Chapter 2

Background

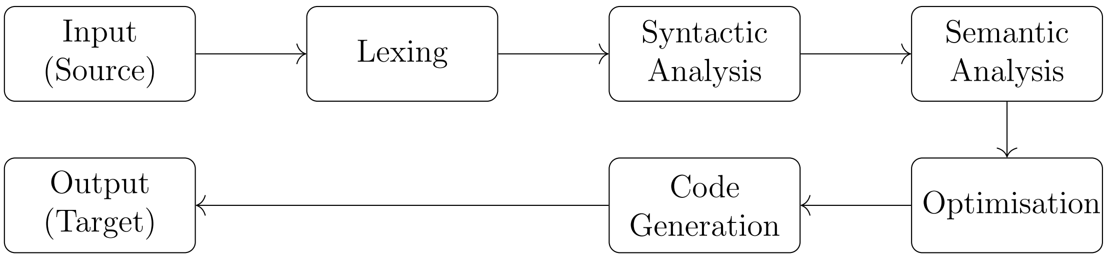
This chapter briefly covers the necessary background information required to understand the project, both to inform the reader and to demonstrate understanding of the topic.

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).

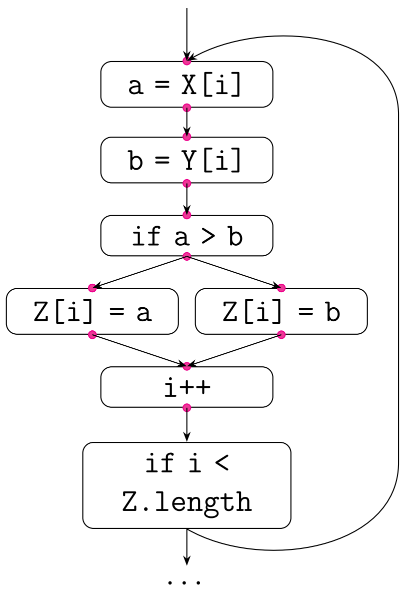


Fig. 2.1: A control-flow graph.

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.

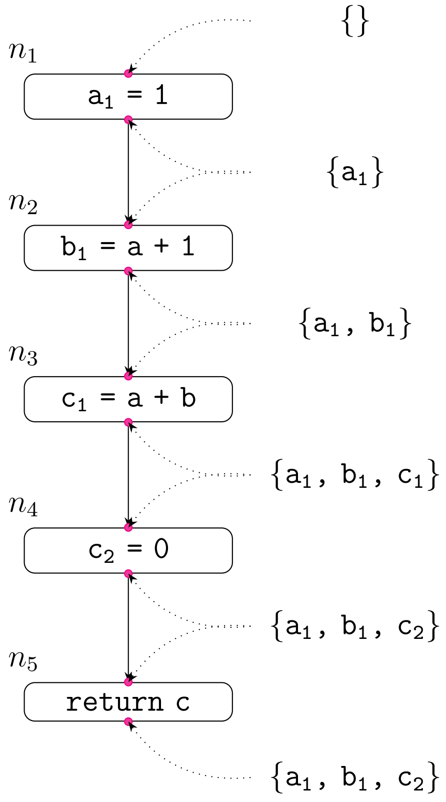
2.2.1 A Simple Example

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.


{}
n_1
$\mathtt{a}_1$ = 1
{$\mathtt{a}_1$}
n_2
$\mathtt{b}_1$ = a + 1
{$\mathtt{a}_1$, $\mathtt{b}_1$}
n_3
$\mathtt{c}_1$ = a + b
{$\mathtt{a}_1$, $\mathtt{b}_1$, $\mathtt{c}_1$}
n_4
$\mathtt{c}_2$ = 0
{$\mathtt{a}_1$, $\mathtt{b}_1$, $\mathtt{c}_2$}
n_5
return c
{$\mathtt{a}_1$, $\mathtt{b}_1$, $\mathtt{c}_2$}

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:

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

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 $\{\texttt{a}\}$ and $\{\texttt{a,b,c}\}$ is represented by the path $\{\texttt{a}\} \rightarrow \{\texttt{a}, \texttt{b}\} \rightarrow \{\texttt{a}, \texttt{b}, \texttt{c}\}$.

$$\begin{tikzcd}
 & \emptyset \arrow[dl] \arrow[d] \arrow[dr] & & (\top) \\
\{\texttt{a}\} \arrow[d] \arrow[dr] & \{\texttt{b}\} \arrow[dl] \arrow[dr] & \{\texttt{c}\} \arrow[dl] \arrow[d] & \\
\{\texttt{a, b}\} \arrow[dr] & \{\texttt{a, c}\} \arrow[d] & \{\texttt{b, c}\} \arrow[dl] & \\
 & \{\texttt{a, b, c}\} & & (\bot)
\end{tikzcd}$$

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.

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

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.

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.

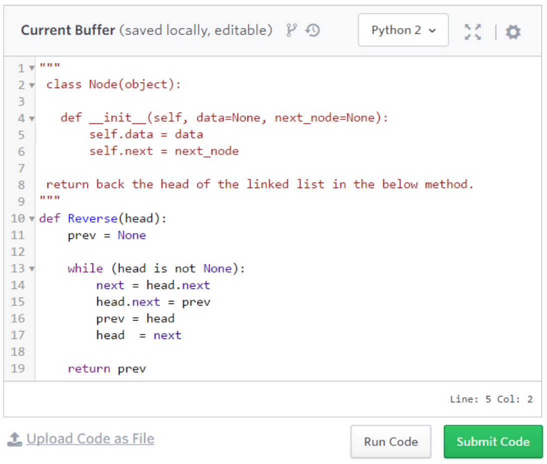
The videos are very similar in format to a traditional lecture, consisting of a voice-over accompanied by related imagery or text-based slides. However, the advantage over the traditional format is two-fold: the ability to view the content at the student's convenience, rather than in a specific location 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.



```
"""
 class Node(object):

   def __init__(self, data=None, next_node=None):
       self.data = data
       self.next = next_node

 return back the head of the linked list in the below method.
"""
def Reverse(head):
    prev = None

    while (head is not None):
        next = head.next
        head.next = prev
        prev = head
        head  = next

    return prev
```

```
Line: 5 Col: 2
```

Fig. 3.1: Code editor from HackerRank[5]

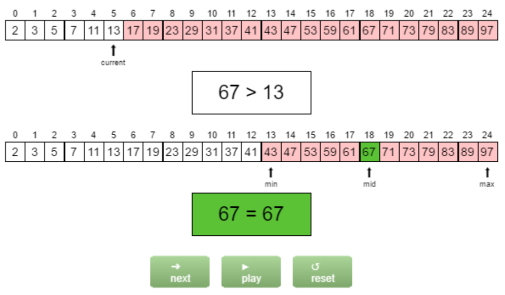


Fig. 3.2: Binary search visualisation from Khan Academy[6]

- 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.

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
```

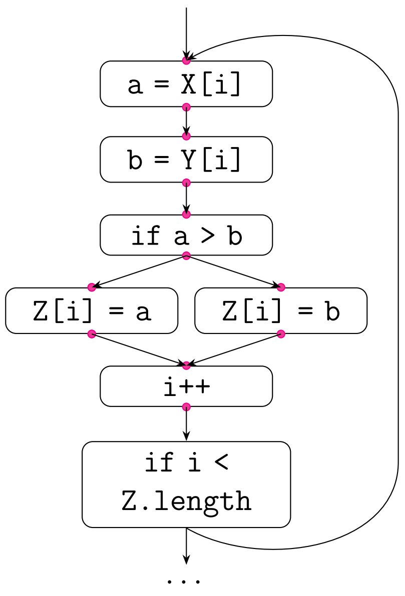


Fig. 5.1: A control-flow graph.

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
    }
  })();
}
```

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.

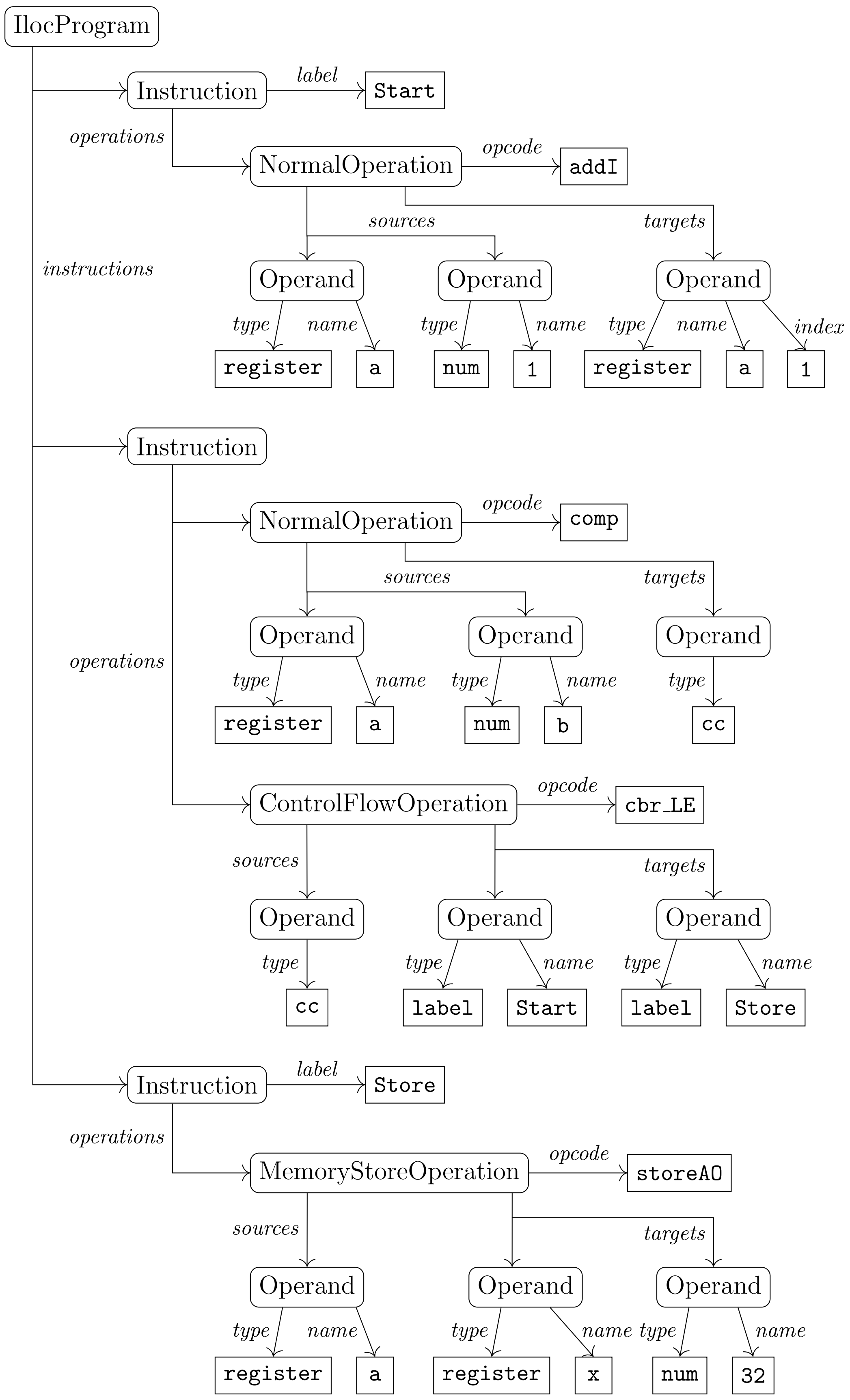


Fig. 5.3: The AST for the ILOC program in fig. 5.4.

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 $x \rightarrow y$ and from $y \rightarrow z$, any edge from $x \rightarrow z$ may be removed as it is represented by the path $x \rightarrow y \rightarrow z$. 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.

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.

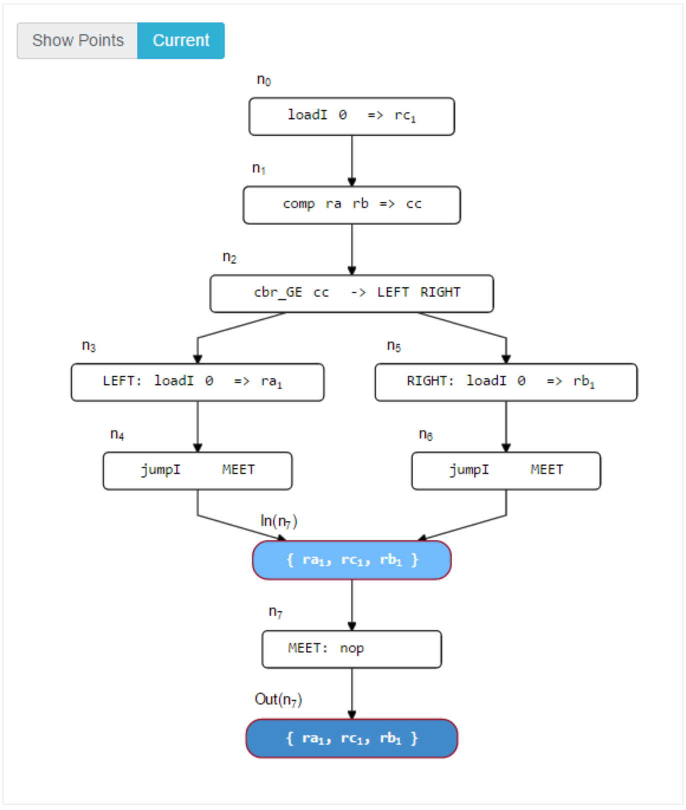


Fig. 6.1: Control-Flow Graph Visualisation

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.

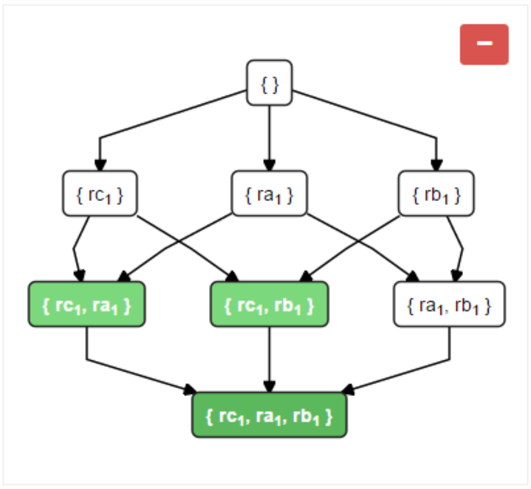
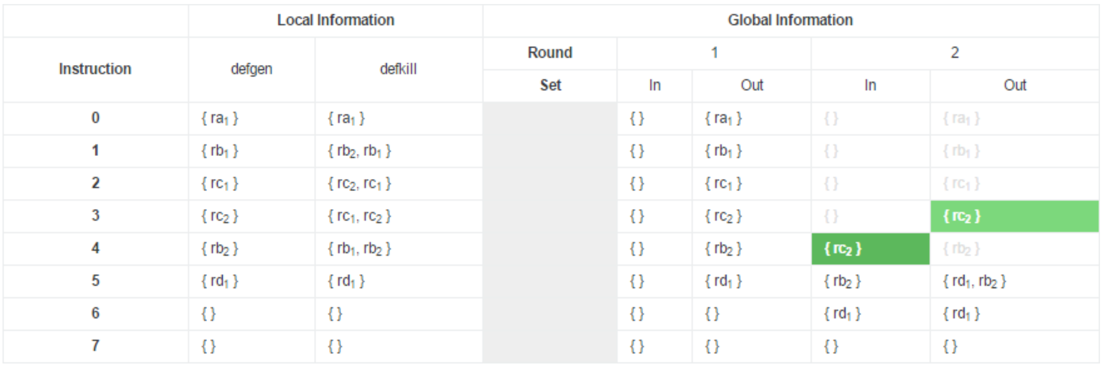


Fig. 6.2: Hasse Diagram Visualisation

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.

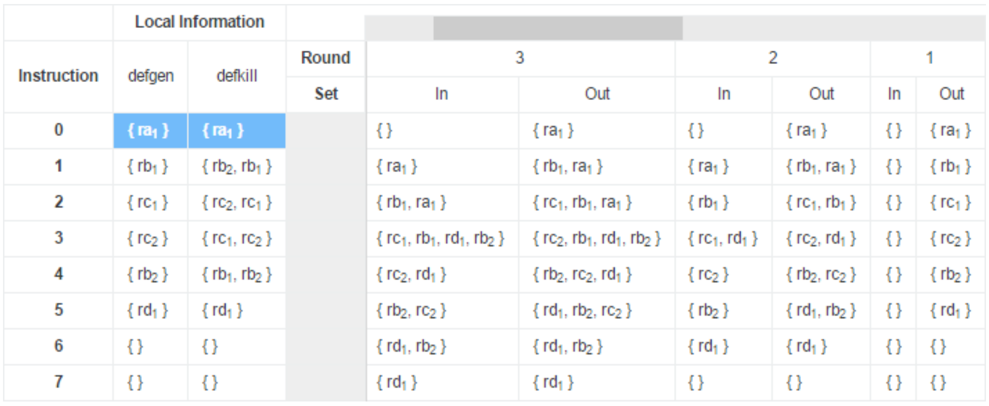


	Local Information		Global Information				
Instruction	defgen	defkill	Round	1		2	
			Set	In	Out	In	Out
0	$\{ ra_1 \}$	$\{ ra_1 \}$		$\{\}$	$\{ ra_1 \}$	$\{\}$	$\{ ra_1 \}$
1	$\{ rb_1 \}$	$\{ rb_2, rb_1 \}$		$\{\}$	$\{ rb_1 \}$	$\{\}$	$\{ rb_1 \}$
2	$\{ rc_1 \}$	$\{ rc_2, rc_1 \}$		$\{\}$	$\{ rc_1 \}$	$\{\}$	$\{ rc_1 \}$
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.

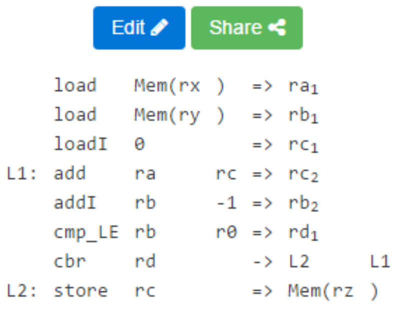


	Local Information								
Instruction	defgen	defkill	Round	3		2		1	
			Set	In	Out	In	Out	In	Out
0	$\{ ra_1 \}$	$\{ ra_1 \}$		$\{\}$	$\{ ra_1 \}$	$\{\}$	$\{ ra_1 \}$	$\{\}$	$\{ ra_1 \}$
1	$\{ rb_1 \}$	$\{ rb_2, rb_1 \}$		$\{ ra_1 \}$	$\{ rb_1, ra_1 \}$	$\{ ra_1 \}$	$\{ rb_1, ra_1 \}$	$\{\}$	$\{ rb_1 \}$
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

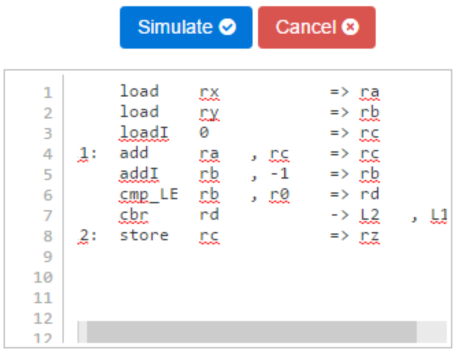
6.3.4 Code Display

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.



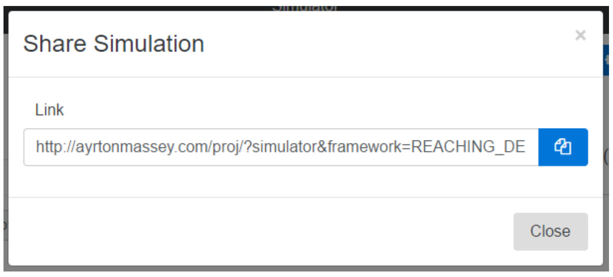
```
    load    Mem(rx )  => ra_1
    load    Mem(ry )  => rb_1
    loadI   0         => rc_1
L1: add     ra     rc => rc_2
    addI    rb     -1 => rb_2
    cmp_LE  rb     r0 => rd_1
    cbr     rd        -> L2     L1
L2: store   rc        => Mem(rz )
```

Fig. 6.5: Default Code Display



```
    load    rx            => ra
    load    ry            => rb
    loadI   0             => rc
1:  add     ra   , rc     => rc
    addI    rb   , -1     => rb
    cmp_LE  rb   , r0     => rd
    cbr     rd            -> L2   , L1
2:  store   rc            => rz
```

Fig. 6.6: Editing Code in the Simulator



Share Simulation ×

Link

http://ayrtonmassey.com/proj/?simulator&framework=REACHING_DE

Close

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.

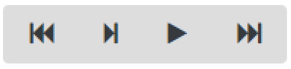


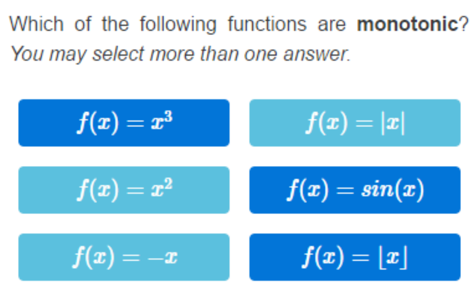
Fig. 6.8: Simulator Controls

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:

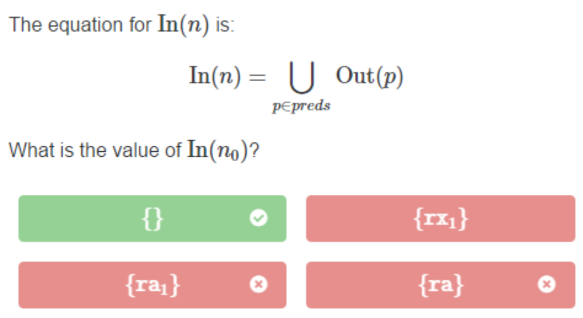
- The order in which answers should be presented (or shuffled).
- Whether a question allows multiple answer choices.
- Which answers should be revealed upon selecting an answer, if any.
- Which answers should be disabled upon selecting an answer, if any.

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



Which of the following functions are **monotonic**?
You may select more than one answer.

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



The equation for $\text{In}(n)$ is:

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

What is the value of $\text{In}(n_0)$?

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.

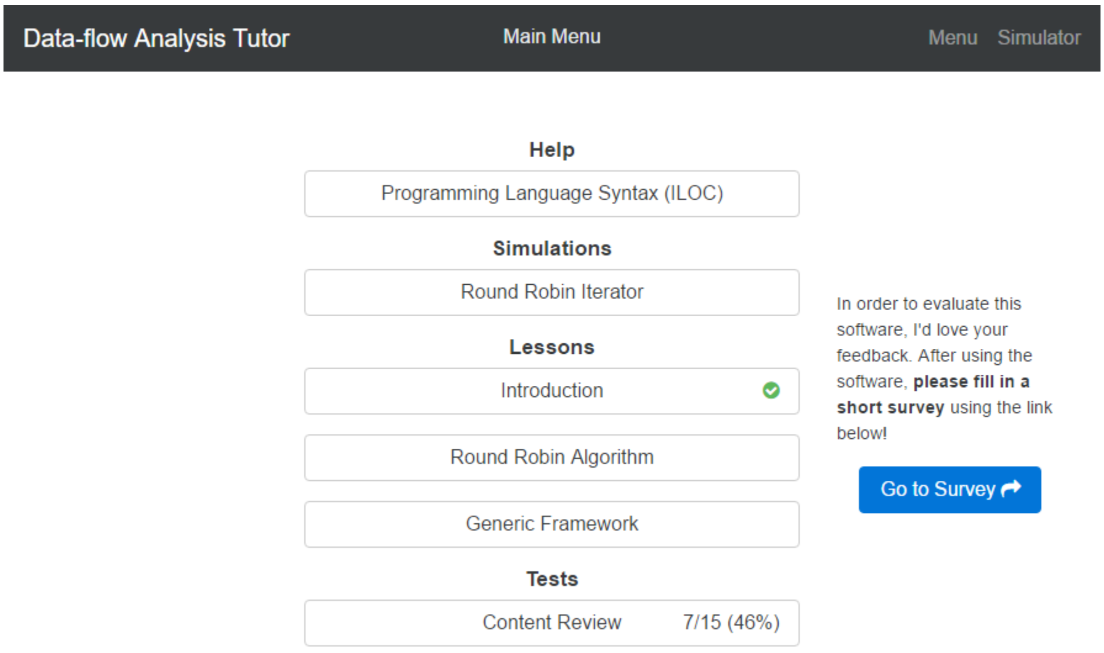
A combination of the Bootstrap design framework, jQuery, Handlebars and MathJax libraries were used to produce the overall look and feel of the UI. The latest development version of Bootstrap (v4-alpha-2) was used due to the addition of CSS `flexbox` support – whereas the current version (3.5.6) uses `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



In order to evaluate this software, I'd love your feedback. After using the software, **please fill in a short survey** using the link below!

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.

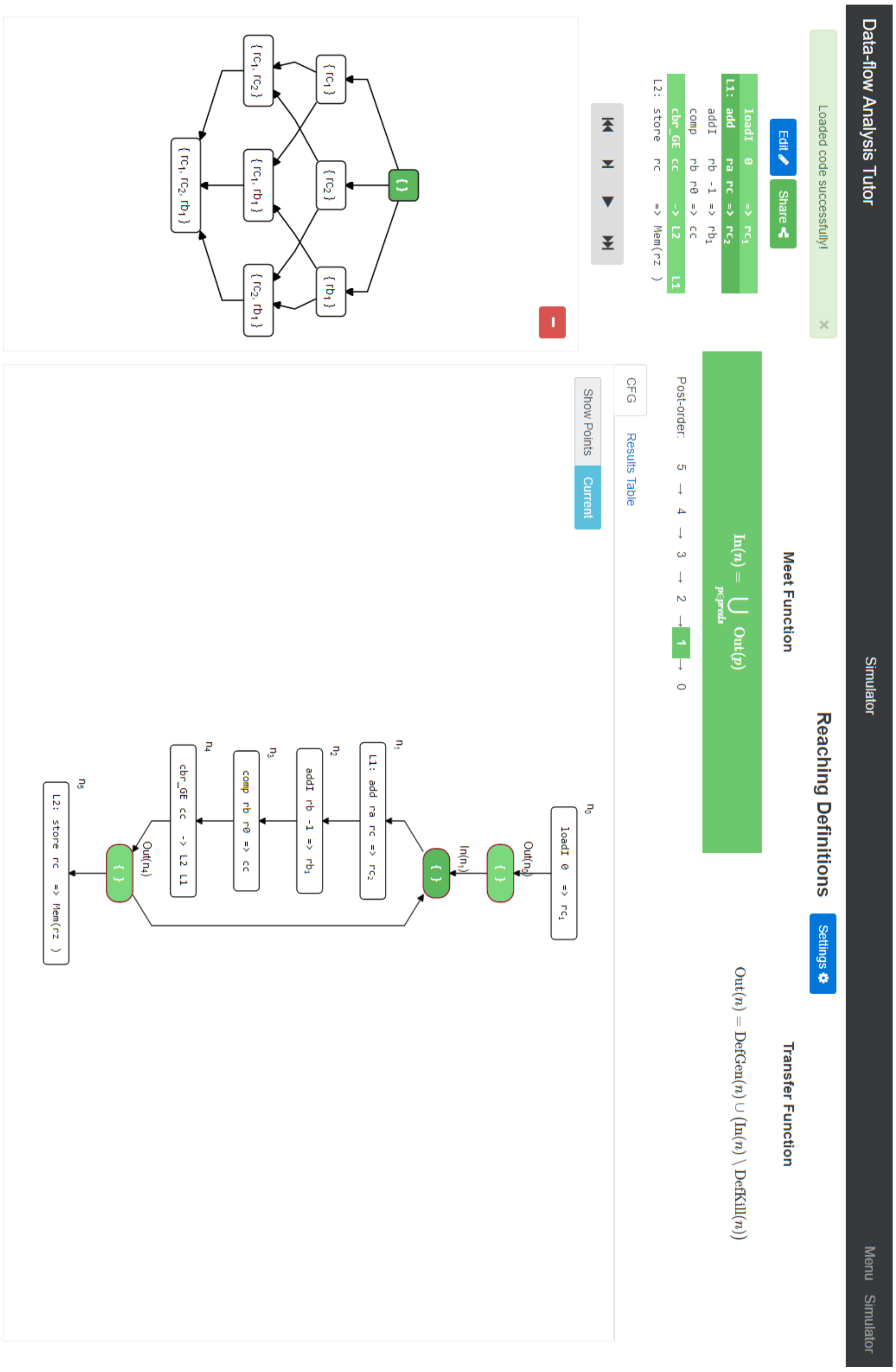
6.5.4 Tutorial Interface

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.



```
    loadI 0      => rc1
L1: add   ra rc  => rc2
    addI  rb -1  => rb1
    comp  rb r0  => cc
    cbr_GE cc    -> L2  L1
L2: store rc     => Mem(rz )
```

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

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

Post-order: 5 → 4 → 3 → 2 → 1 → 0

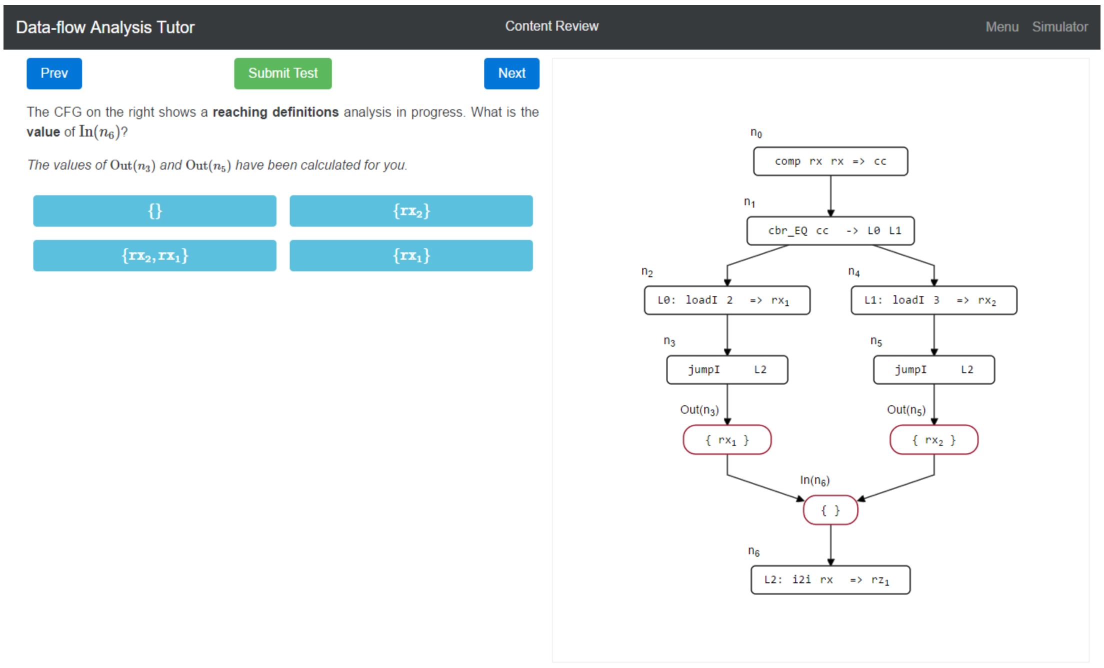
Fig. 6.12: The simulator interface.

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.

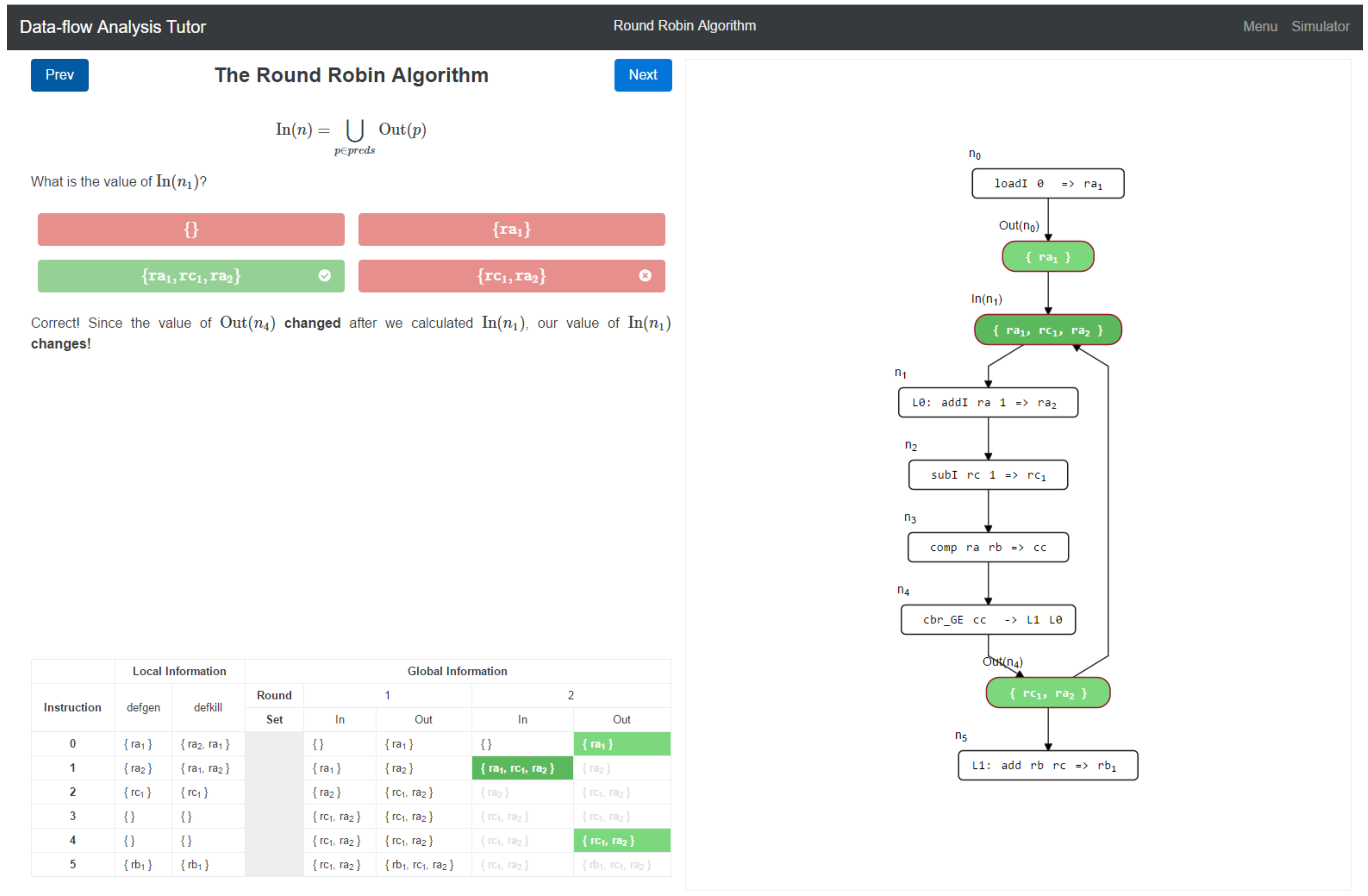


The CFG on the right shows a **reaching definitions** analysis in progress. What is the **value** of $\mathrm{In}(n_6)$?

The values of $\mathrm{Out}(n_3)$ and $\mathrm{Out}(n_5)$ have been calculated for you.

$\{\}$	$\{\mathbf{rx}_2\}$
$\{\mathbf{rx}_2, \mathbf{rx}_1\}$	$\{\mathbf{rx}_1\}$

Fig. 6.13: An interactive test.



The Round Robin Algorithm

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

What is the value of $\mathrm{In}(n_1)$?

$\{\}$	$\{ra_1\}$
$\{ra_1, rc_1, ra_2\}$ ✓	$\{rc_1, ra_2\}$ ✗

Correct! Since the value of $\mathrm{Out}(n_4)$ **changed** after we calculated $\mathrm{In}(n_1)$, our value of $\mathrm{In}(n_1)$ **changes!**

	Local Information		Global Information				
			Round	1		2	
Instruction	defgen	defkill	Set	In	Out	In	Out
0	$\{ra_1\}$	$\{ra_2, ra_1\}$		$\{\}$	$\{ra_1\}$	$\{\}$	$\{ra_1\}$
1	$\{ra_2\}$	$\{ra_1, ra_2\}$		$\{ra_1\}$	$\{ra_2\}$	$\{ra_1, rc_1, ra_2\}$	$\{ra_2\}$
2	$\{rc_1\}$	$\{rc_1\}$		$\{ra_2\}$	$\{rc_1, ra_2\}$	$\{ra_2\}$	$\{rc_1, ra_2\}$
3	$\{\}$	$\{\}$		$\{rc_1, ra_2\}$	$\{rc_1, ra_2\}$	$\{rc_1, ra_2\}$	$\{rc_1, ra_2\}$
4	$\{\}$	$\{\}$		$\{rc_1, ra_2\}$	$\{rc_1, ra_2\}$	$\{rc_1, ra_2\}$	$\{rc_1, ra_2\}$
5	$\{rb_1\}$	$\{rb_1\}$		$\{rc_1, ra_2\}$	$\{rb_1, rc_1, ra_2\}$	$\{rc_1, ra_2\}$	$\{rb_1, rc_1, ra_2\}$

```
n0  loadI 0 => ra1
n1  L0: addI ra 1 => ra2
n2  subI rc 1 => rc1
n3  comp ra rb => cc
n4  cbr_GE cc -> L1 L0
n5  L1: add rb rc => rb1
```

Fig. 6.14: An interactive tutorial.

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.

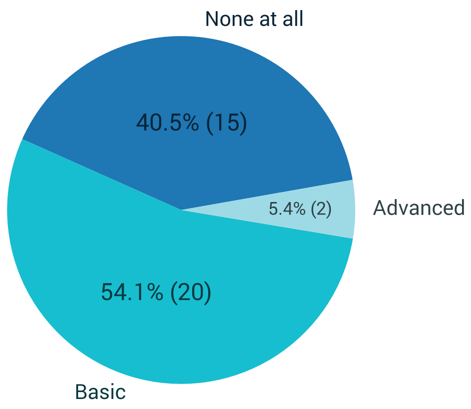


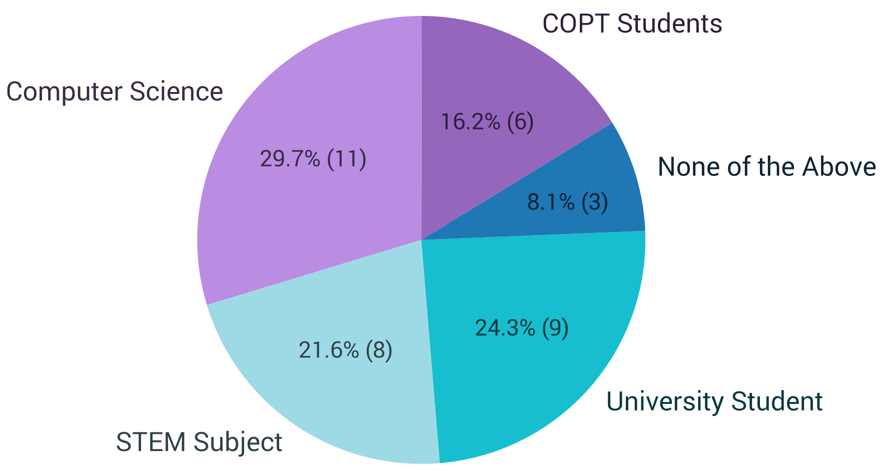
Fig. 7.1: Distribution of User Subject Experience

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.

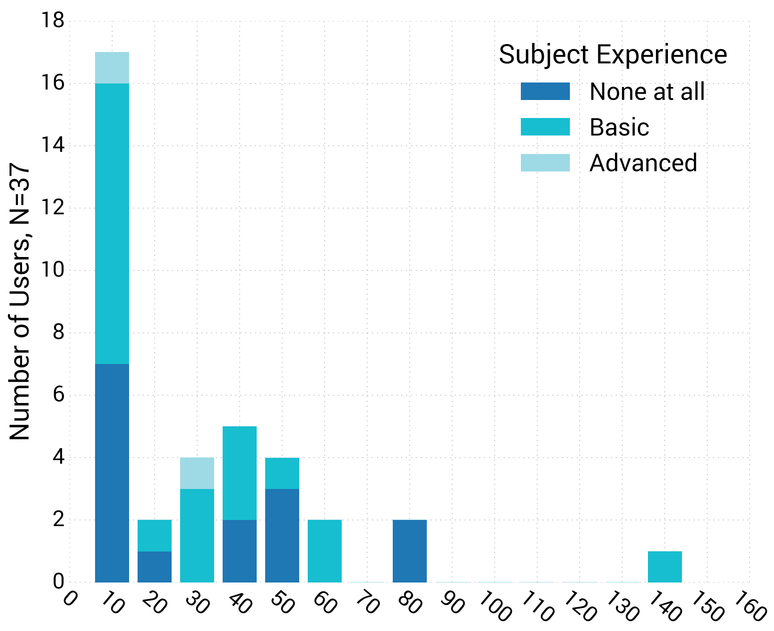
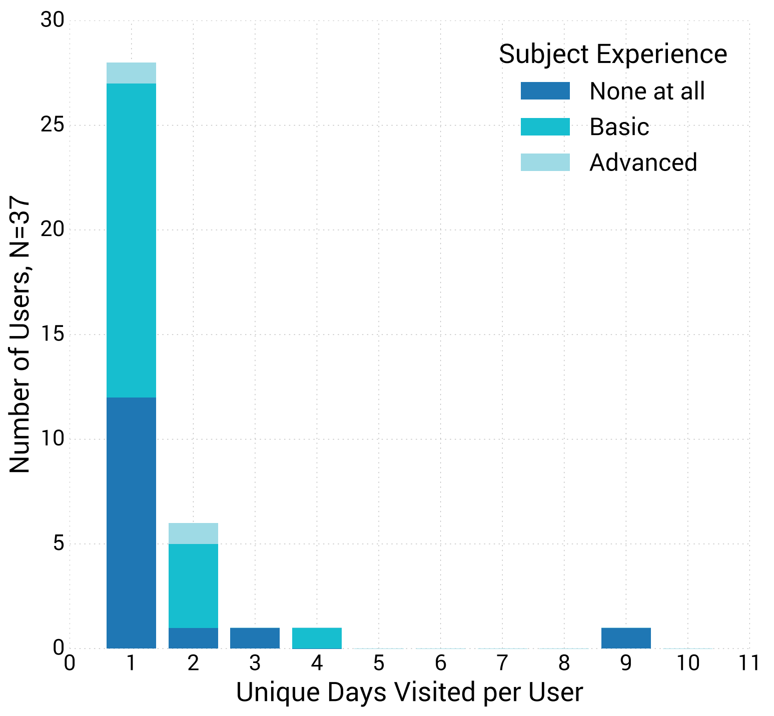


Fig. 7.3: Distribution of Time Spent Using the Application per User

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.

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.

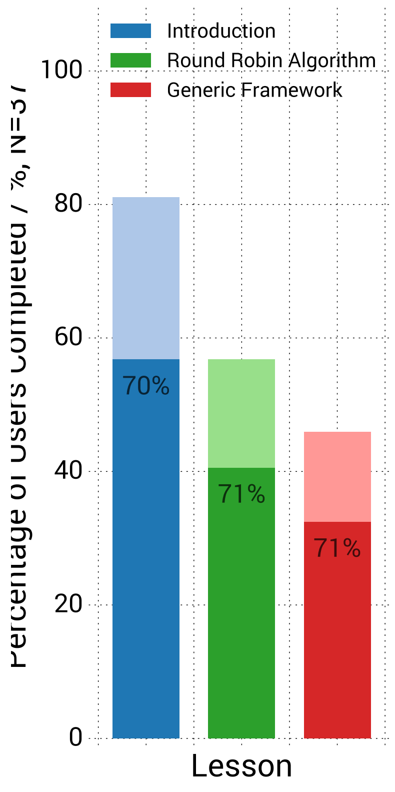
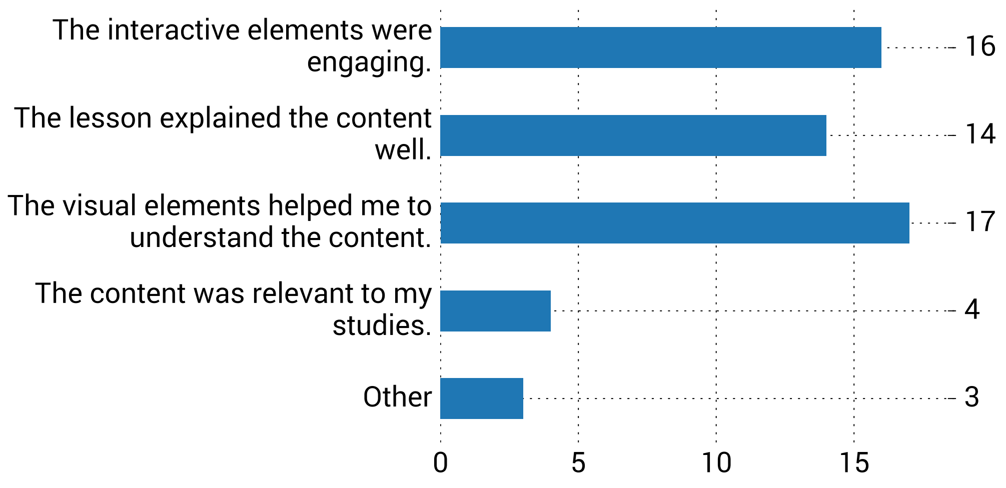


Fig. 7.5: Lessons Completed by Participants

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.

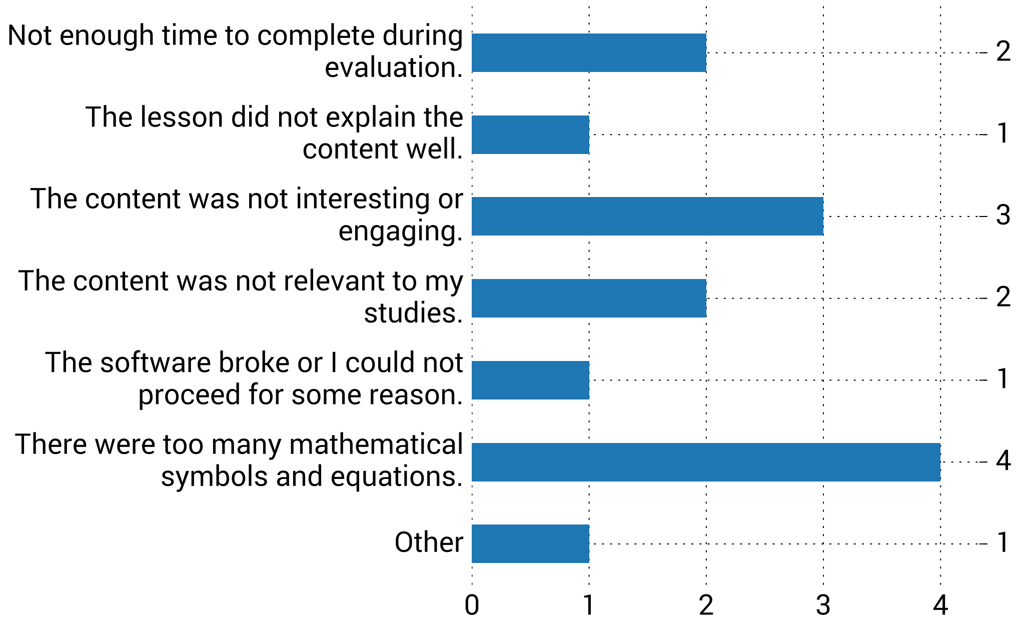


Fig. 7.7: Reasons Provided for Abandoning Lessons (N=25)

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

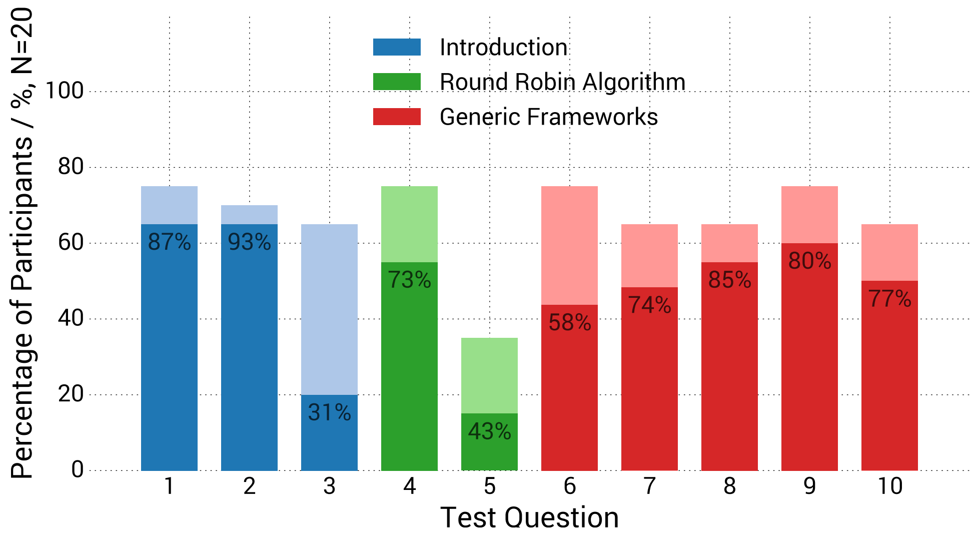


Fig. 7.8: Attempts & Answers to Test Questions

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.

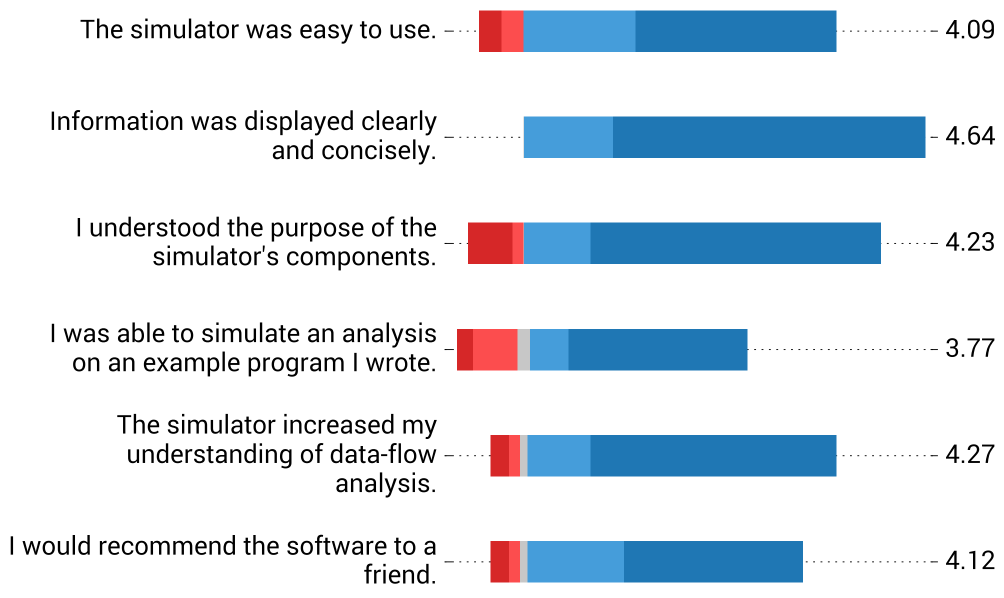
Of course, these results are only an indication. It is highly possible that only academically-gifted participants took the test or that the questions covered content with which they were already familiar. However, the results are well above the expected average and optimistically show that the system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

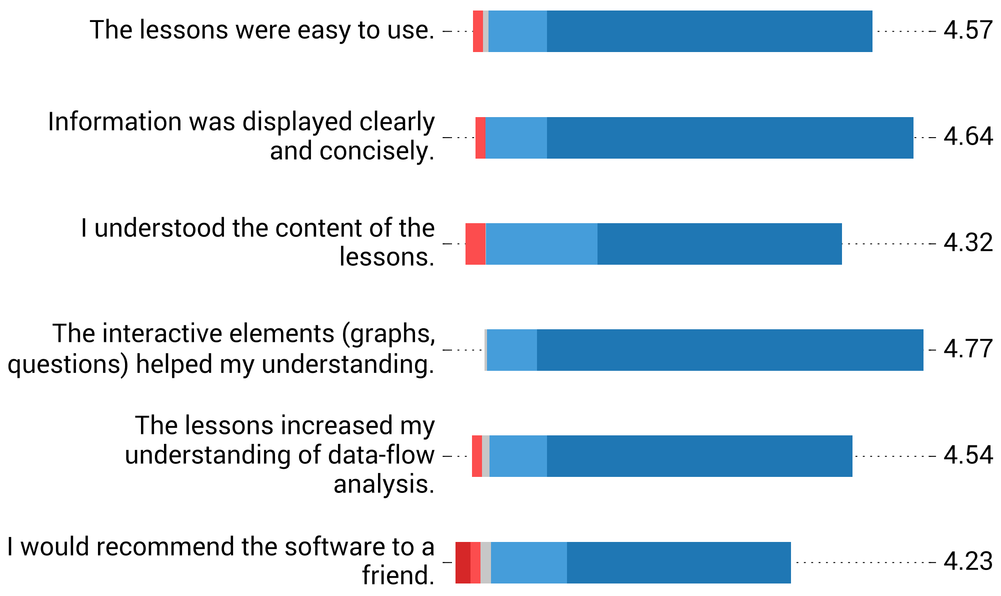
Fig. 7.9: Opinion of the Round-Robin Simulator, N=22



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 & 7.11.

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

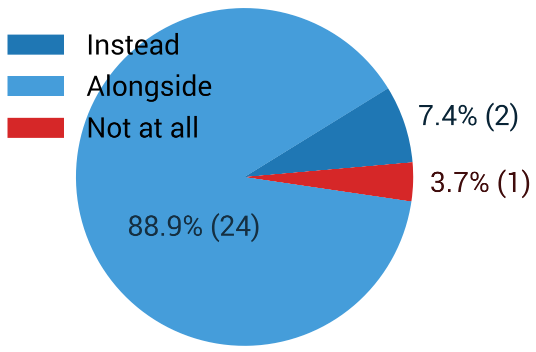
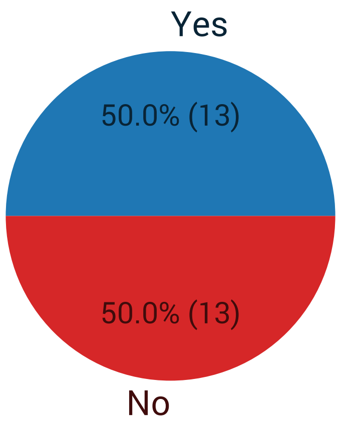


Fig. 7.12: Opinion on Future Use, N=27



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.

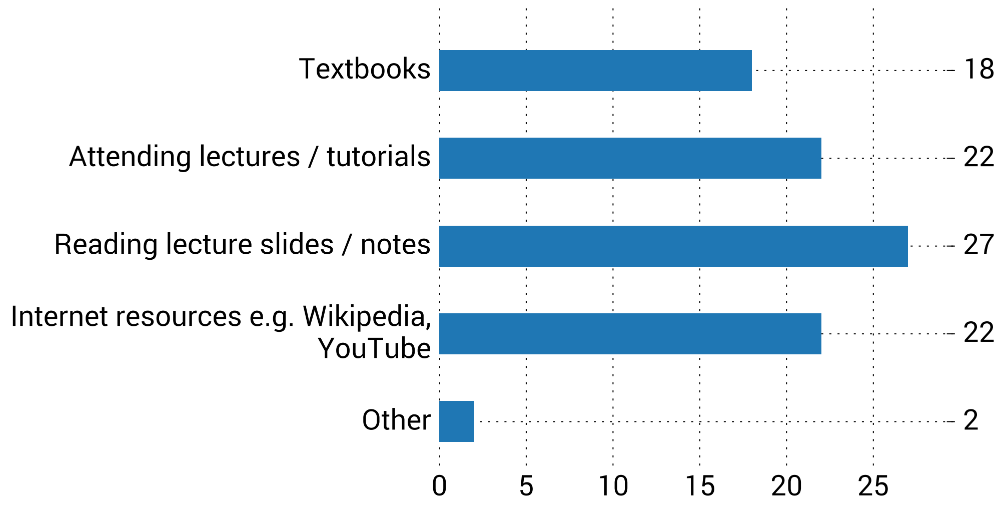


Fig. 7.13: Methods of Study, N=27

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 (§6.5.2), 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.

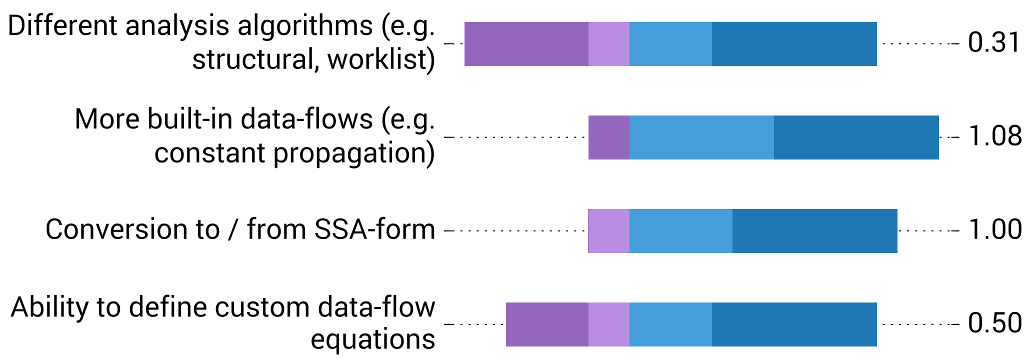
7.5 Future Developments

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.