Visualizing Compiler Passes with LastPass

Paulette Koronkevich
pletrec@cs.ubc.ca

Braxton Hall
braxtonh@cs.ubc.ca

Jonathan Chan
jcxz@cs.ubc.ca

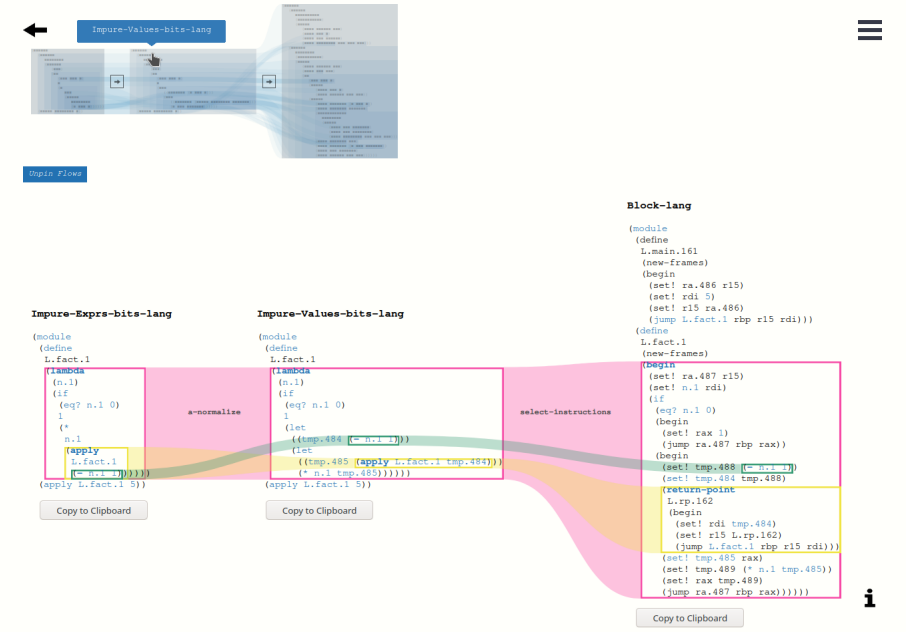


Figure 1: The main screen of LastPass, with an overview at the top and a detailed view at the bottom.

Abstract

In UBC's upper-year compilers course CPSC 411, students are tasked with adding features to a compiler every week. The compiler is comprised of multiple passes over some source program to gradually transform it to an assembly program. Unfortunately, students currently do not have a good tool to explore the compiler and understand how certain programs may be compiled. We present LastPass[1], an interactive multi-pass compiler visualization tool for the CPSC 411 compiler. Using our tool, students can gain an intuitive understanding of the structural transformations that the compiler performs, trace subexpressions through compilation, and interactively examine compilation on an a pass-by-pass level.

1 Introduction

A compiler from a high-level language down to an assembly-like language traditionally involves a sequence of compiler passes, each of which performs a specific transformation on an input program to produce an output program for the next pass. In an undergraduate compilers course, the goal is to understand why these passes are necessary and how each pass is constructed. One difficulty in the learning process is that the compiler pass implementation itself may be too abstract to gain an intuitive understanding of its purpose, while merely reading the input and output programs of the pass does not reveal much about the transformation.

In this paper, we present LastPass, a multi-pass compiler visualization for UBC's upper-year compilers course CPSC 411 that aims to elucidate the connections between input and output program substructures in each compiler pass. It also aims to provide an intuitive understanding of the kind of structural transformations that compilers do. Given a particular input program, it presents intermediate and final programs – the results of multiple stages of compilation – side by side, with interactive flows between related components of input and output programs.

We limit the scope of this project to only two particular fundamental compiler passes: `a-normalize` and `select-instructions`, described in detail in Section 2. Furthermore, as this is primarily a pedagogical tool, it focuses on effective visualization of relatively small input programs, as opposed to existing visualizations which focus on large source code files or optimizing compiler passes.

1.1 Motivation

In the fourth week of CPSC 411: *Introduction to Compiler Construction*, a student and her two project team-

[1]formerly SecondPass[2]
[2]formerly FirstPass[3]
[3]formerly Untitled Compiler Pass Visualization

mates are tasked with introducing control flow to their compiler. As with every week, she looks at the language specifications introduced in weekly batches, selects an arbitrary 7 of the 21 tasks presented, and starts working. Unknown to her at the time of task selection, two of her selected tasks amount to *half* of this week's work for the entire team (when measured in hours of effort or lines of code). Oblivious, she continues work on her team's compiler.

One large task has her scrolling up and down on the course website, comparing the definitions of two languages that define the input and output of a particular compiler pass. She attempts to build a mental model of what the transformation is doing by looking at the static definitions one at a time. She also uses a tool called the Interrogator[4] to assist her in building this mental model. This tool allows her to query any concrete test program through the reference implementation of the particular pass. However, she discovers that the tool erases her original queried test program from the textbox, and provides the transformed output below the textbox. The output has no interactive elements, and she is forced to make comparisons between the test program and output in her head.

The complexity of the compilers in CPSC 411 grows with each passing week. Debugging equates to going back to the language specifications and performing manual language transformations before comparing them to her generated output. The student can also use the Interrogator once again to query certain test cases and trace the input and output of each transformation in the entire compiler. However, seeing the large amount of static textual output below the textbox, she quickly gets lost in which programs correspond to inputs and outputs of certain passes. She also has no way of knowing which generated assembly instructions correspond to which expressions in her queried test program, and is forced to make the connections in her head.

Building a compiler is hard enough already without misestimating the distribution of work, poring over scattered, static language definitions, or getting too few examples of how to make it *right* in the exact situations where students' bugs arise.

With LastPass, we aim to reduce these student frustrations by providing a tool that visualizes the programs before and after transformation to effectively present what a compiler is doing – without directly presenting the solution. As domain experts who have taken undergraduate compilers courses (including CPSC 411) and are intimate with their struggles, and who have worked on other compilers as well, we have the experience and the expertise to identify the relevant problems and propose an effective solution.

We begin in Section 2 with a description of the domain details relevant to the CPSC 411 compilers course. Section 3 discusses relevant work in visualizing code and compilers, and how existing solutions fall short of our goals. We then outline the primary tasks students are expected to complete throughout the course in Section 4, as well as an abstracted representation of programs and compiler passes which we manipulate in Section 5. We introduce our visualization and discuss its design in Section 6, then detail the implementation in Section 7. Section 8 examines possible use cases of LastPass and results from informal user studies. The strengths and limitations of LastPass and future work are discussed in Section 9. Finally, we present our development schedule in Section 10, and conclude in Section 11.

[4] https://www.students.cs.ubc.ca/~cs-411/2019w2/a10-interrogator.cgi

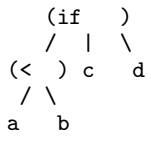
2 Background

In CPSC 411, a compiler begins with a *high-level program* and ends with an *assembly program*. But it does not compile directly from the former to the latter in one go, as a *single-pass* compiler would. Rather, it is a *multi-pass* compiler, divided into many *compiler passes* which sequentially compile a program in a high-level language to programs in several *intermediate languages*. The program that a compiler pass transforms is called the *source program* and its output is the *target program*. These programs are often written in different languages, whose syntax can be specified by a *BNF grammar*. The process of a multi-pass compiler is summarized in Figure 2.

2.1 Programs

A program begins as a string. Depending on the BNF of the language it is written in, it then gets parsed into an abstract syntax tree (AST). A tree is a special case of a graph – specifically, a directed, acyclic graph, where edges point to child nodes, and each child node has at most one parent. This is the representation of a program that a compiler handles.

In the CPSC 411 languages, all programs are *expressions*, themselves containing further subexpressions. In the AST representation, each expression is a node, and subexpressions are its child nodes. For instance, given the program text `(if (< a b) c d)`, we can form the following tree (with directed edges implicitly pointing downwards):



We can see that nodes consist of their program text, which we call the *prefix* for non-leaf nodes (`if` and `<` in the example above), or the *value* for leaf nodes (`a` through `d` above). Note also that the children of a node have an order; `(< a b)` is certainly not the same thing as `(< b a)`.

2.2 Compiler Passes

A compiler pass involves the ASTs of a source program and a target program. Starting with an arbitrary source AST, it transforms it into a target AST by recursively inspecting all nodes of the AST. Each source node may produce zero or more target nodes.

Given some specific, *concrete* source AST, the compiler pass produces a concrete target AST, as well as associations between source and target nodes due to compilation. We refer to the collection of these associations as a *concrete compiler pass*. As an example, we show part of the concrete compiler pass between the

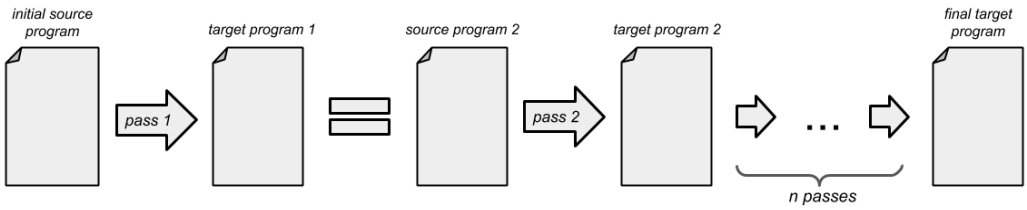


Figure 2: Overview of a multi-pass compiler. The initial source program gets transformed by the first pass into the first target program, which is equivalent to the source program of the second pass. The second outputs a second target program, which becomes the source program for the next pass. This process repeats until the final target program is reached. The intermediate source programs are often written in an intermediate language (IL).

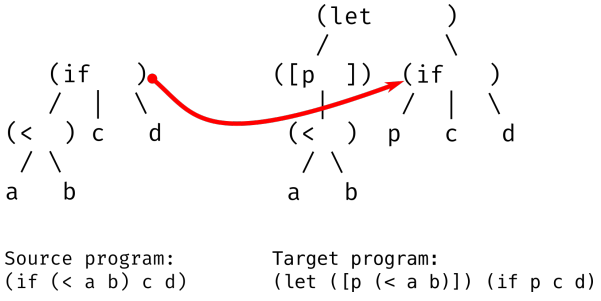


Figure 3: A source AST and a target AST, presented using the same visualization in Subsection 2.1. The thick, red arrow is one of the many inter-AST edges that represent a concrete compiler pass. The other edges of the concrete pass are omitted from this example to prevent occlusion.

source program `(if (< a b) c d)` and the target program `(let ([p (< a b)]) (if p c d))` in Figure 3.

As mentioned, we focus on two specific compiler passes:

1. **a-normalize**, which sequentializes necessary computational steps and makes control flow explicit; and
2. **select-instrucitons**, which adds further information necessary for sequential assembly programs, such as converting function calls to jumps.

3 Related Work

We separate related work into three subsections: visualizations of internal compiler implementations for educational purposes, visualizing compiler optimizations for correctness, and visualizing transformations over source code directly. Transformations over source code include compilation and user changes to source code (i.e. edits for correctness). We find that our solution more closely matches transformations over source code directly, rather than visualizing compiler implementations or optimizations.

3.1 Visualizations of Compiler Implementations

Visualizing compiler implementations for educational purposes has been studied in the field of computer science education for many decades, dating back to the late 1980s. One of the first instances of this was the University of Washington Illustrated Compiler (ICOMP) (2). ICOMP was a tool in conjunction with the course compiler Mplzero, which students had to understand and modify throughout the course. ICOMP allowed the students to visualize the data structures used internally by the compiler (such as an AST or finite state machine) and updated the visualizations at various breakpoints during compilation. Multiple windows and highlighting linked the visualization with the tabular format of the data structure, as well as highlighting the source program text, if applicable. The authors specified that the visualization worked over "moderately large source programs" given by the user. The visualization enabled a better understanding of the internal data structures used by the compiler, and how they transformed through the compilation process.

VAST (1) is another tool for visualizing compiler instrumentation. This tool is specifically designed to aid students' understanding of parsing source program text into an AST. A student sees the textual representation of the source program highlighted as the parser steps through the program text. A corresponding AST is constructed and displayed as a triangular vertical node-link diagram. The authors specify that they expect the generated trees to be "big and without any fixed structure (symmetry, width or height)". As a result, VAST has a larger window showing a focused subtree and a smaller overview window with the entire AST.

While the overall goal of our visualization is similar to both VAST and ICOMP, there are some key differences. In CPSC 411, the students build their own compiler, so we cannot include the source code in our visualization as ICOMP does. Additionally, this compiler does not have to perform any parsing to create an AST. Furthermore, the ICOMP compiler is a single-pass compiler, whereas the CPSC 411 compiler is a multi-pass compiler.

3.2 Visualizations of Compiler Optimizations

VISTA (8) is a system for interactive code improvement, specifically for the domain of embedded systems applications. VISTA works over assembly and Register Transfer Language (RTL) programs. It displays source program instructions in squares representing blocks with arrows between them indicating control flow. The left-hand side of VISTA displays a list of transformations that can be performed, as well as buttons labelled with arrows that allow the user to step

through applying a transformation. As the user steps through a transformation, the instructions that change are highlighted.

CCNav (4) is another tool for visualizing compiler optimizations in assembly code. CCNav displays the source code text directly beside the disassembled binary code text, along with multiple other views of the call graph and function inlining. Since a single line of source code may span multiple lines of assembly code, CCNav has a separate window for displaying a highlighted source code line with its corresponding assembly code without context, that is, not in the view with both programs side by side.

In VISTA, transformations are done in place, and changes between transformations are highlighted in the transformed program. To prevent the user having to step back and forth to see the program changes, our solution displays the programs side by side for easier comparison, like in CCNav. Since our solution focuses on passes that perform small changes on the source program, we will not encounter the issue of a single line of code spanning multiple lines. Since CCNav focuses on visualizing compiler optimizations, there is a greater focus comparing the source code to the generated assembly, rather than the intermediate representations.

3.3 Visualizations of Transformations over Source Code

MieruCompiler (5) is an educational compiler with similar goals to ICOMP. However, MieruCompiler also visualizes the source code text and assembly code text, and implements "horizontal slicing". Horizontal slicing is highlighting a particular expression in the source program text along with the corresponding compiled code in the target program text, and vice versa. The source program text and assembly text are displayed side by side. However, MieruCompiler compiles a subset of C directly to assembly in a single pass, so there is no visualization or horizontal slicing of intermediate languages.

Code Flows (7) visualizes detailed changes to source code for understanding fine and mid-level scale changes between C++ files. This visualization allows programmers to get a global overview of the changes to a source code file as well as find where code may have been deleted, split, or merged. Versions of a file are displayed as horizontally mirrored icicle plots side by side, with paths between matching fragments of code. These paths are shaded as tubes and coloured cyclically to better differentiate between them.

Vis-a-Vis (3) is a "meta-visualization" linking the source code of a visualization algorithm to the generated visualization. It enables comparison of the visualizations generated by the algorithm as the user edits the source code. When hovering over a visualization in a certain state of the algorithm, a tooltip appears showing the differences between the source code at that state compared to the current state.

These projects are the most similar to our solution. However, the MieruCompiler does not trace the flow between expressions in the source program to the compiled program, but uses highlighting instead. This makes it difficult to get an overview of the changes made to the source program. In contrast, Code Flows does visualize the flow between expressions in different versions. However, Code Flows is better optimized for larger programs, as it does not display the source program text directly. The visualization also is static, and does not allow for selecting particular expressions to track their flow. Vis-a-Vis is interactive, but heavily geared towards visualization algorithms and comparing visualizations across versions of the algorithm.

4 Task Analysis

For each CPSC 411 assignment, student groups of three are given new language features to implement. They must augment their compilers to support these features by modifying compiler passes from previous assignments and/or by adding new passes. As an aid for the assignments without revealing exactly how the compiler passes are implemented, LastPass takes a source program and provides a visualization for each compiler pass. We divide the tasks students can accomplish with LastPass in the following categories:

T1. Identifying and comparing complex compiler passes:

1. Comparing the amount of generated target code to the amount of source code
2. Identifying structural changes between source and target code
3. Estimating the distribution of work needed to implement the compiler passes

T2. Understanding the nature of the compiler passes:

1. Comparing how different kinds of expressions are compiled
2. Identifying the result of compiling subexpressions in context
3. Identifying the result of multiple compiler passes to understand their overall effect

We integrate these tasks into the scenarios presented in Subsection 1.1.

4.1 Dividing the Work

To begin tackling a CPSC 411 assignment, student groups must figure out how to divide the work among themselves (**T1.3**). This requires an understanding of the complexity of the compiler passes. Specifically, a compiler pass that generates more target code might be considered as more complex (**T1.1**), but one that moves code around a lot might be more complex as well (**T1.2**). These can be gleaned from the overview visualization of the compiler passes given some well-chosen source programs. With this knowledge, students are ready to create an informed distribution of labour within their teams.

4.2 Compiler Exploration

Although students are given language definitions of the source and target programs for each compiler pass, the relationship between the source and target languages is not always clear, and may require additional pages of documentation to explain. Furthermore, these language definitions, which come in the form of a BNF grammar,

are static and abstract. This often makes it more difficult to intuitively get a sense of how the textual documentation links to the symbols of the BNF grammar and their relationships across languages.

In order to gain insight about a compiler pass, students must create a concrete source program, map the concrete program back into the abstract BNF grammar, glean the semantic relationship to the target language through the textual documentation, and then map the target language's BNF grammar down to some concrete target program generated by the reference compiler. A student's mental model of a compiler pass is frailly constructed through frequent context switches between a student's IDE, the reference compiler, the textual documentation, and the source and target language specifications, costing students time and efficiency! There are too many links that must be maintained in the student's memory.

Students require a way to explore a compiler pass, browse its transformations, and gain intuition about what the compiler pass *does* (**T2.1**), which might not be evident without also knowing what the pass immediately before or after it does (**T2.3**). This model should reduce the number of contexts that a student must use to build their understanding. A student must also be able to discover semantic information about the compiler pass without compromising a context that provides both concrete source and concrete target programs (**T2.2**).

4.3 Compiler Debugging

As with all complex systems, compiler passes are prone to error. When students have a failing test case, they can refer to LastPass to see what the given test case should produce, but most importantly they understand *why* by following the flow of source expressions. In this way, the visualization is acting as an interactive and executable documentation of the behaviour of a certain pass. Students require a method for querying a reference compiler to gain a deeper understanding how certain complex aspects of passes are implemented (**T2**). Equipped with this better understanding, they are better prepared for creating correct compilers.

5 Data Abstractions

In this section, we describe how programs are represented as trees (ASTs), and how a concrete compiler pass is defined over a source AST and a target AST. We conclude with a description of our dataset and the summary of our abstractions, which is also summarized in Figure 4.

5.1 Program Representation

Just as the compiler manipulates the ASTs of programs, so does LastPass to produce visualizations. Abstractly, our programs are collections of nodes and directed edges. Because we expect students to use our tool to visualize small example programs for the tasks described in Section 4, the cardinality of the nodes and edges are in the dozens.

Each node's program text is a categorical attribute. They also have another categorical attribute, whether it is a leaf node or a non-leaf node, derived from the absence or presence of outgoing directed edges. Finally, as child nodes are ordered, each edge has a sequential attribute indicating its position relative to other sibling nodes. These are summarized in the first two rows of Figure 4.

In the miniature tree visualization in Subsection 2.1, we encode nodes as text, edges using line marks, leaf or non-leaf status by the absence or presence of surrounding parentheses, the directional component of the edges using the vertical position channel, and the sequential attribute of the edges (and by extension, the nodes they point to) using the horizontal position channel. However, this is not the natural representation of programs for programmers. For the CPSC 411 languages, the example program would look more familiar like this:

```
(if (< a b)
    c
    d)
```

This is the style we use in our visualization, which is discussed in detail in Section 6.

5.2 Compiler Representation

Given some source program, what we want to visualize is not the general effect of a compiler, but rather the concrete compiler pass associating source AST nodes with target AST nodes. These correspond to additional directed edges between source and target nodes. To distinguish these edges between nodes of different ASTs from the edges within a single AST, we call the former *inter-AST* edges and the latter *intra-AST* edges. The number of outgoing edges from a single source node for the compiler passes we've chosen does not exceed 1; for the remaining compiler passes in the course, they are unlikely to exceed 2 or 3. Therefore, the cardinality of these edges is in the dozens, similar to the cardinality of the nodes. Again, these are summarized in the second and third rows of Figure 4.

An inter-AST edge connects a source AST node to a target AST node. We can redundantly encode this information as a derived categorical attribute on the AST nodes by assigning a unique tag to the the source node and the target node for every inter-AST edge. Nodes that have no indicent inter-AST edges remain untagged and have some null value for this attribute. This representation is mostly for convenience of implementation.

In the miniature visualization in Figure 3, intra-AST edges are again encoded using line marks with their attributes encoded in the position channel. One inter-AST edge is shown as an arrow (i.e. a line mark with endpoint shapes). In LastPass, we visualize programs as indented text, not as the trees above; this figure merely illustrates how source and target nodes may be connected.

5.3 Dataset and Summary

Given some arbitrary source program, our dataset contains an ordered list of three ASTs for the source program, the intermediate program after `a-normalize`, and the target program after `select-instructions`. Each AST is a tree composed of nodes and intra-AST edges. We also have an ordered list of two sets of inter-AST edges, corresponding to the two concrete compiler passes. Finally, we have an ordered list of three language names and an ordered list of two compiler pass names. The various kinds of data items are summarized

Domain Data	Cardinality	Attributes [Type]	Derived Attributes [Type]
AST node	Dozens	Program text [Categorical]	Leaf/Non-leaf [Categorical]; Tag [Categorical]
Intra-AST edge	Dozens	Position [Sequential]	None
Inter-AST edge	Dozens	None	None
Language names	Three	None	None
Pass names	Two	None	None

Figure 4: A summary of our network data with attributes, cardinality, and mark.

in Figure 4. In general, if we want to consider n compiler passes, then we would have n pass names, $n + 1$ language names, and $n + 1$ ASTs.

6 Solution and Design

Our solution is to create an interactive visualization, where students input an arbitrary source program to see an overview of the compilation process. Providing an overview of the compilation process allows students to pinpoint a particular passes of interest (**T1**). An overview is not sufficient to understand the effects of certain passes; to study and understand a particular pass, a student can also choose to view it in detail (**T2**).

We elaborate on the overview and detailed views in the following subsections. However, we first address visually encoding program ASTs as code that students are used to seeing. A given node is displayed as its program text with the visual encoding of its child nodes either to the right on the same line, separated by spaces, or below on separate lines with indentation. In other words, the intra-AST edges and their ordering are encoded using either only the horizontal position channel with left-right ordering, or the horizontal position channel (as indentation) and the vertical position channel with top-down ordering. The leaf/non-leaf attribute of nodes are encoded by the absence or presence of parentheses before the node's prefix and after the node's children. This corresponds to the conventional arrangement of code programmers use.

6.1 Detailed View

The detailed view, shown in Figure 6, gives students an up-close interrogation of the relationship between two concrete programs on either side of a compiler pass. The detailed view shows the source program and target program of a pass side by side, in their textual representation. The textual representation allows students to directly identify expressions of interest. To facilitate a user's exploration of the relationship between source and target programs, a user can hover over a program expression in the source program and see the corresponding compiled expression in the target language highlighted.

The first prototype for the detailed view included highlighting the source expression text and corresponding target expression text by changing the colour of the text to a new colour. This early experiment can be seen in Figure 5. However, identifying smaller expressions such as single-digit numbers in a large body of program text becomes much more difficult. An explicit connection on screen guides the student directly to the corresponding target program text. Additionally, we wished to eventually support highlighting multiple expressions, and changing the colour of the text in a subexpression would remove the connection of colour that it has with its highlighted parent expression.

Instead, when a student hovers over an expression in the source program, the expression and its children are outlined with a coloured box. In Figure 6 the expression beginning with `lambda` is hovered over. The corresponding expression in the target program, based on the inter-AST edges of the pass, is also outlined. Early prototypes used solid coloured boxes, but putting program text against these coloured boxes reduced the text's contrast and in turn its readability. To combat this, we took inspiration from containment diagrams, and enclosed the text of a node and its subtree with a coloured box outline, giving a visual indicator of which expression is being hovered over. A curve is drawn between the boxes to display the inter-AST edge of the pass. We refer to these connections as *flows*. Displaying connections in this way shows how certain subexpressions are compiled the in context of the larger concrete source and target programs (**T2.2**).

To aid the student in discerning which expressions have hover events, we use text colour (blue and black) to encode a hover event's presence or absence. Blue text indicates program expressions that directly compile to or are directly compiled from an expression in a neighbouring program, and thus allows for a cursor hover to display a flow. Expressions that neither directly compile from any expressions in the source program nor compile to another program are shown in black. These expressions often appear in code added by the final pass, as there are no additional passes afterwards for them to be compiled to.

To compare the compilation of multiple expressions (**T2.1**), students may click to pin a flow. Figure 7 shows an example of pinned and hovered flows. The node prefix text is bolded to indicate that the flow has been pinned. Once a flow has been pinned, the next flow on hover appears in a different colour. The flows cycle through the colours yellow, pink, and green, also seen in Figure 7. These colours, along with the blue for hoverable elements, are together colourblind-safe (6). There is no limit to the number of flows that can be pinned; we discuss how this can be a limitation further in Subsection 9.2.

Allowing multiple expressions to be seen at once presents its own challenges. As two selected expressions may either horizontally or vertically aligned, one's box may occlude another bounding box. We attempted reducing the width of an expression's bounding box according to its node depth to reduce this occlusion. Unfortunately, as the bounding boxes needed to be thin to prevent occluding the program text, we found that we had few pixels to work with. This feature, shown in Figure 8, was removed from our final solution as we believe that the flows add sufficient redundancy in encoding the

Figure 5: A prototype of a detailed view of the `a-normalize` and `select-instructions` passes, with one subexpression in each program coloured in red.

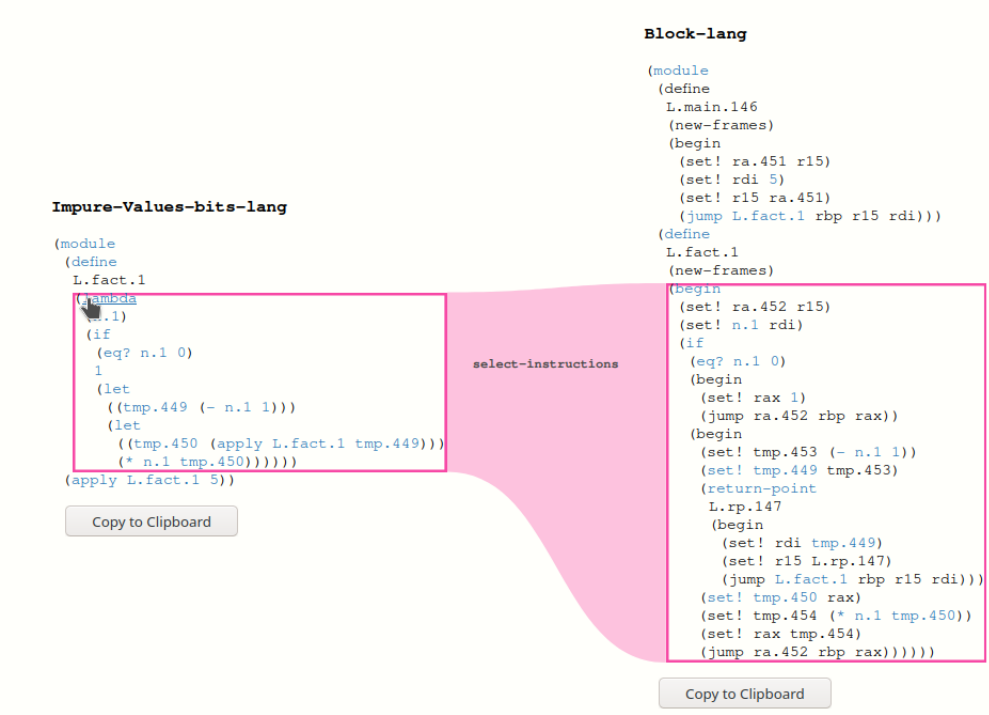


Figure 6: A detailed view of the `select-instructions` pass.

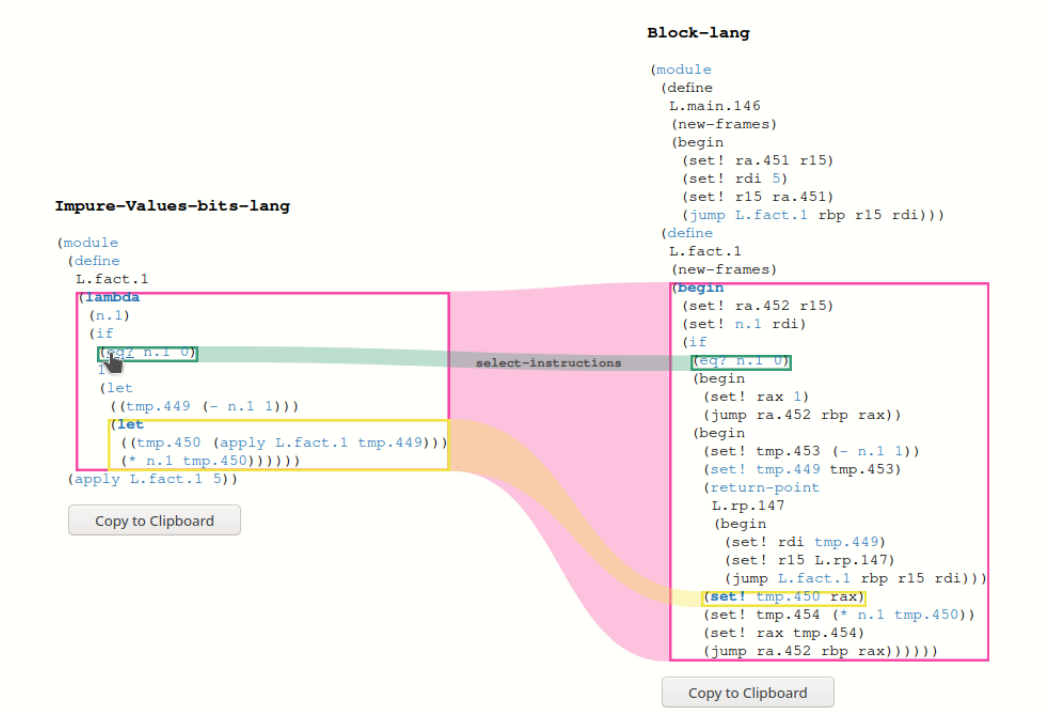


Figure 7: A detailed view of the `select-instructions` pass. Two flows have been pinned (indicated by the bold text parent expression), one flow appears on hover.

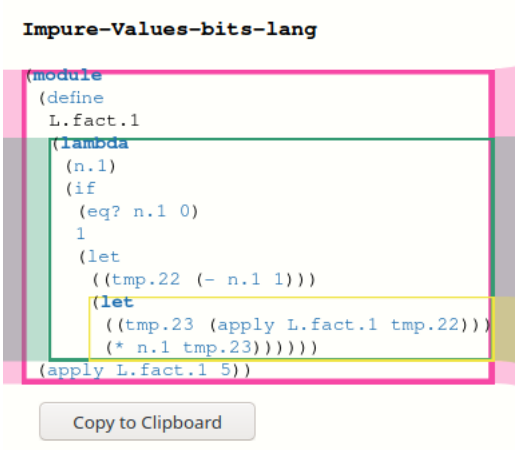


Figure 8: Bounding boxes in the detailed view with variable border thicknesses depending on node depth.

selection of an expression to offset any possible misreading of the visualization due to occlusion.

With multiple flows pinned, one may intersect another. A naïve solution may attempt to discern an ordering of the flows based on the node depth of their corresponding expressions and draw them opaque on top of one another. However, the node depth order may not be preserved from source to target programs in a compiler pass, so an ordering that works for the source program may not work for the target program. To account for this, flows are made to be semi-translucent so all flows can be seen in full in the event of occlusion.

A flow can be unpinned by clicking on the expression again. We also include a button to unpin all flows for convenience.

Finally, instead of clicking on a single pass, the student may select multiple passes. Alternatively, the student may select an arbitrary source and target program to see the effect of multiple passes on the source program. This shows the effect of multiple passes on the source program, with or without the intermediate representations (**T2.3**). The student can continue hovering to see flows and clicking to pin flows, as seen in Figure 9 and Figure 10.

6.2 Overview

The overview, shown in Figure 11, is intended to give students a high-level understanding of the compilation process.

An early design choice was to display programs from left to right, using horizontal position to encode the order in which programs are compiled. This shows the compositional nature of the compiler, i.e that the

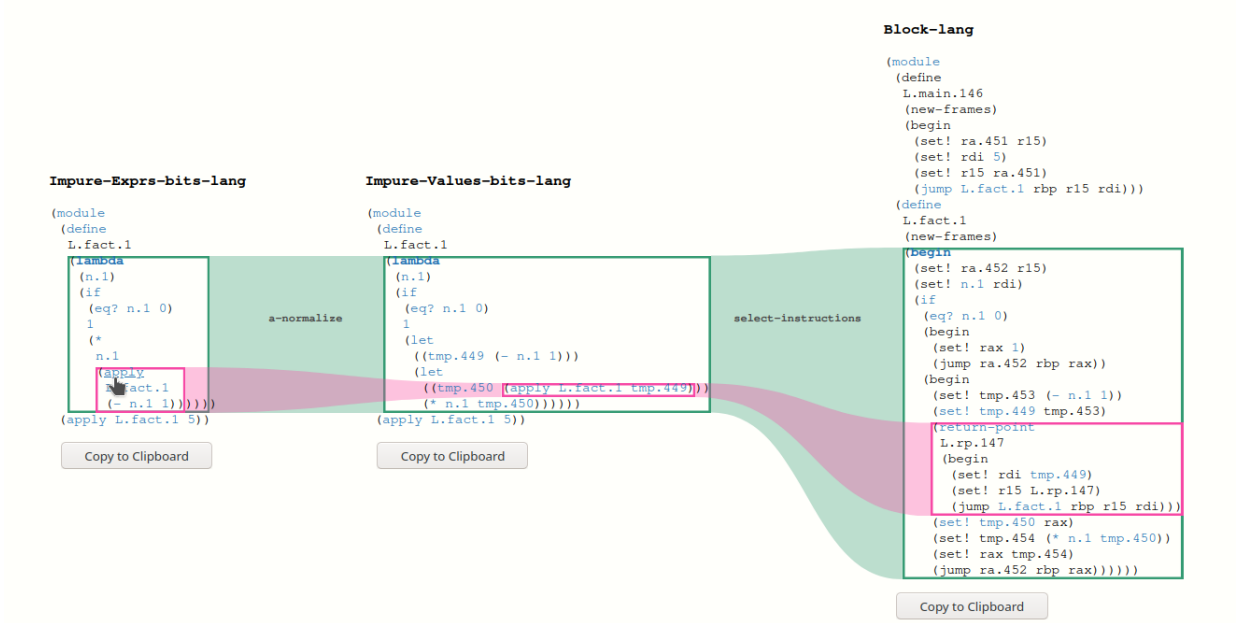


Figure 9: A detailed view of both the `a-normalize` and `select-instructions` passes applied.

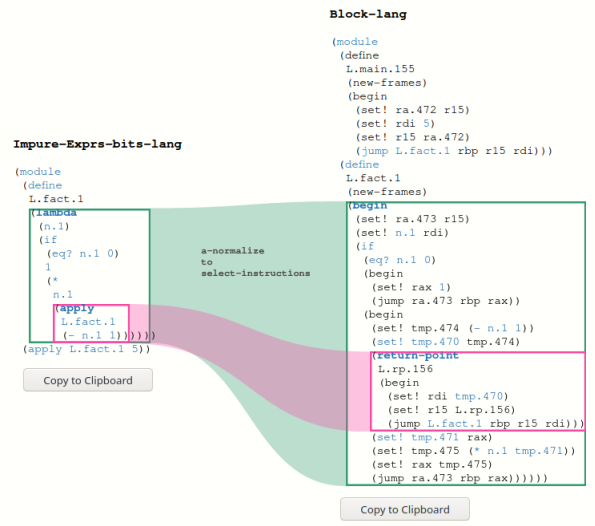


Figure 10: A detailed view of the source program and the target program after the `a-normalize` and `select-instructions` passes.

`a-normalize` pass is applied first to the source program, followed by the `select-instructions` pass. Displaying the programs side by side, positioning them aligned and on a common scale, allows students to compare the lengths of the three programs and determine how much code a given pass has generated (**T1.1**). Finally, the overview also provides students with context of where they are in the compilation process when in the detailed view.

A prototype considered for our overview attempted to communicate the compilation process by highlighting each target program according to what has changed between the source program. A mockup is given in Figure 12. The darker luminance highlights code that has been added to the target program, while the lighter luminance highlights code that has been compiled from the source program of the pass. This overview provides an easy way to distinguish how much code a given pass has generated (**T1.1**), and where it has been generated. However, it fails to show structural changes (**T1.2**).

We then moved to capturing this structural change by showing the pass connections between the programs. This required intentionally putting horizontal space between the programs in the overview to provide screen space to draw connections between expressions in neighbouring programs. The first mockup for this proposed change can be seen in Figure 13.

Building on this prototype, we introduced translucent flows between expressions like those in the detailed view. Overlapping translucent connections had the added benefit of drawing flow-like shapes of increased luminance where groups of expressions moved together, or small squares of increased luminance where flows crossed each other, showing large structural changes.

We also emphasize that the overview is to be used to detect high level structural changes (**T1.2**) rather than understanding the compilation of each individual pro-

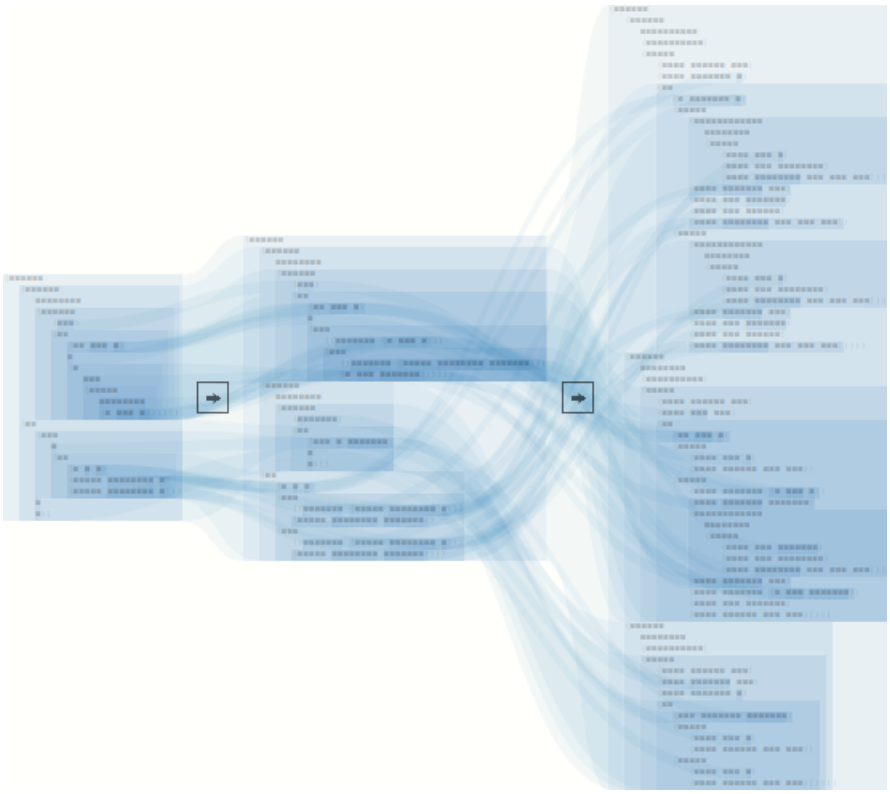


Figure 11: The overview visualization of the two passes.

gram expression. To bring further focus to the flows, we abstract the programs from their textual details. Each node of the program is replaced with a rectangle with the same width, height, and position as the text corresponding to its name. This prevents students from trying to read the smaller text. Showing the abstracted program still allows the students to directly compare the amount of code generated by each pass.

On hover, a text popup displays the name of the language the hovered program is written in. Additionally, an arrow is added between each program to further emphasize the compilation relationship between programs, and displays the pass name in a popup on hover.

If we scale up to more passes, the overview would expand horizontally. There are limitations to this approach, which we discuss in Subsection 9.2.

6.3 LastPass

As seen in Figure 1, LastPass is composed of both the detailed view and the overview presented to the user in a single faceted view. Putting both views on the screen at the same time allows them to be linked. Selecting a program in the detailed view decreases the luminance of the same program in the overview, indicating where in the overall compilation process the detailed view is currently presenting.

Additionally, this allows the overview to double as a control mechanism for the detailed view. When a student is interested in studying a certain pass, they can click on that program in the overview to bring it into the detailed view, and click it again to remove it.

Lastly, an optional sidebar view is revealed when clicking on the hamburger icon, as seen in the upper-right corner of Figure 1. It shows a list of program names and pass names that can be hidden in a collapsible sidebar menu, where each name is listed in order of the compilation process. This provides users with a view of the data where program and pass names are not hidden behind hover events, and users can quickly find a pass by name. Clicking a program name in the sidebar view manipulates the overview and detailed view just as clicking a program in the overview would. Additionally, a checkbox is display attached to each program and pass name, where selection denotes whether that program or pass is selected in the detailed view.

7 Implementation

The architecture of our implementation is divided into two parts: the backend, which contains the compiler; and the frontend, which contains the visualization. LastPass is hosted on the Software Practice Lab's website, `https://se.cs.ubc.ca/compiler-viz/index.html`.

7.1 Backend: Racket Compiler

The backend is written in Racket[5], where source programs are compiled to target programs. Here, programs are represented as *s-expressions*, where the tree structure of ASTs are encoded as lists of either other lists or symbols. For instance, in `(if (< a b) c d)`, the

[5] `https://racket-lang.org/`

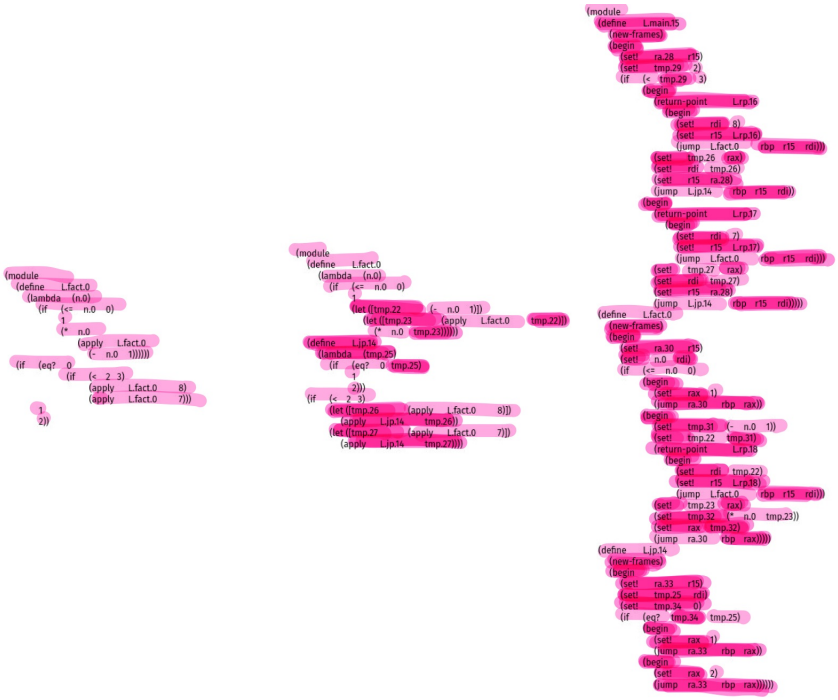


Figure 12: A mockup of an alternative overview visualization, showing the code added by each pass by highlighting in a darker luminance.

expression itself and `(< a b)` are both lists, while `if`, `<`, and `a` through `d` are symbols. Compiler passes take source s-expressions and manipulate them to produce target s-expressions.

We lift the `a-normalize` and `select-instructions` passes from the 2019 Term 2 offering of CPSC 411[6]. We modify these passes so that a unique tag (specifically, a generated symbol) is attached to every node of the source s-expression. Then during each compiler pass, if a target node is produced from some source node with a unique tag, that target node is also tagged with the same tag. Newly generated target nodes are left untagged. If there is a subsequent pass, untagged nodes are uniquely tagged first before the pass is applied. For the previous example expression, the source s-expression might be tagged as (with indenting for legibility):

```
((if . t1) ((< . t2) (a . t3)
                     (b . t4))
           (c . t5)
           (d . t6))
```

while the corresponding target s-expression after `a-normalize` would be

```
(let ([p ((< . t2) (a . t3) (b . t4))])
  ((if . t1) p
             (c . t5)
             (d . t6)))
```

where `let` and `p` have been left untagged for now. The inter-AST edges can then be recovered by connecting the AST nodes between the two programs with the same tag.

[6] https://www.students.cs.ubc.ca/~cs-411/2019w2/

After source programs have been tagged and compiled through all passes, we then re-encode the program s-expressions into JSON representations for ease of manipulation in the frontend.

Finally, there is a lightweight Racket server, which receives requests containing the source program to compile. The program is parsed from a string into an s-expression, tagged, compiled through two passes to produce three program s-expressions in total, then re-encoded as a JSON string, which is sent back as the request response.

7.2 Frontend: Visualization

The frontend is written in TypeScript using axios[7] to perform requests and React[8] for building the user interface. The overview and the detailed view of the visualization itself are both composed of a top and a bottom layer.

The top layer displays the compiled programs using pure React JSX components written from scratch that represent each node. These elements are recursively inserted into the browser document following the structure of a program's AST. With each element, an event listener for the mouseover browser event was added where, upon a cursor hovering over the element, other elements in the document attached to expressions with a matching tag would be passed to a bounding-box and flow-drawing subsystem.

The bottom layer displays the bounding boxes and flows between nodes with the same tag. These bounding boxes and flows are also constructed from scratch

[7] https://github.com/axios/axios/
[8] https://reactjs.org/

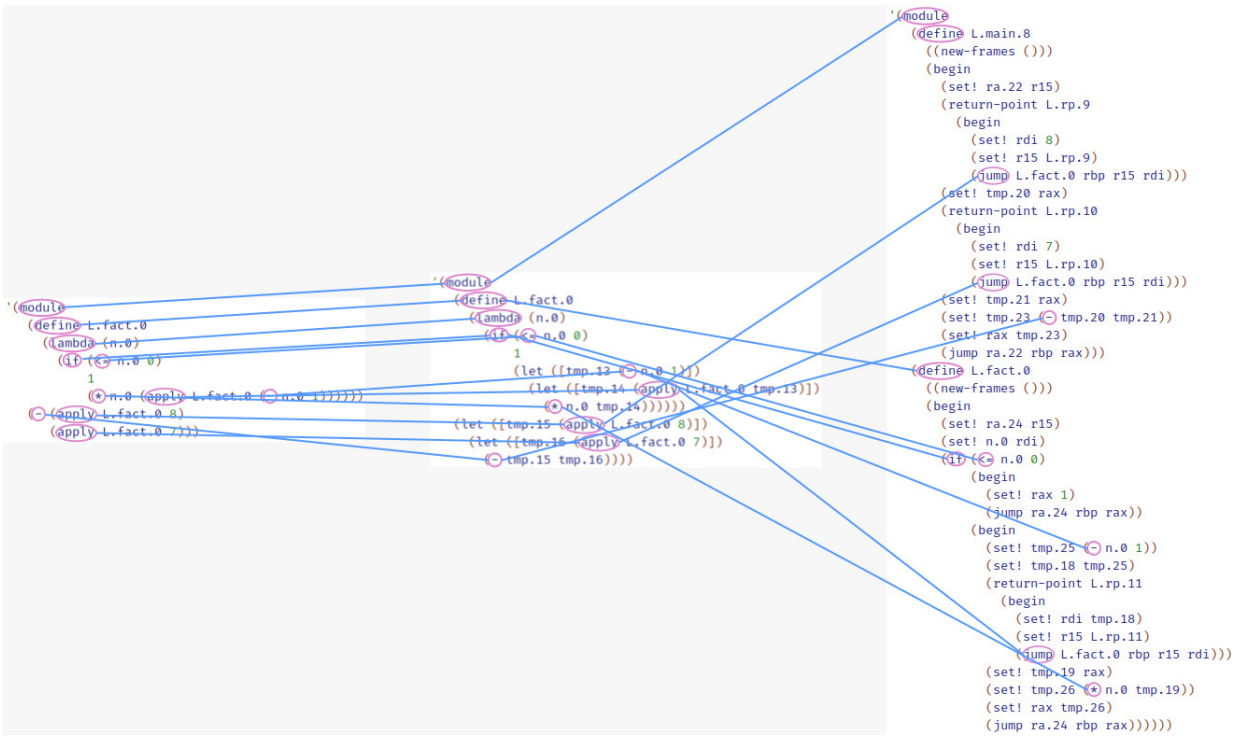


Figure 13: An early prototype of the overview using straight line marks to encode inter-AST links.

as SVG elements and inserted into the browser document using jQuery[9] for convenience. The process of constructing the flows themselves is designed after a jQuery plugin that uses Bézier curves[10].

Additional elements, such as the sidebar and code editor, were implemented by importing the popular npm packages `react-ace`[11] and `react-burger-menu`[12].

8 Results

This section begins with possible scenarios of use of LastPass, and concludes with analysis on some informal user studies conducted with former CPSC 411 students.

8.1 Use Cases

When a student first visits the site where LastPass is hosted, they are presented with an editor where they can input their desired program to visualize, as seen in Figure 14. By default, a program with a top-level function definition for computing a factorial and the application of the factorial function on the number `5` is given.

One possible use case of LastPass is for students to learn how a certain expression is compiled. For example, one assignment in CPSC 411 involves compiling control flow, which means students learn how to compile `if` expressions. A student is interested in understanding how an `if` expression slowly transforms into two separate

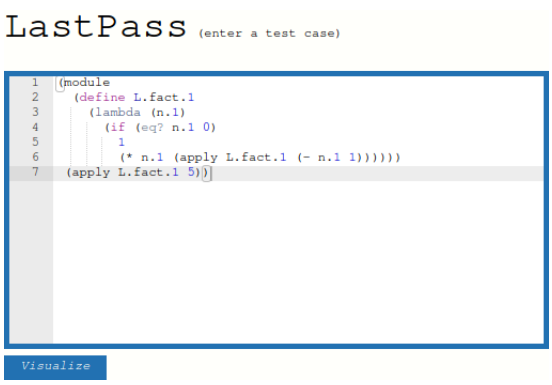


Figure 14: The program input area for LastPass with a default example program.

blocks of assembly code with jumps, and making sure to make note of particularly important passes. In this case, a student may input a small program containing an `if` expression. From the overview, they select passes that include a lot of structural changes, and skip passes that show relatively straight flows, such as `a-normalize`. By selecting these passes in detail, and mousing over the `if` expression in the source program, the student can directly pinpoint the result of compiling this expression. They avoid getting lost in the large textual output program that includes boilerplate for running assembly.

Another possible use case is compiler debugging. A student may have implemented several passes, but finds that the provided test suite fails on certain tests. They

[9] https://jquery.com/

[10] https://www.jqueryscript.net/other/animated-lines-elements-bezier.html

[11] https://www.npmjs.com/package/react-ace/

[12] https://www.npmjs.com/package/react-burger-menu/

input one of these failing test programs, and can check the results of their implemented passes. By mousing over expressions in the source program of the pass, they can gain a sense of the correct transformation their implementation should follow. However, our tool does not let them directly compare against the incorrect output of their pass, forcing them to switch between them and do the comparison in their head. Allowing students to provide their incorrect output for comparison could be a future feature of LastPass.

8.2 End-User Study

We conducted an informal user study of three former CPSC 411 students to survey the perception of LastPass by former members of the target demographic of LastPass. We could not use current students of CPSC 411, as the course was not being offered during the time of this study.

Participants were provided with LastPass and a sample program while their screen was recorded. This was paired with an exit interview probing their thoughts on the system.

Participants recognized several advantages of LastPass over the tools they had used when they were enrolled in CPSC 411.

> *It would definitely make it easier to parse what the reference implementation is doing, because you had to do this in your head before.*

One participant stated task **T2.2** as one which could be accomplished by students given this tool.

> *When I make this change* here*, I can see the effect it has on my output right away, as opposed to spending ten minutes or more collecting text documents [of programs] and running a diff.*

Another echoed task **T2.3**, but in terms of creating test cases.

> *It would be much easier to see which parts of my test case are actually used and how they are used in the language above.*

Unfortunately, participants were less confident that students would be able produce an informed estimate of the distribution of effort across creating multiple passes given only LastPass.

> *I feel like it could be helpful but it could also be misleading. Like* `select-instructions`*, obviously there's a lot going on here [...] versus* `a-normalize` *which I think is more conceptually tricky.*

Finally, users highlighted that the interface for interacting with LastPass left room for improvement.

> *The output is kind of confusing, but after clicking random things it appears it's showing me different passes.*

All three users failed to discover that a flow could be pinned when clicked on, indicating that the user interface could benefit from further iteration. One user suggested that this may have occurred because the tools that they were familiar with had different behaviours upon performing a similar action.

> *I was so used to IntelliJ I was afraid I was going to get swung away if I clicked any of these [expressions].*

9 Discussion and Future Work

We divide this section to individually discuss the strengths and limitations of LastPass, and what we have learned from creating this visualization. We include a discussion of our proposed project schedule along with the actual hours it took to complete, and then conclude with future work.

9.1 Strengths

A significant strength of our tool is showing our students the result of compilation of all subexpressions in a larger program context. Even tiny programs are greatly expanded by the reference compiler, as assembly programs often require a great deal of boilerplate instructions to run. The subexpressions of the original source program quickly get lost in the large amount of final instructions. Additionally, since the compiler sequentializes data flow in the program, many expressions are pulled away from their original context. This transformation makes it even more difficult to pinpoint certain compiled source expressions, as they can end up far from their original source context.

9.2 Limitations

There are three key limitations in LastPass. The first limitation is that the overview may not be a suitable method for determining the complexity of passes. As seen in Figure 11, the `a-normalize` pass does not look as complex as `select-instructions`, as the latter has many overlapping flows. The flows in the `a-normalize` pass are mostly straight, with a few flows indicating that some expressions have been pulled up out of their context. However, this pass requires students to implement it in a programming style called *continuation-passing style*. Students may not have much experience with this programming style, and may therefore find `a-normalize` more difficult to implement than `select-instructions`. We discovered that **T1.3** might not be a feasible task after creating mockups of and implementing our overview.

Another key limitation is scale. We mentioned that additional passes can scale horizontally in our overview, and have ensured the frontend is implemented in a way that enables this. However, thirty-three passes in a row may not be very distinguishable in this horizontal format. Additionally, the detailed view does not restrict the number of passes or programs that one can select. This could lead to students selecting a lot of passes to

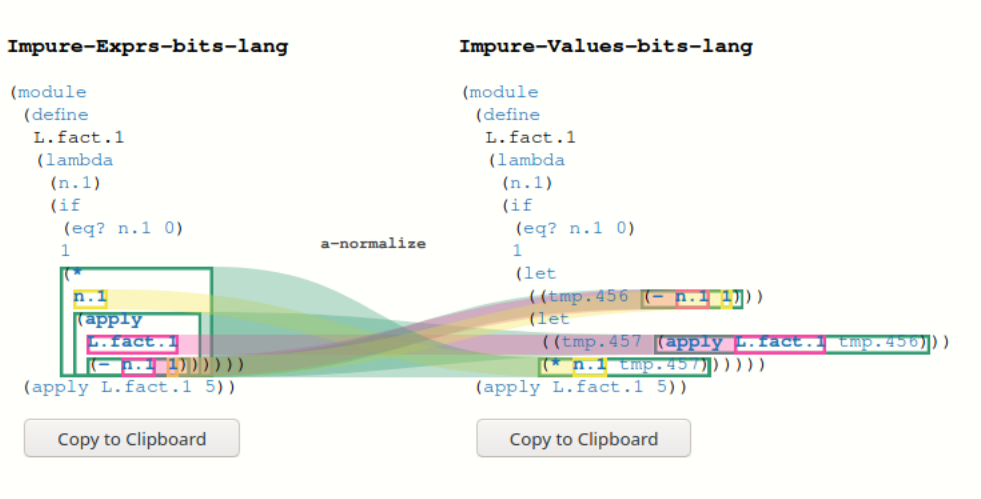


Figure 15: A detailed view of the `select-instructions` pass with seven flows pinned, causing a lot of occlusion.

see in detail, resulting in our visualization running out of horizontal screen space to display all the programs.

A final key limitation is the number of pinned flows. As seen in Figure 15, pinning many flows introduces a lot of occlusion, despite the semi-transparent flows. This makes it difficult to compare different subexpressions, especially subexpressions that are children of larger expressions with a pinned flow. One solution could be to limit the number of displayed flows, but it remains unclear what number to limit it to. Another possible solution is to have inner flows to be an outline of a curve rather than a filled curve; however, we have not explored this option.

9.3 Future Work

There are a few features and ideas that we have in mind but were unable to complete. These range from minor technical features to additional visualization components that need more analysis and consideration.

Flow no-go. Currently in LastPass, if some source AST node does not compile to any further target nodes, a coloured box still appears around the source node, but there are no flows coming out of it, and no indication that this lack of flow is intentional and not an error. To make this clear, there should be some visual representation of the fact that the flow does not go any further.

Branching flows. In `a-normalize` and `select-instructions`, at most one target node is produced from any source node. However, other passes may produce two or more target nodes instead. There are several visualization design considerations to keep in mind when supporting this behaviour: Should there simply be multiple flows from the same source node, or should a single flow split into many and how? When a target node is selected, should its flow siblings be selected as well? How would we deal with the visual clutter resulting from flows splitting into many over several passes?

Sophisticated indentation. In our visualization, we choose to place child AST nodes either on the same line as their parent or directly below the parent with indentation, but there are many ways a programmer may choose to indent their code. In Racket-like languages such as those in CPSC 411, by convention, there are four different styles depending on the type of expression. Displaying code in these four different styles is mostly an aesthetic consideration, but would enhance the user experience.

Grammar integration. One of our targeted tasks is to enable students to compare how different kinds of expressions are compiled (**T2.1**). When focusing on a single expression, the student may wish to map its structure back to the BNF grammar to better understand how the substructures of a certain kind of expression is compiled. Additionally, the initial motivation for LastPass states that the many sources of information required frequent context switches, and we wished to reduce this pain, but even with LastPass, they would still have to open up the BNF grammar separately and mentally perform the mapping.

It would be helpful if the visualization not only displays the BNF grammar for the relevant language as an additional linked view, but also highlights instances in the concrete program of a BNF grammar rule that a student hovers over, so as to locate unfamiliar rules in context and trace them back to their origin in the source program.

Custom program comparison. Suppose a student has implemented a compiler pass incorrectly, producing an incorrect target program. They may wish to compare it with the correct target program, which can be obtained from LastPass. However, they will have to perform this comparison on their own, outside of our tool. Being able to visualize the differences between a compiled target program and a custom user-provided program would be helpful in the compiler pass implementation debugging process.

9.4 Lessons Learned

What we learned from the development of LastPass coalesced as we wrote this final report, and reflected on our process. In compiling responses from our end-user

study, we found plenty of valuable feedback that we would have benefitted from receiving in a *Formative* User Study. We learned that we should have front-loaded prototype development to elicit as much feedback from potential users as early as possible so that we could still have time to incorporate said feedback into our final solution, instead of spending too much time perfecting pen-and-paper prototypes.

We also learned that this early prototype development period should have been performed before selecting which libraries to use. Selecting D3.js early in development for its fancy example projects put blinders on our development to alternatives, and overall slowed our development process when we decided to remove the D3.js dependency from the project.

10 Milestones and Schedule

Our proposed development schedule of LastPass is as seen in Table 1. We include the name of the team member that completed the proposed task as well as the actual hours it took. "All" indicates that all team members contributed relatively equally, a team member name followed by "and rest" indicates all members contributed, but that team member took the lead. We made a mistake when providing the proposal, only allocating 140 proposed hours instead of the required 240. We leave the original 140 proposed hours as they were in the proposal, but were pleased to have the 100-hour buffer. Additionally, the "Learn D3.js" Milestone remains an artifact of an early belief that D3.js[13] would be an integral part of the project before it was removed later in development.

11 Conclusion

We have presented LastPass, an interactive visualization tool for multi-pass compilers. This tool is intended to be used in CPSC 411 to aid student's understanding of the compilation process overall, as well as understanding the effects of particular passes. We achieve this by showing students the transformations performed in each compiler pass. In the overview, students are able to see all possible passes with all the pass edges illustrated. Students can then selectively choose passes to show in detail, allowing them to see individual subexpression changes on a pass-by-pass level. We believe LastPass is a suitable tool for understanding how a compiler performs its incremental transformations of an input program.

[13] https://d3js.org/

References

[1] Almeida-Martinez, F. J., and Urquiza-Fuentes, J. Syntax Trees Visualization in Language Processing Courses. In *9th IEEE Intl. Conf. Advanced Learning Technologies* (2009), pp. 597–601.

[2] Andrews, K., Henry, R. R., and Yamamoto, W. K. Design and Implementation of the UW Illustrated Compiler. In *Proc. ACM SIGPLAN 1988 Conf. Programming Language Design and Implementation* (1988), PLDI '88, p. 105–114.

[3] Bolte, F., and Bruckner, S. Vis-a-Vis: Visual Exploration of Visualization Source Code Evolution. *IEEE Transactions on Visualization and Computer Graphics* (2019), 1–1.

[4] Devkota, S., Aschwanden, P., Kunen, A., Legendre, M., and Isaacs, K. E. CcNav: Understanding Compiler Optimizations in Binary Code. *IEEE Transactions on Visualization and Computer Graphics* (2020), 1–1.

[5] Gondow, K., Fukuyasu, N., and Arahori, Y. MieruCompiler: Integrated Visualization Tool with "Horizontal Slicing" for Educational Compilers. In *Proc. 41st ACM Technical Symp. Computer Science Education* (2010), SIGCSE '10, p. 7–11.

[6] Krzywinski, M. Designing for Color Blindness, 2020. http://mkweb.bcgsc.ca/colorblind/palettes.mhtml.

[7] Telea, A., and Auber, D. Code Flows: Visualizing Structural Evolution of Source Code. In *Proc. 10th Joint Eurographics / IEEE - VGTC Conf. Visualization* (2008), EuroVis'08, p. 831–838.

[8] Zhao, W., Cai, B., Whalley, D., Bailey, M. W., van Engelen, R., Yuan, X., Hiser, J. D., Davidson, J. W., Gallivan, K., and Jones, D. L. VISTA: A System for Interactive Code Improvement. In *Proc. joint conf. on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems* (2002), LCTES/SCOPES '02, Association for Computing Machinery, p. 155–164.

Table 1: Project Schedule. "All" means all members contributed equally. "$FIRSTNAME and rest" indicates all members contributed, but the member mentioned took the lead.

Milestone	Description	Proposed Hours	Performed By	Actual Hours
Pitch	Prepare slides, record video	1	Braxton and rest	2
Proposal	Refine ideas, write proposal	10	All	20
Learn D3.js	Read documentation, experiment with D3.js	12	All	16
Scaffold Project	Set up the repository, gather dependencies	4	Braxton	4
Build Prototype	Hard coded minimal prototype	20	Jonathan and rest	24
Racket Compiler	Instrument the reference compiler	16	Paulette	20
Data Transformation	Write transformation from s-expression to JSON	5	Jonathan	6
Updates	Refine abstractions, update writeup	5	All	8
Peer Project Review	Prepare for project exchange	4	All	4
Build MVP	Build LastPass	20	Braxton	45
Polish Viz	Iterate on MVP	10	Braxton and rest	13
User Study	Conduct End-User Study	-	Braxton	3
Final Presentation	Record video, rehearse for Q&A	10	Paulette and rest	22
Final Report	Finalize the paper	20	All	24
	Total Hours	140	All	211