# CS 345H Final Project Report

Ethan Arnold
Anurag Bakshi
Quang Duong
Naren Manoj
Ryan Rice

November 2016

## Contents



# 1 Introduction

The L programming language is a toy functional programming language very similar to untyped lambda calculus. It has features that extend lambda calculus (such as named functions, natural recursion, and `let` bindings, to name a few) which make it more expressive. Over the past semester, we implemented a lexer, parser, and interpreter for the L language.

However, interpretation has its limitations. For instance, it is well-known that interpreted code running atop a runtime is generally slower than machine code preserving the semantics of the original program. In particular, if code is compiled to

machine language, the program directly benefits from hardware optimizations such as branch prediction, which can greatly reduce the runtime of the program. A program running atop a runtime usually cannot exploit these optimizations. On the other hand, some amount of type checking can be done at compile time, which can sometimes eliminate some pesky type errors that wouldn't be caught before program runtime in the interpreter.

The goal of this project is to therefore develop a compiler for the L programming language. We support all features of the L language while enjoying substantial speedup over the interpreter. Specifically, across our benchmarks, we compute an average mean weighted speedup[1] of 76.58%. We also utilize the LLVM compiler infrastructure, which theoretically allows us to exploit the many optimizations that are applied when converting LLVM Intermediate Representation (IR) to machine code.

However, given more time, there are some further optimizations that we could make. For instance, type inference on the original source code would allow us to perform some type checking at compile time. Currently, we cannot perform any type checking at compile time because we box all the primitive types (in L, these are integers, lists, strings, `Nil`, and lambdas) into a single wrapper datatype. The corresponding operators are then applied on this general type. This process defers the type checking to the runtime (more details can be found in the Implementation section). This leaves some additional headroom for further optimizations on the type checking. If some of the type checking is done at compile time, then some of the autoboxing can be circumvented, thereby improving execution speed of the program. Doing this would require knowing the types of variables at compile time; hence, it becomes necessary to implement type inference for this to work.

The benefits of removing the boxing could also be seen in the final outputted machine code. Currently, since all operations are deferred to external functions implementing the operations, LLVM cannot necessarily make optimizations to remove these calls. Therefore, there is a substantial headroom for further speedup. The speedup would be realized when LLVM can optimize out certain operations that do not affect the final semantics of the program.

Furthermore, we noticed that the compilation time for large programs is quite high. While we are not certain about the reason for this, we have some conjectures about the source of the overhead. It is well-known that optimizations account for the majority of compiler runtime. For a large program written in LLVM IR, the compiler has to pass through the entire text of the IR to attempt to make optimizations. While no optimizations are actually made, the compiler still has to iterate through the entire program, which could cause great slowdown. On the other hand,

[1]More details about this computation can be found in the Results section.

the compilation time could have been high as observed due to the large source file; however, it is reasonable to expect that a program will be run far more often than it is compiled. Therefore, it is acceptable to sacrifice time in compilation and make time gains through the actual program execution.

This report is organized as follows. In Section 2, we consider the implementation details of the compiler. In Section 3, we describe our testing methodology. In Section 4, we discuss the performance benefits of the compiler over the interpreter. And, in Section 5, we explore some of the challenges we faced throughout the design and implementation of this compiler.

### 1.1 Installation

The following instructions mirror those inside the README. We verified them on a system running Ubuntu Desktop Edition 16.04 LTS.

To build the compiler:

1. `sudo apt-get install llvm`
2. `sudo apt-get install flex`
3. `sudo apt-get install bison`
4. `sudo apt-get install lib32z1-dev`
5. `make`

To compile and run a ELF executable for an L source file of the name file.L:

1. `make file.elf`
2. `./file.elf`

## 2 Implementation

Before discussing the implementation details of the compiler, it is necessary to outline a high-level overview of the compilation process:

1. The programmer writes the L source code.
2. The L source is lexed and parsed as before.

3. Each AST node is passed to the compiler, implemented in `Compiler.cpp`, for processing. All the primitives are boxed into one union type, which allows us to defer type checking to the runtime. This results in the generation of LLVM IR.

4. The LLVM IR references external functions for each operator, and these functions use the operands of the original operator as parameters. The external functions for the operator are implemented in C (`operations.c`).

5. LLVM compiles the IR, applying optimizations as necessary to generate the final executable.

We will now consider the specifics of the compiler infrastructure. Implementing the operators is a file titled `operations.c`. We dub this as "libl" (inspired by libc). We also have a replacement for the interpreter's `Evaluator.cpp` called `Compiler.cpp`, which converts the AST generated by the L parser into LLVM code.

The goal of `operations.c` is to provide external functions implementing all the unary and binary operators in L. These external functions are referenced by the machine code; hence, every operation visible in the LLVM IR consists of calling the appropriate function with the appropriate arguments. For instance, the expression `1 + 2` compiles to

```
...declarations...

define i32 @main() {
entrypoint:
  call void @symboltable_new()
  %0 = call %struct.Object* @make_int(i64 1)
  %1 = call %struct.Object* @make_int(i64 2)
  %2 = call %struct.Object* @plus_any(%struct.Object* %0, %struct.Object
    * %1)
  call void @display_any(%struct.Object* %2)
  ret i32 0
}

declare %struct.Object* @make_int(i64)

declare %struct.Object* @plus_any(%struct.Object*, %struct.Object*)

declare void @display_any(%struct.Object*)

declare void @symboltable_new()

...additional declarations...
```

under our design. Notice the external declarations of the functions `make_int` and `plus_any`, which are implemented in `operations.c`. These functions are tasked

with determining the type of the `Object` generated, verifying that there are no type errors, and handing off the final evaluation to the specific function. For instance, the `plus_any` function is:

```
Object *plus_any(Object *a, Object *b) {
    if(is_list(a) || is_list(b)) {
        error("Binop @ is the only legal binop for lists");
    }
    if(is_nil(a) || is_nil(b)) {
        error("Nil can only be used with binop @");
    }
    if(a->type == INT && b->type == INT) {
        return plus_int(a, b);
    } else if(a->type == STRING && b->type == STRING) {
        return plus_str(a, b);
    } else {
        error("Binop can only be applied to expressions of same type");
    }

    return NULL;
}
```

and the `plus_int` function is:

```
Object *plus_int(Object *a, Object *b) {
    return make_int(a->int_val + b->int_val);
}
```

Similar code is generated for any of the four basic arithmetic operations. The code generated for a unary operator is similar in idea, but varies between operators. For instance, `print 1` generates

```

define i32 @main() {
entrypoint:
  call void @symboltable_new()
  %0 = call %struct.Object* @make_int(i64 1)
  %1 = call %struct.Object* @print_any(%struct.Object* %0)
  call void @display_any(%struct.Object* %1)
  ret i32 0
}

(external declarations omitted)
```

, and `!1` generates

```
define i32 @main() {
entrypoint:
  call void @symboltable_new()
  %0 = call %struct.Object* @make_int(i64 1)
  %1 = call %struct.Object* @hd_any(%struct.Object* %0)
  call void @display_any(%struct.Object* %1)
  ret i32 0
```

```
}

(external declarations omitted)
```

. We will discuss the unary operators `readInt` and `readString` once we discuss our symbol table implementation.

In the interpreter, the symbol table is implemented with a vector of tree maps. Every push would push the current symbol table onto the vector and add the identifier-value pair into the tree map.

To make this faster in most cases, the compiler scans throughout the source file and collects all the identifiers that are present. Instead of a tree map or a hash map, the compiler generates an array of stacks. Each stack corresponds to each name, whenever a push occurs, it pushes onto the stack of the corresponding identifier. Each identifier is mapped to a unique number starting from 1. Popping likewise just removes the top element of the the identifier's stack. Since the identifier is known at compiler at time, all of these push and pop operations occur in O(1) constant time. Furthermore, this scan occurs at the same time as the IR byte code generation, maintaining this at a single pass.

With this in mind, consider the code generated for the program `let x = readInt in x`:

```
@0 = private unnamed_addr constant [2 x i8] c"x\00"

define i32 @main() {
entrypoint:
  call void @symboltable_new(i64 2)
  br label %programmain

programmain:                                      ; preds = %entrypoint
  %0 = call %struct.Object* @read_int()
  call void @symboltable_push(i64 1, %struct.Object* %0)
  %1 = call %struct.Object* @eval_identifier(i64 1, i8* getelementptr
    inbounds ([2 x i8], [2 x i8]* @0, i32 0, i32 0))
  call void @symboltable_pop(i64 1)
  call void @display_any(%struct.Object* %1)
  ret i32 0
}
```

Notice the declaration of `x` at the top of the program, the pushing of the symbol table before the `let` is processed, and the popping once it is done being processed. This serves as a simple, concrete example of how we process `let` statements. The idea behind the use of the symbol table here extends to all programs. Additionally, note that the string name of the variable is still preserved. This is required for error reporting, as giving a seemingly arbitrary integer index into the symbol table to the

user during an error is essentially useless.

To implement integers, lists, functions, and strings, we introduced a single polymorphic object struct. We designed it as a union type of name `Object`, which allowed us to generalize the values being passed around. Since we autobox all values in the original program into this general type and unbox them at runtime, all type checking is deferred to the runtime. This is a marked departure from other compiled programming languages without type annotations (for instance, Haskell and OCaml use type inference to detect and report possible type errors during compilation), and it is one of the biggest areas that could potentially be optimized on in future iterations of this compiler.

# 3 Correctness and Testing Methodology

For this project, verifying correctness involves devising and executing a thorough testing methodology. Since we implement the same operational semantics as specified in the L manual (bar one exception, which will be described shortly), the compiler should always output a new program that has the same end result and side effects as the original. Hence, it remains to test the compiler on a variety of interesting test cases.

By personal choice, our implementation of L, henceforth dubbed L-lvm, changed the dynamic scoping of lambdas over to static scoping. The static scope is captured at the declaration of the lambda. So the change in the operation semantics would be:

$$\frac{}{E \vdash \lambda id.\ S : \langle \lambda id.\ S, E \rangle} \ (Lambda)$$

$$\frac{\begin{array}{c} E \vdash e_1 : \langle \lambda id.\ e_1', E' \rangle \\ E' \vdash e_1'[e_2/id] : e \end{array}}{E \vdash (e_1\ e_2) : e} \ (Application, single)$$

$$\frac{\begin{array}{c} E \vdash e_1 : \langle \lambda id.\ e_1', E' \rangle \\ E' \vdash e_1'[e_2/id] : e \\ E' \vdash (e\ e_3\ ...\ e_k) : e' \end{array}}{E \vdash (e_1\ e_2\ e_3\ ...\ e_k) : e'} \ (Application, multiple)$$

As shown above, lambdas on creation capture the current environment and are paired with this environment from then on. On application, the first expression has to evaluate to a lambda-environment pairing. The lambda's environment is then used

when the substitution occurs to restore the state to when it was when the lambda was created.

Our testing consisted of running the public tests for PA3 which did not have differences in output based on static or dynamic scoping. This allowed us to ensure that the majority of our constructs were correct. To test static scoping, consider the following program:

```
let f = (lambda y. let y = 5 in (lambda x. y+x) 1) in
let y = 3 in
(f 7)
```

Under the semantics specified in the L manual, this program should return `10`; however, under our new semantics, it should return `12`. We used tests similar to this to verify the change in semantics we implemented. Our full test suite can be found in the `tests` folder in the project submission.

# 4 Results

In order to thoroughly evaluate the benefits of a compiler over an interpreter, a variety of workload programs should be used on both the compiler and interpreter. The workloads we chose to test were:

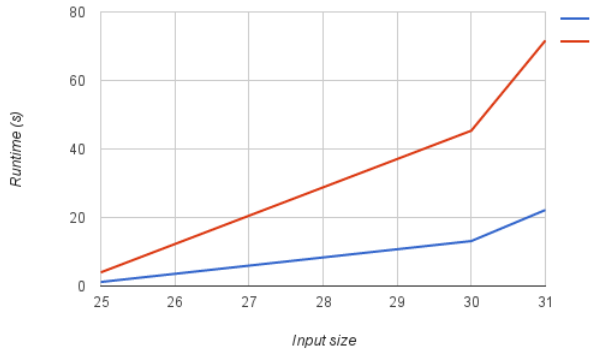
- Recursion-heavy programs. Specifically, we tested the speedup of the computations of large Fibonacci terms.
- Intensive list operations. We sorted lists of various sizes and observed the speedups under these cases. Additionally, we tested the construction of large lists to observe the speed of the `cons` operator.
- Large quantities of basic operations. We tested varying quantities of every basic operator.

We ran the tests on a computer running Arch Linux on a 2.4 GHz Intel Core i7-5500U with 8 GB RAM. We will explore the results of these tests in the order we specified them in. To compute our mean weighted speedup, we used the following formula:

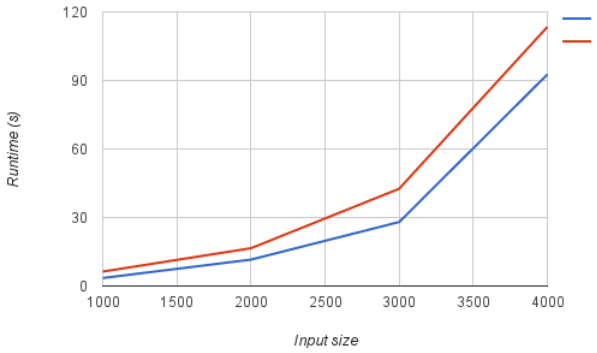
$$\bar{x} = \frac{\sum_{i=1}^{k} n_i p_i}{\sum_{i=1}^{k} n_i}$$

where $k$ is the number of input sizes we tested, each $n_i$ represents each individual input size, and each $p_i$ is the percent speedup of the compiled program over the interpreter for a given input size $n_i$.

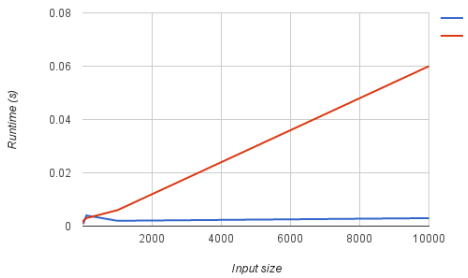
First, consider the recursion-heavy (Fibonacci) tests. The graph of the results is given below. In every subsequent graph, the blue line represents the runtime of the compiler, and the red line represents the runtime of the interpreter.

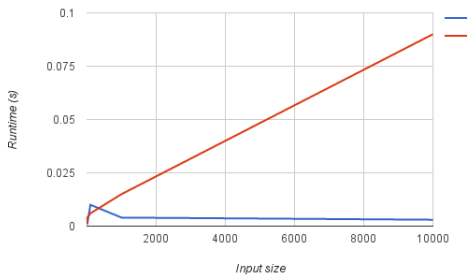



Notice that while both forms of execution experience exponential growth in runtime, the interpreter's runtime grows at a faster rate than that of the compiled program's. All our tests, bar one, experience similar behavior.

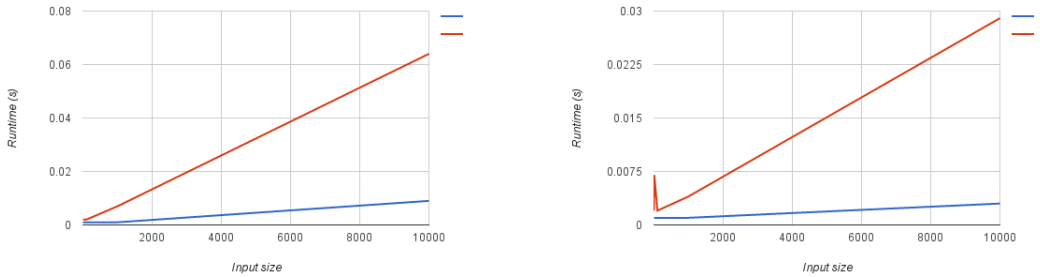



Similar behavior can be seen in the sorting benchmark, shown above. This test used quicksort on a list of varying size.

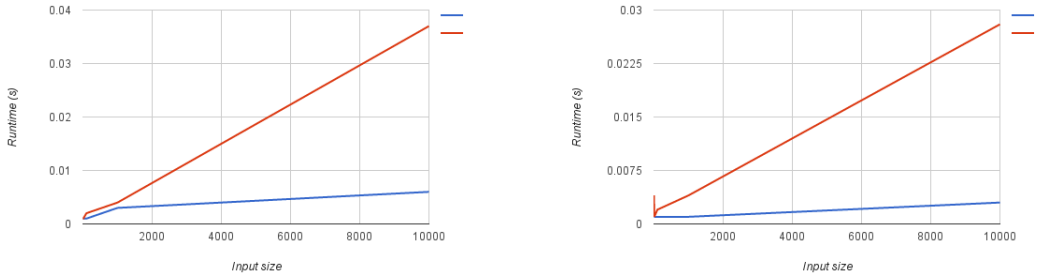


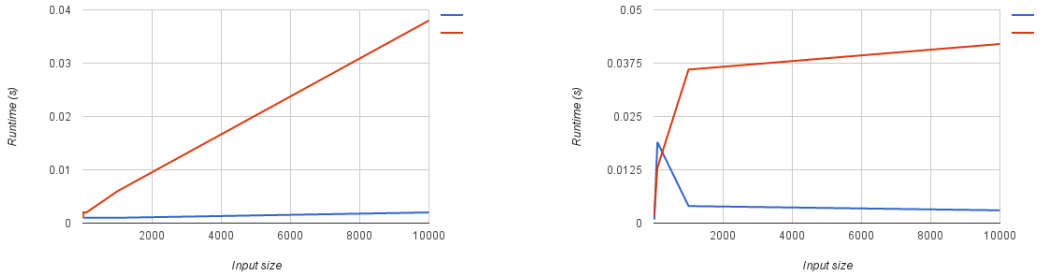
The left graph displays the runtimes for many additions, and the right graph displays the runtimes for many subtractions.



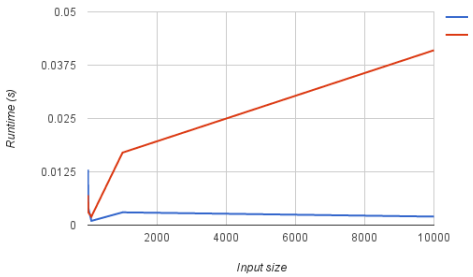

Left: Multiplications. Right: Divisions

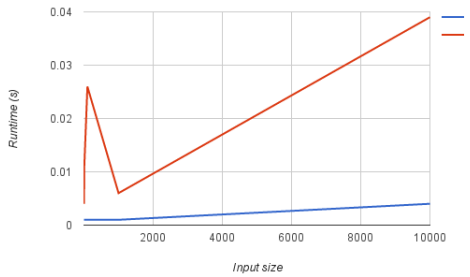



Left: Greater than comparison. Right: Greater than or equal to comparison

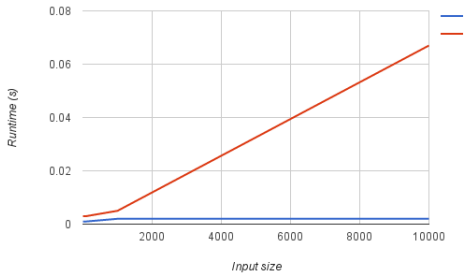


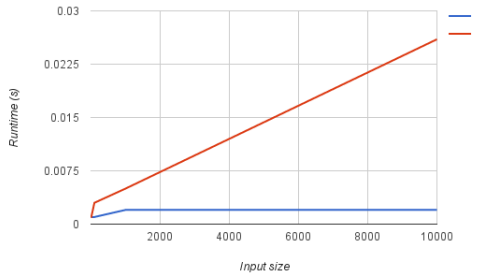

Left: Logical and. Right: Logical or.



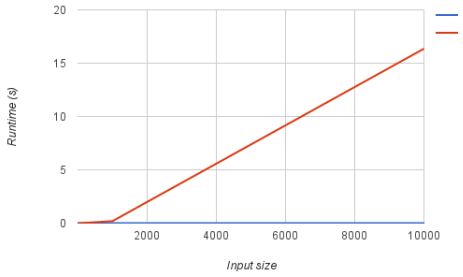



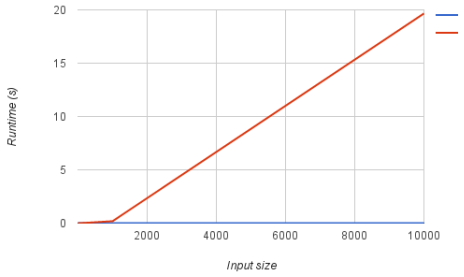

Left: Less than comparison. Right: Less than or equal to comparison



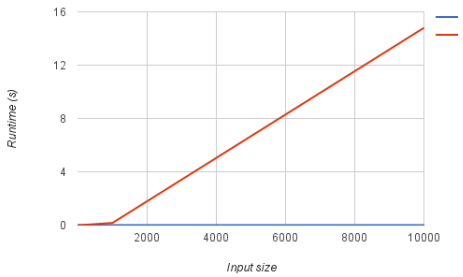



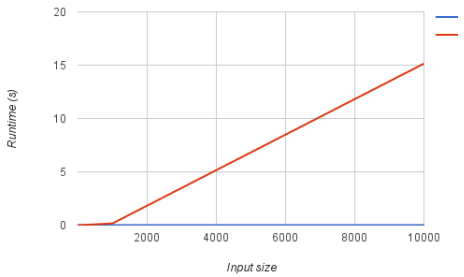

Left: Not equal to comparison. Right: Equal to comparison.



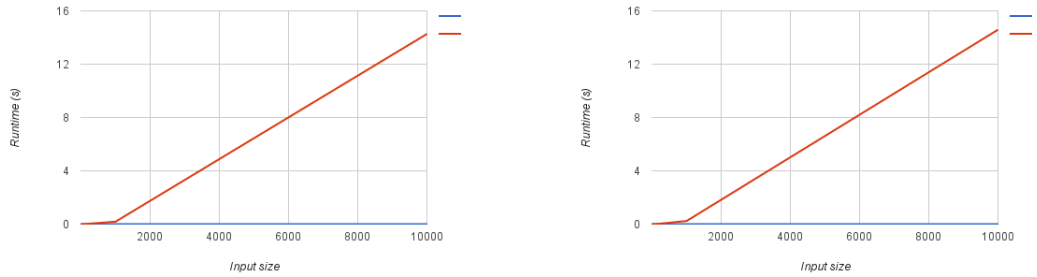




Left: Using `isNil` on a list. Right: Using `isNil` on a non-list.

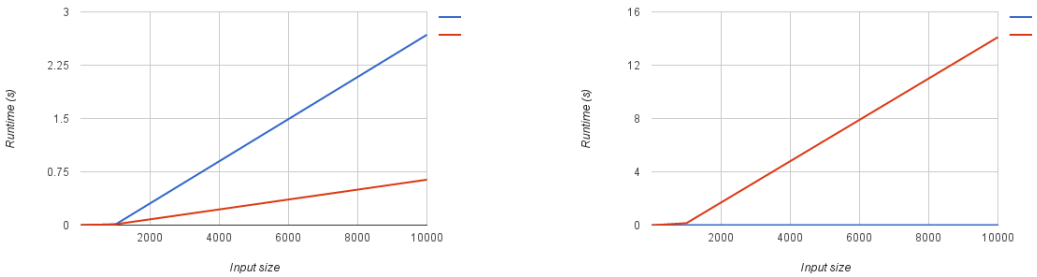


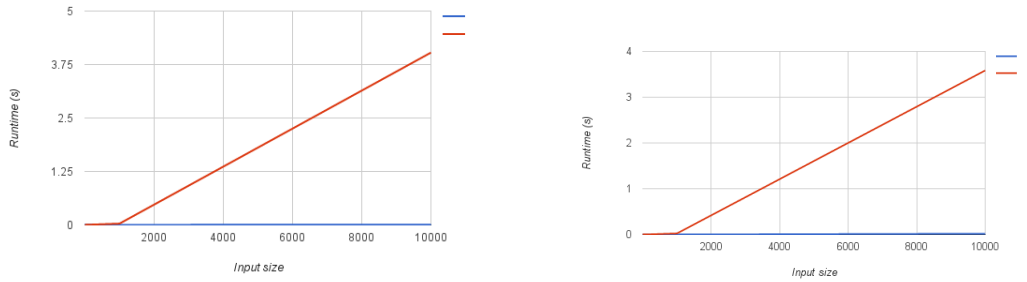
Left: Finding the head of a variable that wasn't a list. Right: Finding the head of a variable that was a list.



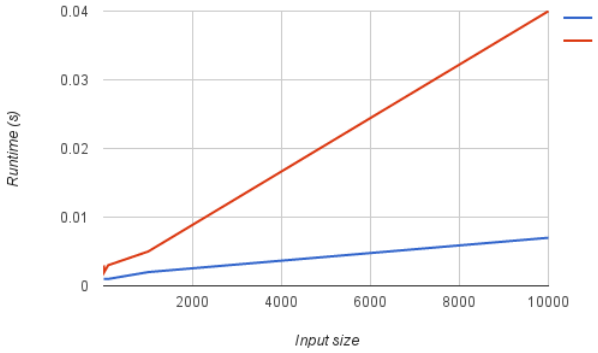

Left: Finding the tail of a variable that wasn't a list. Right: Finding the tail of a variable that was a list.




Left: Calling a function with an increasingly large namespace. Right: Performing function applications.




Left: `readInt`. Right: `readString`




Above: Cons

Impressively, the compiled programs enjoy runtimes far lower than those of the interpreter for the same programs in nearly every case. This result is promising, as it indicates that the uses of L can theoretically be extended far beyond the tasks that the interpreter can handle. Furthermore, the results strongly validate our hypothesis that a machine language version of a program runs faster than a more high-level version of the program meant for an interpreter. Additionally, since there are known further optimizations that could be made to this compiler, these results serve as a lower bound for the improvements that can be made.

However, one case we ran demonstrated a weakness of our compiled solution. The function test, which creates a namespace of size $O(n)$ and calls one function from that, ran far slower as a compiled program over an interpreted program. We believe this is because application runs in linear time with respect to the size of the namespace since the lambda binds everything defined in its local symbol table. However, in practice, this performance loss is compensated for by all the performance gains of the other operations since the other binary and unary operators are used at a higher quantity than the size of the function namespace.

We also evaluated compile time of these programs, and we found them to get quite high as the programs got larger. This is acceptable, as a program is likely run far more often than it is compiled. While we didn't do this, we could analyze the breakdown of the compilation time by running the lexer, parser, compiler to LLVM IR, and compiler from LLVM IR to elf independently. This would provide us some insight as to where the compiler is spending most of its time. As mentioned earlier, we conjecture that the compile time is quite high because of the attempts at optimizations, but we are unsure of the validity of this.

# 5 Challenges

We found the implementation of this compiler to be far more challenging than we originally expected. Because of this, we had to make some interesting design decisions early on. There were also other difficulties we encountered while doing performane testing, which prompted us to make further optimizations. Particularly pesky were problems involving a slow symbol table, the implementation of branches, and the implementation of lets and lambdas.

The first full iteration of the compiler lost to the interpreter when the set of variables in the program was large. We traced the source of this issue to our implementation of the symbol table, which we proceeded to optimize. For simplicity's sake, the symbol table implementation was originally an open source hash map. Due to the requirement to hash a string and potentially chain values due to collisions, the hashmap was unbearably slow under many test cases (on average, we found that this implementation resulted in the compiled program running 3 to 4 times slower than the interpreted version). We did not immediately employ the array of stacks idea because we could not figure out how to implement that in a single pass because the symbol table has to be at the top, but it is not computed in its entirety until the entire program is parsed. To resolve this, a label was placed at the beginning, the program was parsed, and then the symbol table was generated at the label with a branch, jumping from the start (which was the entry point) to the generated program code.

Particularly difficult was the implementation of branches; specifically, we had to determine how to encode the branching using LLVM syntax. One idea was to generate functions for every if, else, and then blocks, but we quickly realized that this could be accomplished using a phi node. The blocks for `then` and `else` are generated and the values are computed inside of those blocks. The phi node chooses which value to accept based on which was the previous block.

The implementation of lets and lambdas required a lot of complex thought because of how they bound. Lets were more straightforward because they just need to bind the value and push to the global, current symbol table. With lambdas, the substituted value has to bind over the global value. We accomplished this by pairing every single lambda with the current symbol table. Whenever the application occurs, the value is bound to the lambda's symbol table and everything defined in the lambda's symbol table is pushed onto the global symbol table to reset to that previous state and the lambda's function body is then executed. Whenever the lambda is done and about to return its resultant value, it pops all the values from its symbol table off the global symbol table to return the global state back to what it was prior to the lambda application.

Another major issue arose with the lazy evaluation of lambda parameters. We resolved this by wrapping all arguments in the application as a function that was marked in the symbol table as lazy. Whenever the name was accessed, there was an identifier function that took in the identifier unique number and checked the current stack for that variable. If the top element of the stack was marked as lazy, it would immediately call that function and execute; otherwise it would return whatever value was held inside.

One major design decision (the idea to use the general `Object` type for boxing) was heavily influenced by a problem we encountered early on. Consider the following program:

```
fun foo with x =
    if x < 0 then "hello" else 3
in

(foo -1) + (foo 1)
```

Clearly, there is a runtime error here since `(foo -1)` returns ``hello'' while `(foo 1)` returns `3`. However, without type inference, it is not possible to report the type error here during compile time. Doing so would require type annotations, and that would restrict the set of programs greatly. In particular, since this function returns two different types under two different cases, this function would be invalidated. The final solution therefore involves deferring all type checking to the runtime and boxing all these types as `Objects`. However, implementing type inference in some capacity might be able to catch some of these errors (but not necessarily all of them). Given more time, implementing type inference with this compiler is a route of interest, as it entails performance benefits, better verification of L-lvm programs, and better compile-time messages to the end user.

# 6 Conclusion

For our final course project, we proposed to develop a compiler for the L programming language. The motivation for writing a compiler stems from the notion that a program compiled to machine language runs faster than an interpreter interpreting the original source. As a result, we hypothesized that by writing a compiler for the L language, we would experience some significant performance gains when compared to the interpreter. After the development and testing of this compiler, we concluded that our original hypothesis is indeed correct. We enjoy an average speedup of nearly 77% in all tests, and except for the large namespace function call test, every mean weighted speedup was above 80%.

There still exists room for optimization, however. The implementation of type inference would speed up the running time of the programs since the need to box and unbox data would largely disappear. Additionally, a detailed analysis of where the compiler is spending its runtime would provide some insights into why compilation takes a long time in larger programs.

We felt that this project was a worthwhile endeavor, as we learned a lot about programming languages throughout, carefully considered large scale design decisions, and exercised our programming and testing skills. Additionally, we had a lot of fun with this project and we were happy with the fact that we obtained a positive result at the very end.