed: 2016-03-26.

[18] Almende B.V. vis.js. `http://visjs.org/`, 2016. Accessed: 2016-03-26.

[19] Alan Williamson. jQuery Lined TextArea. `http://alan.blog-city.com/jquerylinedtextarea.htm`, 2016. Accessed: 2016-03-26.

[20] Reddit, Inc. reddit: the front page of the internet. `https://www.reddit.com/`, 2016. Accessed: 2016-03-30.

[21] Christie, T. Django REST Framework. `http://www.django-rest-framework.org/`, 2016. Accessed: 2016-03-30.

[22] Google, Inc. Google Forms. `https://forms.google.com/`, 2016. Accessed: 2016-03-30.

[23] Wikipedia. Context-free grammar. `https://en.wikipedia.org/wiki/Context-free_grammar`, 2016. Accessed: 2016-03-25.

Appendix A

Terminology

A.1 Glossary

available expression An expression is *available* if it has been computed along all paths leading to the current node. 1, 27, 71

bit-vector data-flow Data-flows in which values come as single items, and are either included or excluded in each set. Such data-flows may be represented by a vector of bits, each bit representing a value and a 0 or 1 indicating the presence or absence of that value in the set.. 8, 27, 33, 63

boundary The initial value at the starting point of an analysis, i.e. In($ENTRY$) or Out($EXIT$). 10, 26

context-free grammar A context-free grammar describes a formal language using rules of the form $A \to \alpha$, in which A is a single non-terminal symbol and α is a series of terminal or non-terminal symbols. The grammar is "context-free" in that its rules may be applied regardless of the context of the non-terminal A[23]. 29

control-flow graph A graph representing the possible execution paths in a program. Nodes represent instructions, edges represent possible jumps between said instructions. 1–3, 6, 7, 36, 37, 63, 67

data-flow A system of equations and conditions which constitute a data-flow problem, that is, a problem which may be solved through data-flow analysis. 2, 3, 7–11, 19, 25, 27, 31, 33, 34, 43, 63, 64, 67–69

data-flow analysis A technique for gathering information at various points in a control-flow graph. i, 1–3, 6, 9, 13, 17, 19, 21, 25, 29, 41, 49, 51, 53

data-flow equations A system of equations which determine how data flows through a CFG. 7, 9–11, 26

direction The direction in which data flows in a data-flow problem. Either forward (from the entry point of the CFG to the exit point) or backward (the opposite). 7, 10, 26

domain The domain of values considered in a data-flow problem, e.g. definitions or expressions. 10, 11, 26

dominate In a CFG a node n_i is said to dominate a node n_j if every path from n_0 (the entry node) to n_j must go through n_i[2, p. 478]. 68, 71

fixed-point A computation in which the required process is repeated until the state stops changing. 9, 11, 25

Hasse diagram A diagram used to represent partially ordered sets. Nodes represent elements of the sets, edges represent an ordering between a pair of elements. 3, 8, 9, 33, 34, 37

Likert scale A method of gauging opinion on a topic, consisting of a collection of Likert items. Each Likert item contains a statement followed by an odd number of responses, containing an equal number of positive and negative responses balanced such that the difference between responses is uniform. An example Likert item is the oft-used Strongly Agree / Strongly Disagree scale, which may be weighted from from 1-5. The Likert scale is the sum of weights of responses to Likert items. 15, 17, 52, 59

liveness analysis A variable is *live* if its current value will be used later in the program's execution. 1, 27, 71

local information Information specific to a given node in the CFG, e.g. in reaching definitions DefGen is the set of definitions which a given node generates.. 7

meet An equation (or set of equations) which determines how data flows *between* nodes in a CFG. 7, 11, 26

meet operator An operator which defines how the meet of two sets is obtained, such as $\cup$, $\cap$ or another operator entirely. 10, 11, 26

meet semi-lattice A partially ordered set in which there exists a greatest lower bound (or meet) for any non-empty, finite subset. 3, 7, 8, 11, 33, 34

reaching definition A definition of a variable *reaches* a block if there exists at least one path from its definition to the block along which it is not overwritten. 6–10, 27, 31, 68, 71

region A set of nodes in a graph which includes a *header* node which dominates all other nodes in the region[3, p. 669]. 10

single page app a web application in which the page is only loaded once, after which JavaScript is used to update the content and asynchronous calls are used to request more data.. 36

transfer An equation (or set of equations) which determines how data flows *through* a node in a CFG. 7, 10, 11, 26, 27

tuple-valued data-flow Data-flows in which the values come in tuples; for example, in constant propagation each variable is paired with a value indicating whether it holds a constant.. 8, 27, 63

A.2 Acronyms

API application programming interface. 3, 25, 36, 37, 39, 43, 51, 52

AST abstract syntax tree. 5, 25, 29–33

CDN content delivery network. 41

CFG control-flow graph. 6–11, 25–27, 29, 31, 33, 34, 36, 37, 42, 43, 45, 63, 67–69

COPT Compiler Optimisations. 2, 20, 21, 49, 50, 57

DRF Django REST Framework. 52

DRY Don't Repeat Yourself. 43

ILOC Intermediate Language for Optimising Compilers. 3, 25, 27, 29–33

IR intermediate representation. 5, 6

JSON JavaScript Object Notation. 52

PEG Parsing Expression Grammar. 29

UI user interface. 41, 51

UX user experience. 51

Appendix B

Types of Data-Flow Analysis

Data-Flow	**Purpose**	**Applications**
Dominators	Computes the set of nodes which dominate the current node.	Computing SSA form.
Reaching definitions	Computes the set of variable definitions which are available at points in the CFG.	Generating def-use chains for other analyses.
Liveness analysis	Computes the set of variables whose current value will be used at a later point in the control flow graph.	Register allocation. Identifying useless store operations. Identifying uninitialised variables.
Available expressions	Identifies expressions which been computed at a previous point in the CFG.	Code motion.
Anticipable Expressions	Computes expressions which will be computed along all paths leading from the current point.	Code motion.
Constant Propagation	Computes the set of variables which have a constant value based on previous assignments.	Constant propagation. Dead code elimination.
Copy Propagation	Computes the set of variables whose values have been copied from another variable.	Dead code elimination. Code motion.
Tainted Flow Analysis[1]	Identifies unsafe operations which have been passed unsanitized *(tainted)* input as a parameter.	Preventing security vulnerabilities such as SQL injection and XSS attacks.

Table B.1: Types of data-flow analysis.