

Visualizing Compiler Passes with FirstPass

Paulette Koronkevich
pletrec@cs.ubc.ca

Braxton Hall
braxtonh@cs.ubc.ca

Jonathan Chan
jcxz@cs.ubc.ca

23 October 2020

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 source and target programs does not reveal much about how the pieces of the programs were transformed.

In this paper, we present FirstPass¹, a multi-pass compiler visualization for UBC's upper-year compilers course (CPSC 411) that aims to elucidate the connections between source and target program substructures and to provide an intuitive understanding of the kind of structural transformations that compilers do. Given a particular source program, it presents intermediate and final programs – the results of multiple stages of compilation – side by side, with interactive flows between related components of source and target programs.

We limit the scope of this project to only two particular fundamental compiler passes: A-normalization and instruction selection. 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 the UBC's CPSC 411: *Introduction to Compiler Construction*, a student and her two project teammates 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 program transformation. She attempts to build a mental model of what the transformation is meant to be by looking at the static definitions one at a time.

¹formerly Untitled Compiler Pass Visualization

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.

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 our new compiler visualization, FirstPass, 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 giving away too much of the *how*. 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. We then outline the primary tasks students are expected to complete throughout the course in [Section 3](#), as well as an abstracted representation of programs and compiler passes which we manipulate. [Section 5](#) discusses relevant work in visualizing code and compilers, and how existing solutions fall short of our tasks. We introduce our proposed visualization in [Section 6](#) and detail its design and implementation. Finally, our milestones and schedule can be found in [Section 7](#).

2 Background

In CPSC 411, a compiler begins with a *high-level program* and ends with an *assembly program*. But it doesn't 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 compiles 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*.

As mentioned, we will focus on two specific compiler passes:

1. **A-normalization**, which sequentializes necessary computational steps and makes control flow explicit; and
2. **Instruction selection**, which sequentializes the program further by converting function calls to jumps.

To compile a program, a compiler pass inspects and manipulates the structure of a program. A program consists of an *expression*, which itself may contain further *subexpressions*. Passes operate recursively, transforming expressions and subexpressions of the source language into expressions of the target language. Programs are internally represented as *abstract syntax trees* (ASTs), where a root node is an expression and its children nodes are subexpressions.

A compiler pass, in general, is a function that takes a source program in one language and returns a target program in another language. However, what we want to visualize is not the general, abstract compiler pass, but rather the effects of a compiler pass on a

specific, *concrete* program. We will then refer to the associations between subexpressions of source and target programs as a *concrete compiler pass*.

3 Task Abstractions

From our own domain expertise and experience in CPSC 411, we have isolated three primary tasks that students must complete during their weekly development cycles where our approach seeks reduce friction. All three tasks are reiterated from [Subsection 1.1](#).

3.1 Pass Triage

Each week, CPSC 411 students grouped in teams of three are tasked with constructing a series of compiler passes. To divide the compiler passes between the members of the group, students must reason about the complexity each pass, and estimate a measure of effort that it may take to implement them. Students require a method of exploring the new passes of the week so as to discover passes of particular complexity, and create an informed distribution of labour within the team.

3.2 Compiler Exploration

While 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, often making 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* and how it works. This model should reduce the number of contexts that a student must use to build their understanding. Lastly, a student must be able to discover semantic information about the compiler pass without compromising the a context that provides both concrete source and concrete target programs.

3.3 Compiler Debugging

As with all complex systems, compiler passes are prone to error. When students have a failing test case, they can refer to FirstPass 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

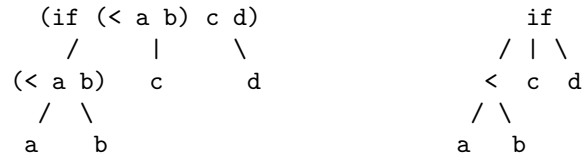
to gain a deeper understanding how certain complex aspects of passes are implemented. Equipped with this better understanding, they are better prepared for creating correct compilers.

4 Data Abstractions

4.1 Program Representation

Recall that the internal representation of a program is an AST. This is merely a specialized tree, which in turn is a directed acyclic graph where nodes have at most one parent. Although we will be dealing with more general graphs, for clarity we continue to refer to the portion of a graph that corresponds to a program as an AST.

Each node of an AST is a subexpression of the program, and directed edges between nodes point from parent nodes to children nodes. Often we will fix the orientation of the tree so that edges implicitly point downwards. For instance, given the program `(if (< a b) c d)`, the AST looks like the following tree on the left:



Some representations will use only the prefix of non-leaf expressions as the nodes, like in the tree on the right above. Although more concise, we will not be using this representation, since we want to highlight each subexpression; a compiler pass is ultimately a recursive transformation of an AST by inspecting each subexpression.

4.2 Compiler Representation

A concrete compiler pass involves the ASTs of a source program and a target program. The pass transforms a node from the source AST to zero or more nodes in the target AST. Therefore, we have additional directed edges from source nodes to target nodes, and we can define a concrete compiler pass as a set of these edges from target to source nodes. For example, if the above program compiles to `(let ([p (< a b)]) (if p c d))`, then the `(< a b)` source node may point to the target nodes `(< a b)` and `p`. Note that these directed edges are distinct from the edges within a single AST. Formally, we assign each directed edge a categorical attribute, where inter-AST edges are marked with the relevant compiler pass name, and intra-AST edges are marked with some null value.

A node may also have several relevant derived attributes. An important one is the *node depth*, a sequential attribute counting how many edges it is away from the root. A compiler visualization can use this information to show how much a given pass “flattens” a program. Another sequential attribute is the number of target nodes pointed to by a source node for a given pass, which can be useful in visualizing how much that pass “expands” a program.

There is one more property of programs to consider: geometry. Students and programmers don’t deal with the abstract tree-form of programs, but rather with source code directly, which often has a fixed shape on the screen. A compiler pass visualization, then, must take into consideration the position of subexpressions relative to each other in a given program.

4.3 Dataset

The total dataset that we work with for some arbitrary source program is then a directed graph consisting of the three ASTs representing the source, intermediate, and target programs, and two sets of directed edges from source to intermediate and intermediate to target ASTs representing the two concrete compiler passes. For the tasks at hand in [Section 3](#), we expect to see small source programs with AST node sets in the cardinality range of dozens, which we infer from the set of test programs used for the reference compiler for CPSC 411. Correspondingly, the cardinality of compiler pass edge sets is also in the range of dozens.

5 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. We find that our solution more closely matches transformations over source code directly, rather than visualizing compiler implementations or optimizations.

5.1 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. Finally, as mentioned previously, our proposed solution deals with source program text rather than the AST directly.

5.2 Visualizations of Compiler Optimizations

VISTA (6) is a system for interactive code improvement, specifically for the domain of embedded systems applications. VISTA works over block-based language programs, specifically 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. VISTA also has capabilities for adding and visualizing user defined transformations.

VISTA displays source program text directly, similar to our solution. However, 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.

5.3 Visualizations of Transformations over Source Code

MieruCompiler (4) is an educational compiler with similar goals to ICOMP. However, MieruCompiler also visualizes the static source code and assembly code, 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 static 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 (5) 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.

These two 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.

6 Visualization

Our proposed solution is to create an interactive visualization, where students may input an arbitrary source program to see an overview of the compilation process. By analyzing this overview, the students may pinpoint a particular pass that is of interest, then switch to a detailed view of the pass. Alternatively, if the student is implementing a particular pass, they may query multiple source programs and analyze how the programs change in the detailed view of the pass. We elaborate on these views in the following subsection, and conclude with some high-level implementation ideas.

6.1 Design

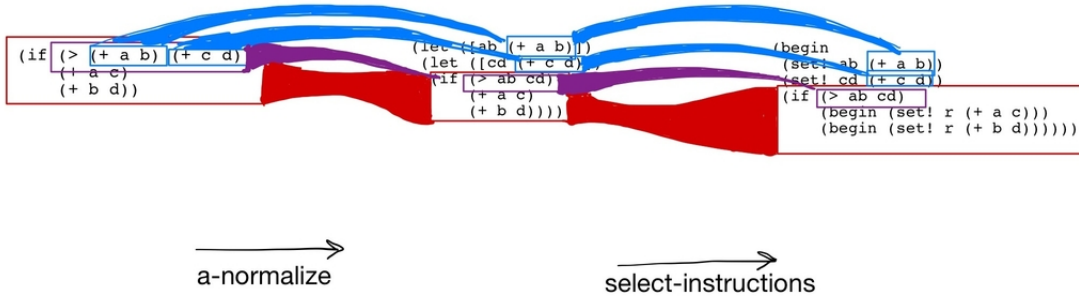


Figure 1: A mockup of an overview visualization of the two passes.

When a user inputs a source program, an overview of the two aforementioned passes will be computed. This overview shows how expressions from the source flow into the target program for each pass. Showing this overview for every possible source expression would be overwhelming; we propose selecting particular expressions to be shown in the overview. A preliminary mockup of this is shown in Figure 1. Note that the effect of the `a-normalize` pass is emphasized by showing how the nested subexpressions in the `if` condition flow to be assigned before the `if` expression.

As shown in the mockup, there is potential for overlap between these flows. We will be searching for literature for solutions; one possible solution is a flow map layout via spiral trees (3), another is by shading the flows as tubes as done in Code Flows (5).



Figure 2: A mockup of a detailed visualization of a selected pass.

When a particular pass is of interest to a user, they may select it by clicking on the name of the pass. The source program and the target program (the output of the pass) will be shown in textual form, side by side. The user can then hover over particular expressions in the source program to highlight the corresponding expression in the target program. The user may click to select a particular expression, which will show the flow of the expression to the target program. This can be seen in Figure 2. Selecting multiple of these code flows could result in similar overlapping issues as the overview, as shown in Figure 3.

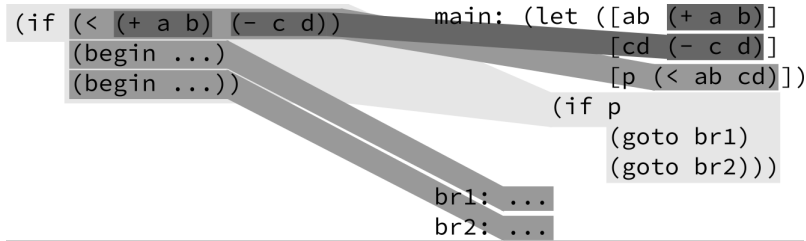


Figure 3: A mockup of selecting multiple code flows at different nesting levels.

6.2 Implementation

To implement our proposed solution we have divided our system into three packages.

1. **The Racket Compiler:** We seek to instrument the reference compiler for CPSC 411 written in Racket.² This compiler does not connect source expressions with target expressions in each pass, so this is a necessary instrumentation step. We will be writing the instrumented compiler in Racket so as to reuse as much of the reference compiler as possible and not rewrite it.
2. **The Server:** We will run a Node³ and Express⁴ server written in TypeScript⁵ which will serve The Frontend, reroute requests to The Racket Compiler, and process the dataset into a usable JSON format. Node, Express and TypeScript were selected as they are familiar and require minimal overhead to set up.
3. **The Frontend:** The client side piece of our system will be written in TypeScript using React⁶ and axios⁷ to handle requests. We will leverage the D3.js⁸ framework to display the interactive visualization to users. We anticipate this piece of our system to house the majority of its complexity and time in implementation. We have chosen to use D3.js despite unfamiliarity with the framework for its popularity and apparent flexibility.

All services will be hosted on the Software Practice Lab’s website, <https://se.cs.ubc.ca/>.

Currently, we foresee the largest challenge to our implementation as being our collective unfamiliarity with D3.js (or any other visualization framework). Spending more time learning how to use this framework may cause small changes to our current implementation plans if we find it will not service our needs.

6.3 Results

Following the creation of FirstPass, we seek to facilitate the following scenarios, each analogous to one task in [Section 3](#).

²<https://racket-lang.org/>

³<https://nodejs.org/en/>

⁴<https://expressjs.com/>

⁵<https://www.typescriptlang.org/>

⁶<https://reactjs.org/>

⁷<https://github.com/axios/axios>

⁸<https://d3js.org/>

Table 1: Project Schedule

Milestone	Description	Hours	Deadline
Pitch	Prepare slides, record video	1	Oct. 1
Proposal	Refine ideas, write proposal	10	Oct. 23
Learn D3.js	Read documentation, experiment with D3.js	12	Nov. 1
Scaffold Project	Set up the repository, gather dependencies	4	Nov. 1
Build Prototype	Hard coded minimal prototype	20	Nov. 10
Racket Compiler	Instrument the reference compiler	16	Nov. 10
Updates	Refine abstractions, update writeup	10	Nov. 17
Peer Project Review	Prepare slides, prepare demonstration	4	Nov. 19
Build MVP	Build FirstPass	20	Dec. 1
Polish Viz	Iterate on MVP	10	Dec. 6
Final Presentation	Record video, rehearse for Q&A	10	Dec. 10
Final Report	Finalize the paper	20	Dec. 14

6.3.1 Pass Triage

A CPSC 411 student and her team have been given 21 compiler passes to implement in seven days. Looking to evenly distribute the work of the compiler passes between herself and her two teammate, she uses FirstPass. She visit the overview page and inputs a sample source program, and mouses over expressions looking to discover passes that incur large changes to its program, whether that be by reordering or locating nodes in the graph with high degree. Armed with an intuition for which passes will require the greatest effort, she assigns passes to each team member.

6.3.2 Compiler Exploration

Confused reading the documentation for a compiler pass named `select-instructions`, the student returns to FirstPass and enters the detailed view. Inputting a sample source program and viewing the `select-instructions` pass, she explores the mechanics of the program transformation. She clicks a subexpression to keep its flow highlighted and continues mousing over other subexpressions to highlight their flows and compare how they differ. She begins to coalesce an understanding of what `select-instructions` *does*.

6.3.3 Compiler Debugging

While finishing up work on the `select-instructions` pass of her compiler, the student gets stuck on a particular failing test cases. She sees one expression in the target output and has no idea where it originated from. She again returns to FirstPass, visits the detail view and inputs the source program from her failing test case. She then mouses over the seemingly extraneous expression in the target program and traces it back to the an expression in the source program, and begins to make sense of her mistake.

7 Milestones and Schedule

The proposed development schedule of FirstPass is as seen in [Table 1](#). We intend to spend upward of 140 hours, and have distributed 137 of them generously in the schedule. We aim to have our minimum viable product deadline as early as possible to create a buffer for unforeseen complications.

While development of the frontend and the core visualization are both to be a joint effort, Paulette will be taking the lead on the Racket compiler, Braxton will be leading service orchestration and server development, and Jonathan will be conducting small user studies concurrent with our MVP development.

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] BUCHIN, K., SPECKMANN, B., AND VERBEEK, K. Flow map layout via spiral trees. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Dec. 2011), 2536–2544.
- [4] 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.
- [5] TELEA, A., AND AUBER, D. Code Flows: Visualizing Structural Evolution of Source Code. In *Proc. 10th Joint Eurographics / IEEE - VGTC Conf. Visualization* (Chichester, GBR, 2008), EuroVis'08, The Eurographs Association & John Wiley & Sons, Ltd., p. 831–838.
- [6] 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/SCOPE '02, Association for Computing Machinery, p. 155–164.