Assemblance:
A Visual Tool for Learning Assembly Language

Rob Whitaker
Adviser: Bob Dondero

Abstract

We describe a new code exploration tool aimed towards those learning an assembly language for the first time. The web-based application allows a user to upload a valid C file and compiles it on the server. By processing the debugging information generated by `gcc -g` *and generating appropriate html markup, the server returns an interactive side-by-side display of the source and compiled assembly. The display indicates the correspondance between lines of C and assembly, and provides mouseover tooltips explaining the effect of a certain subset of* `x86-64` *mnemonics and identifying stack offsets with their variable name and type in the source.*

1. Introduction

Assembly Languages[1] provide the lowest-level bridge between machine instructions and human language. Each line of assembly describes a single machine instruction to be performed by the CPU, but lacks the highly memory-efficient encoding of actual machine or object code.[] As such, writing in Assembly language allows the programmer to have complete control over the state of the machine, and specify the exact sequence of instructions to execute and memory addresses to manipulate. Programs written in assembly are often extremely fast, and are writable in even the most impoverished development environments.[**?**] Before the widespread adoption of higher-level languages like C, even programs as large as text-editors like the GNU *ed* were written entirely in assembly.[**?**]

[1] There are many different flavors of assembly language. In this paper we will restrict our focus to x86-64, the 64-bit version of AT&T's “gas" assembly language. Though much of the arguments here apply to any assembly language, all examples will be in x86-64, and the tool described uses this flavor exclusively.

Over time, with the development of efficient optimizing compilers, high level languages were able to be translated to their assembly-language equivalents without a significant loss of performance.[**?**] Because of the convenience of abstractions like variables and strictly enforced types, high level languages are much easier to learn and write, and programs written in them are much easier to debug. Still it is the case that an understanding of the way in which the machine performs abstract computations – such as assignments, control flow, arithmetic, and function calls – helps the high-level programmer write more efficient code. Understanding what's going on "under the hood" gives a clearer picture of the system on which you are developing and allows you to interact with its services confidently.

Regardless of whether one appreciates the usefulness of writing in assembly languages, many Computer Science courses require the use of assembly language at some point. Both of Princeton's COS217: *Introduction to Systems Programming*[**?**] and COS318: *Operating Systems*[**?**] feature it signficantly, and it is often cited as one of the most difficult aspects of both courses, leading us to investigate ways to augment its teaching.[**?**]

A general algorithm for teaching a concept is to identify students' current knowledge and the subject to be taught, and build a bridge that relates the two. In the case of learning assembly, this means relating the machine-level programming to a high-level language with which the student is already familiar. For COS217, at the time of learning assembly the student can confidently compose programs in C, and is familiar at least with the concept of compilation to a set of instructions the machine can understand. As the students go on to perform optimization by hand, using assembly, the bridge the course aims to build lies between C and the assembly language.

In the course as it currently stands, a number of C programs are presented, along with hand-coded and well-formatted translations into assembly. The students are not shown the assembly version generated by `gcc`, and are even urged not to spend too much time looking at compiled code, as the logic used by the compiler is so sufficiently different from what they know that it would not improve their understanding. In defense of this pedagogical decision, without any optimization, the compiler often writes assembly code that is confusingly redundant or unnecessary, and manages the

stack using both a stack pointer and a “base pointer" pointing to the top of the stack before entering the current function. Looking at compilation under successive levels of optimization only makes things worse, as entire variables and functions can disappear from the code. Trying to understand the rationale behind these sorts of opaque compiler decisions at the same time as learning the instructions available, the different addressing modes, function calling, and programming without variable names could in fact be overwhelming to the student.

Still, this suggests that there is a wealth of information not being utilized. If the most powerful tool that exists for relating C to assembly can not help better understand the relationship, we need to improve the tool.

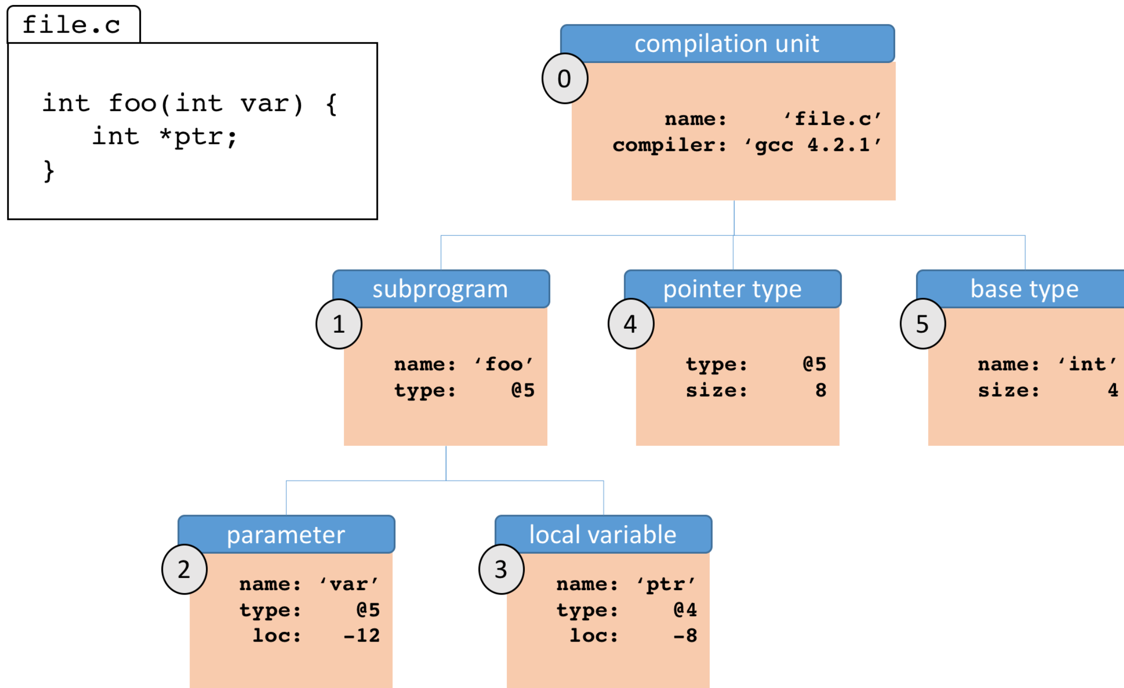
2. Problem Background and Related Work

To inform our development of a tool that visually relates source code to its assembly equivalent, we look to existing tools and information formats that already relate the two in some way. We are particularly interested in how to present that relationship in a clear an concise way, so that those learning the latter can internalize the act of translating between them. We review a format specification and two pieces of software that perform similar services.

2.1. The DWARF Debugging Format

When a file in a high level language (such as C) is compiled to machine code, it undergoes a series of transformations that effectively destroy the program’s high-level structure that makes it human-readable. While the compiler certainly takes advantage of things like type designations and variable names to keep track of the program as it generates machine code, the latter has no representation of either. However, in the process of compilation, the compiler must at one point keep track of both. To see this, imagine a variable is created in C. The compiler will assign it some location in memory (in a register or on the stack), and proceed. It cannot simply forget this information and move on however, as it must be able to know which location must be accessed when that variable is referenced again later in the source. The debugging information is a snapshot of the compiler’s knowledge of the relationship between the source and target.

Figure 1: Cartoon of DWARF debugging information entry tree for a simple C program. Note that much of the debugging information has been decoded to a friendlier human readable form, and some information has been left out.



GCC generates object code in something called Executable and Linking Format, (ELF). The debugging standard to accompany ELF since the 1990s is known as DWARF. DWARF encodes the source-target relationship by maintaining a hierarchical tree of debugging information entires (DIEs) each with several attributes and children, if applicable (see Figure 1 for a simple example). The top level of DWARF DIE tree contains compilation units, and each of its children is some object or instance that can be found in that compilation. As seen in Figure 1, DIEs can be specific, like the function `foo()`, or represent more general language constructs, like an integer or pointer type. Global variables are also siblings of functions, while local variables and formal parameters are children of the functions to which they belong. Each DIE has a “tag," identifying what type of entry it is. Figure 1 uses plain english versions of the tag names for readability, but each is given an unambiguous descriptor in the DWARF specification (e.g. `DW_TAG_formal_parameter` for function arguments).

The attributes of a DIE vary by the type of entry, and the attributes available for different DIEs of the same tag are not necessarily the same. Though inconsistent, the information is valuable and can completely specify the programatic representation of a particular variable. Even when it's been translated out of its space-efficient encoding in the ELF file and into a human-readable form, it takes quite a bit of expertise to make sense of the representation.[2]

2.2. GDB

The GNU Debugger (GDB), is perhaps the most popular client of the DWARF specification. The goal of the debugger is generally quite similar to that of assembly annotation in that it must in some way represent the relationship between the source and compiled version of a program. The programmer would like the convenience of debugging within the abstraction of the high-level language in which the program is written. At the same time, the program can only be faithfully executed on the target machine directly in its compiled and assembled form. These two different levels of abstraction are referred to as the Symbol Side and Target Side of the debugging problem, respectively.[**?**] By parsing the DWARF information described above and keeping track of the current state of the machine, gdb is able to bridge the two sides and respond to a vast array of user commands: displaying arbitrary local variables, dereferencing pointers, and setting breakpoints.[2] Not only can it display the contents of the memory address that corresponds to a given variable name in the source language, but given an address it can indicate the name of the object that resides there. To do this, gdb must be able to map from memory addresses to variable names and vice versa, and to set breakpoints gdb must map lines of code in the source language to the corresponding set of machine instructions.

Some of this information is only available at runtime: the exact addresses and sizes of dynamically allocated chunks of memory can depend on user input. However, the vast majority of Symbol Side information is available at compile time, and resides in the GDB's static symbol tables generated at

[2] In fact, a significant portion of the work described in this paper was just wrangling already-parsed DWARF sections into a usable format.

application startup. [3]

2.3. Compiler Explorer

3. Approach

While all the tools above represent the relationship between C and assembly, they do not do so in a way that is meant to teach the beginner. The DWARF format is only readable by machines, and even its decoded form is unenlightening to a beginner. GDB can help even beginners find bugs in their programs, but it does not aim to teach any concepts itself. Finally the compiler explorer does bill itself as an educational tool, but is focused on studying compilation from an experienced programmer's perspective: comparing subtle variations accross compilers rather than explaining their process and the code they generate. In all, this motivated us to develop a tool inspired by those mentioned above, but with an explicitly educational focus.

3.1. Design Goals

While the software was designed with COS217 students as the primary audience, the tool is also intended to be useful to a student studying assembly language in any context. To better guide the project, we first enumerated a number of design goals:

1. *Handle arbitrary C programs*

   Currently COS217 provides example assembly implementations of several programs originally written in C with which the student is already familiar. These prove useful, but given that each one must be written by hand by an instructor, their impact is finite. We would like the student to be able to learn from any program with which she is already familiar, and so the output of the software must handle any valid C file. Valid C programs for which the debugging information is too complex to process should not crash the program, but should instead be displayed with as much information as possible.

[3] In reality, GDB generates partial symbol tables for large applications to decrease startup time and only completes them when the user stops in a particular function, but though it may not always use it, the information is available outside of runtime.

2. *Easy to use*

   As the target audience is systems programming novices, all information must be easily and quickly accessible. This means the interface should be intuitive, not too cluttered, and without lag. In particular, the implementation of all data-display features must be fast. Additionally, we would like this tool to be available out-of-the box, and require minimal effort on behalf of the user to get it running.

3. *Target-side focus*

   In the language of gdb's source/target division, this tool should principally be concerned with the target side. That is we will pursue features that help explain concepts specific to assembly language, or C concepts only insofar as they relate to assembly constructs. For example, we are interested in identifying a particular location in memory as representing pointer data, but we are not interested in explaining the function of pointers in a C program more generally.

4. *Extendable*

   As the amount of possible assembly and C constructs worth explaining is quite large, we do not expect to implement every possible convenience by the first release. As such we require that the progam's architecture allow additional features to be integrated easily, clearly separating distinct parts of the process.

5. *Usable for COS217*

   While the information we wish to provide should be general to a student learning Assembly language anywhere, our first priority is Princeton Students in the Introductory Systems Programming class. That is, the particular flavors of C and Assembly should be the same as those used in COS217, and content covered in the class should be prioritized over additional information.

3.2. The Development Process

To first prove the concept of the tool, the first iteration was simply a python script to insert lines of source code as comments before their corresponding blocks of assembly. This required the user to first compile their program with `gcc -g` and specify the locations of both the original C file and

the assembly version. Though successful, a text tool is extremely limited in how much data it can contain before becoming overwhelming.

Considering goal number two led us to settle on a browser-based application, as HTML provides a much richer set of display options than plain text. Further, hosting the application on a server would allow it to be accessed easily from anywhere, also in line with goal two. While a purely browser-based app (served as static HTML/CSS/Javascript) was considered, it would have suffered from a number of weaknesses that we ultimately deemed unacceptable. First, the two most crucial functions of the application – parsing C and Assembly syntax, and parsing DWARF debugging information – lacked significant library support in javascript, while python has stable (and open-source!) libraries for both. Further, without a way to compile C code in a browser, the user would have been left to supply both files. Not only is this more tedious (and contrary to goal two) but it introduces more opportunities for error, as the user may choose to use a different compiler than the application is expecting. Further, since COS217 requires students to develop on a linux cluster, and the majority of students do not use linux on their personal machine, we can faithfully live up to goal five only if the application handles compilation.

Performing compilation on the server also allows for much more control over the debugging information and exact parameters of compilation. Because the only available python package for parsing the DWARF format operates on the ELF (.o) file rather than the assembly file,

In terms of UI design, we used the same side-by-side display used by the Godbolt Compiler Explorer.

3.3. Functionality

The final product

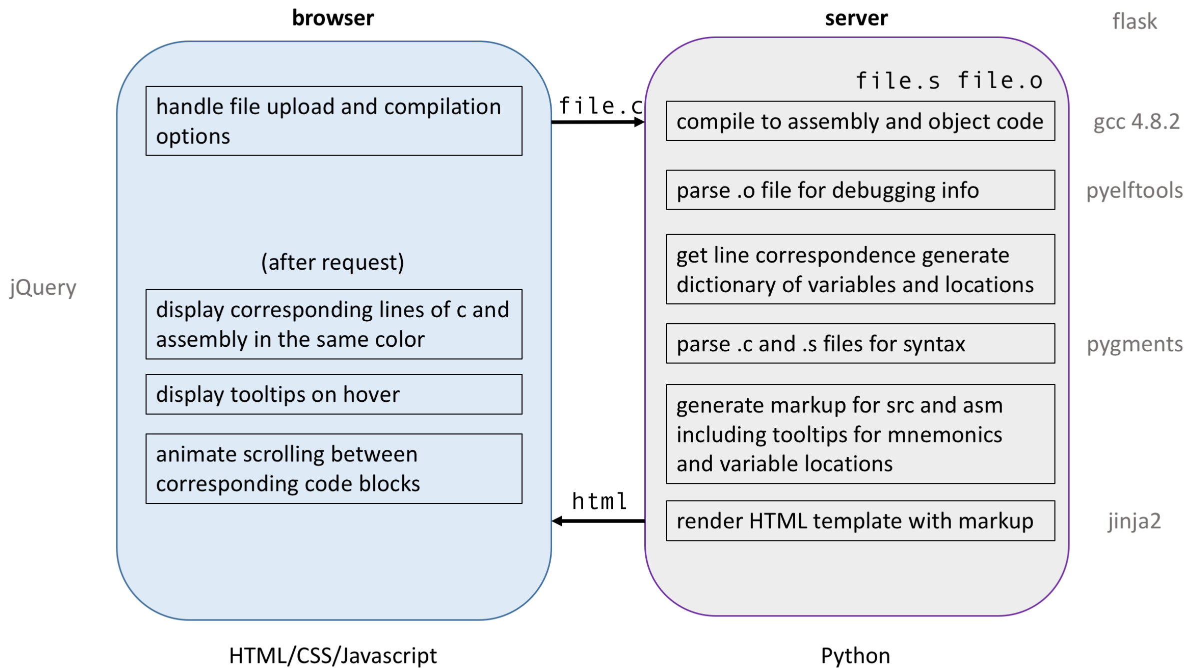


Figure 2: Schematic of program organization and data-flow.

4. Implementation

4.1. Overview

In this order, we will proceed to describe the details of each part of the program, following the path of data through application.

4.2. Compiling and Optimization

The application's server side has a single endpoint (the root), and all requests are handled by the function at that endpoint. The server also has an HTML template that is used to render all content on the page, with the server processing each request and returning a rendering of the template based on the data it receives.

4.3. DWARF Parsing

The DWARF information is parsed in its own module providing a single API function which accepts an open stream pointing to a ELF file, and returns a dictionary for each function mapping string

locations (as they would appear as an assembly operand) to parameter and local variable names, along with their type and the source line in which they were declared.

Our implementation performs this service in three steps, illustrated in Figure 3. The first step is the low-level parsing of DWARF information, which is done by the python package pyelftools[**?**]. Then in a first pass over the DIE tree, we generate a simplified representation of each variable used in the program (and its attributes), and each data type. A second pass over this simplified representation resolves complex types and generate the desired relationship: associating memory locations with names and types.

The primary function of the first pass is to align the data in a reliable format, as the DWARF specification does not require entries to have particular attributes. Data for the same attribute is sometimes stored in different formats, and so we also have a set of decoders to convert all attributes into a form we can use.

4.3.1. Resolving types

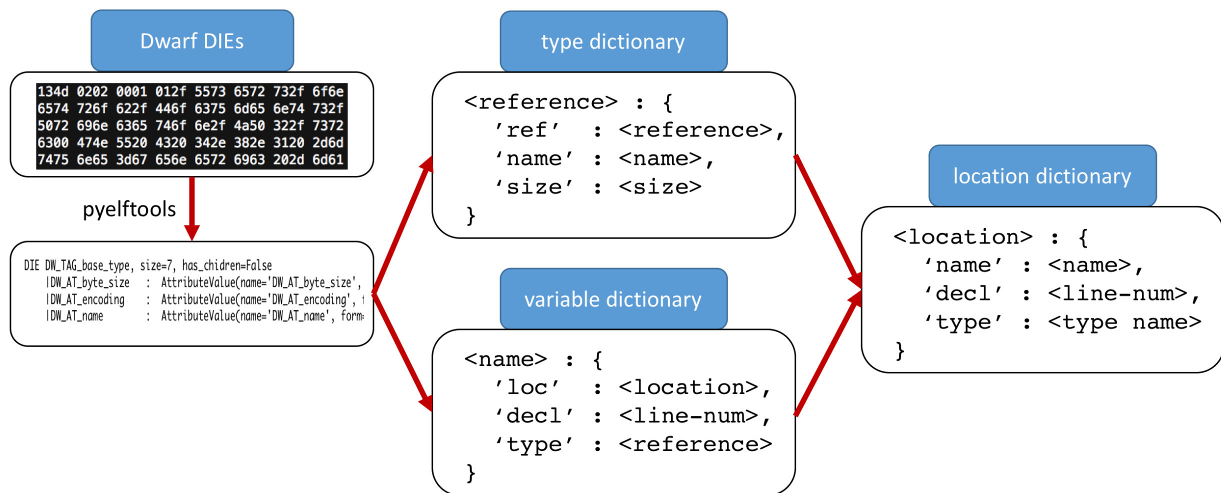


Figure 3: Flow of DWARF data through Assemblance backend. The pyelftools package decodes the DWARF format from its terse memory-efficient version into python objects which contain the same information. These objects are iterated over to generate a dictionary of all types used by the program and a dictionary of variables used in each function. The information in each of these is then combined to make a dictionary of memory addresses used by each function with references resolved into a string describing them.

4.3.2. Resolving locations

4.4. Markup

To facilitate speed on the browser side, all of the markup for our side-by-side assembly display is generated by the server in a single pass. Both the C and assembly markup are generated one line at a time, wrapping each aspect of the text in html elements with classes that describe their function, as determined by the lexers described below. We erred on the side of overspecifying the code structure in the markup, so that specific elements or classes that are not used in the software's current version could easily be utilized in future versions.

4.4.1. Syntax parsing Syntactic analysis is done on both the assembly and C side as the markup is being generated. A python package named pygments[**?**] was used to highlight the source, using its built in `CLexer` and `HtmlFormatter` classes. The lexer contains a specification of the language as a set of states, with a list of triplets in each state indicating a regular expression to match, a token to yield, and a new state to take.

4.4.2. Tooltip generation In addition to the HTML formatting provided by pygments, we hook our own formatting steps immediately after the lexing. Specifically, this is the injection of an additional `<div>` element as a child of the `<div>` holding the particular assembly language token. In its current version, the software supports two types of tooltips: mnemonic information, and variable information. Each type of tooltip has an associated format string that defines the layout (with html) of the tooltip, and is then formatted with the particular information for each token.

As each token is added to the markup, the text of mnemonic tokens is looked up in a dictionary of mnemonic information, stored in a static json file on the server. This returns an entry with information such as the mnemonic's english name, it's syntax, the size and type of its operands, its effects on the EFLAGS registers, and a description of its function. Not all of this information is used in the generated tooltip, though this could easily be changed by replacing the tooltip's format string.

4.4.3. Line matching

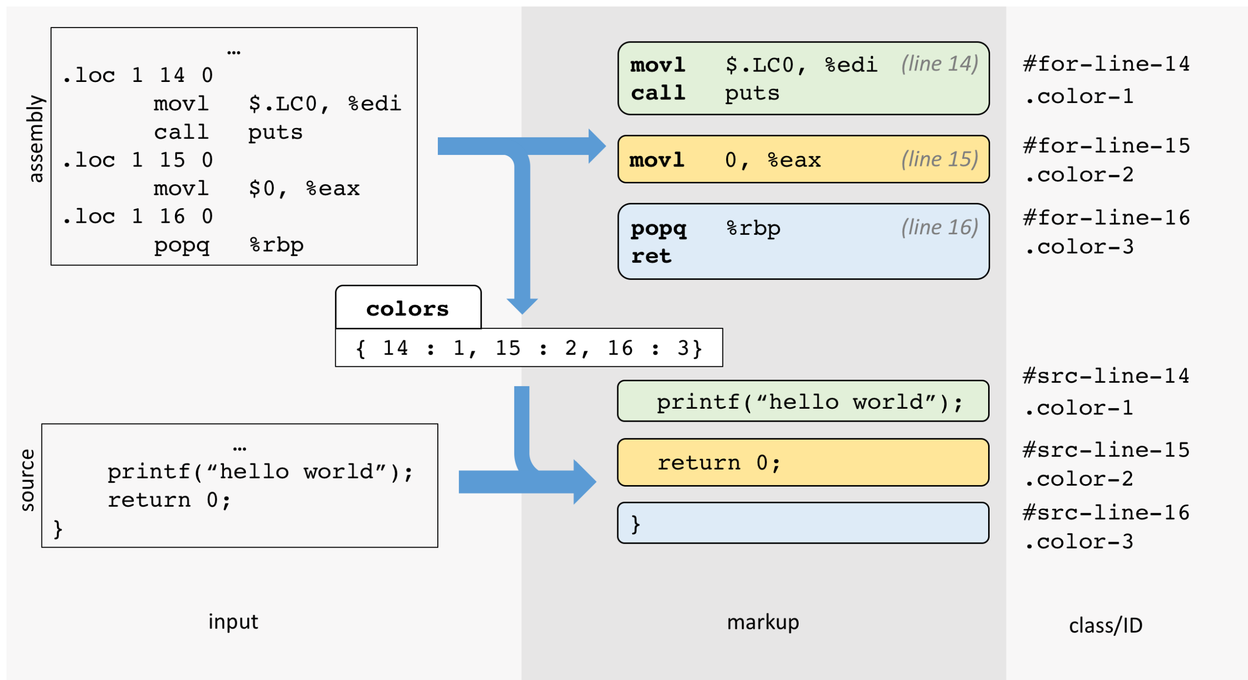


Figure 4: A cartoon depicting an example of the line-matching process.

4.5. UI Interaction

All interaction with the UI is specified with javascript or built-in behaviors of CSS3.

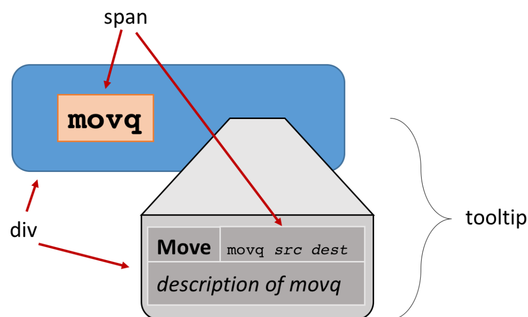


Figure 5: Cartoon depicting tooltip markup. Each assembly language token is wrapped in a `<div>` of class `asm-token`. This contains a `<span>` element with the token text, and if applicable another `<div>` holding the tooltip for that token. Visibility and position of the tooltip is controlled via javascript and described below

4.5.1. Hover-display tooltips. Detailed information about assembly tokens generated on the server and inserted into the markup is only displayed when the user hovers over a particular token. The CSS `:hover` pseudoclass allows selection of elements only when a user mouses over them. Children

of pseudoclassed elements can be selected as well, and our stylesheet sets the visibility property of tooltip elements only when they are children of hovered `asm-operand` class elements.

Unfortunately, in the case of mnemonics which lie on the edge of their parent container, the tooltips are rendered outside of it. CSS provides an overflow property[1] that specifies how to handle child content that extends beyond a container, however, allowing scroll in any direction prohibits overflow from being set to visible in any other direction, and instead content that would render outside of a scrolling container is clipped. To solve this, we remove the relative positioning of the parent, and add a relatively positioned `div` element as the parent of our scrolling container. This element uses `overflow: visible`, and achieves the desired effect (a similar approach is described in [3]). Though now the ancestor controlling the position of the tooltip has no information about the parent mnemonic's position on the page – adding a mouseover event listener to each element with a tooltip to set the parent container's position solves the issue.

4.5.2. Scroll-to-match.

4.5.3. Form Submission. In keeping an intuitive user interface, it makes more sense to keep the upload button on the same side as the source display, and optimization level selection on the same side as the assembly. It also would be destructive to the layout of the user interface to allow elements to spread accross both halves. Thus the file upload and optimization data belong to different form elements, but only a single form submission (POST request) can be sent at a time.

To get around this problem, we replicate an optimization level input element in the file upload form, but set it's visibility to hidden. Then by adding an event listener to a change in state of the interactive optimization level selector and mirroring the value in the hidden input element, we allow any change in either form to submit the one that contains the file upload, which will always contain both values.

5. Evaluation

5.1. Unit Testing

Sample programs

DWARF testing script

5.2. Performance

time dwarf tree parsing and compare to the time it takes to send to server

Currently sends entire markup each time, not as efficient as it could be... use react or angular?

6. Conclusions and Future Work

7. Acknowledgements

I'd like to thank my advisor, Bob Dondero for all his guidance and enthusiasm throughout this project. I'd also like to thank Eric Mitchell for helpful discussions on UI design, and all my friends who've listened to me ramble on about variables and stack offsets for the last few months.

This paper represents my own work in accordance with University regulations

Rob Whitaker

References

[1] Css overflow property. W3Schools. [Online]. Available: https://www.w3schools.com/cssref/pr_pos_overflow.asp
[2] A. Gilpin. (2004) Debugging under unix: gdb tutorial. [Online]. Available: https://www.cs.cmu.edu/~gilpin/tutorial
[3] A. Shirinian. Popping out of hidden overflow. [Online]. Available: https://css-tricks.com/popping-hidden-overflow/

tag	description	keyword	limitations
DW_TAG_base_type	built-in types	int, double, ...	
DW_TAG_pointer_type	pointer to variable	*	
DW_TAG_array_type	arrays	[]	only front of array
DW_TAG_const_type	fixed-value variables	const	not supported
DW_TAG_enumeration_type	integer enumerations	enum <name>	
DW_TAG_typedef	user-defined types	<name> (<base type>)	
DW_TAG_structure_type	C structures	struct	

Table 1: Type support for symbol and stack lookups.

Table 2: Description of syntax used for custom Assembly Operand lexer

A. Supplemental Figures

B. Project Code

Table 3: CSS Classes for specifying syntax