Spring 2018

Less-java, more learning: Language design for introductory programming

Zamua Nasrawt
James Madison University

Less-Java, More Learning: Language Design for Introductory Programming

---

An Honors College Project Presented to

the Faculty of the Undergraduate

College of Integrated Science and Engineering

James Madison University

---

by Zamua Osman Nasrawt

May 2018

Accepted by the faculty of the Department of Computer Science, James Madison University, in partial fulfillment of the requirements for the Honors College.

FACULTY COMMITTEE:

Project Advisor: Michael O. Lam, Ph.D.
Assistant Professor, Computer Science

Reader: John C. Bowers, Ph.D.
Assistant Professor, Computer Science

Reader: Christopher J. Fox, Ph.D.
Professor, Computer Science

HONORS COLLEGE APPROVAL:

Bradley R. Newcomer, Ph.D.,
Dean, Honors College

PUBLIC PRESENTATION

This work is accepted for presentation, in part or in full, at the Computer Science Department Research Seminar on

April 13, 2018.

Contents

1 Introduction **5**

1.1 Desired Features 5

2 Background **8**

2.1 Learning Languages 8

2.2 Scripting Languages 8

2.3 Compilers 10

3 Methods **12**

3.1 Development Environment 12

3.2 Implementation Details 12

3.3 Data Types 13

3.4 Collections 14

3.5 Type Inference 16

3.6 Unit Testing 17

3.7 OOP Support 17

4 Results **19**

4.1 Test Suite 19

4.2 I/O Examples 19

4.3 OOP Examples 21

4.4 Unit Testing Examples 21

4.5 Larger Programs 21

5 Future Work **26**

6 Conclusion **29**

A Decaf Grammar (ANTLR) **30**

B Generated Java Code **35**

C Collections **38**

List of Figures

1 Basic “Hello, world!” program in Java . . . 5
2 In-line unit test demonstration . . . 6
3 Various assignment statements in Grace . . . 8
4 A simple program in Scratch. . . . 9
5 Unintuitive Boolean evaluations in JavaScript . . . 9
6 A demonstration of the Counter class in Python . . . 10
7 Ten different ways to calculate the sum of an array of integers in Ruby . . . 11
8 A wrapper around the == operator in Less-Java . . . 16
9 A simple unit test in Less-Java . . . 17
10 A simple class in Less-Java . . . 18
11 A simple Hello, World! program in Less-Java . . . 19
12 A simple Hello, World! program in Java . . . 19
13 A simple Hello, World! program in Python . . . 19
14 Write to standard out in Less-Java . . . 20
15 Write to standard out in Java . . . 20
16 Read from standard in in Less-Java . . . 20
17 Read from standard input in Java . . . 21
18 OOP in Less-Java . . . 22
19 OOP in Java . . . 23
20 Testing in Less-Java . . . 23
21 Testing in Java . . . 24
22 A solution to the 3n + 1 problem in Less-Java . . . 25
23 Object oriented ambiguity in Less-Java . . . 27

Abstract

Less-Java is a new procedural programming language with static, strong, and inferred typing, native unit testing, and support for basic object-oriented constructs. These features make programming in Less-Java more intuitive than traditional introductory languages, which will allow professors to dedicate more class time to overarching computer science concepts and less to syntax and language-specific quirks.

1 Introduction

Introductory computer science courses lay the groundwork for future courses by teaching fundamental problem-solving techniques and introducing basic algorithms. Furthermore, they teach fundamental programming concepts like loops, conditionals, and data structures.

Many CS departments choose to use mainstream languages like Java in the introductory courses, and there are good reasons to choose Java. It is ubiquitous in industry, available on many platforms, and provides an extensive standard library.

Unfortunately, there are drawbacks to using Java as an introductory language. Even simple Java programs are very verbose and unintuitive to beginners, with required class declarations, long method signatures, and required semicolons. Students must write a lot of boilerplate code before actually implementing a program, which is a roadblock to learning. Figure 1 shows the classic "Hello, World!" assignment in Java, but the highlighted section is the only portion of code that directly pertains to the assignment.

```
public class HelloWorld
{
    public static void main (String[] args)
    {
        System.out.println("Hello, world!");
    }
}
```

Figure 1: Basic "Hello, world!" program in Java

1.1 Desired Features

One goal of this project was to devise a language with a simpler, less verbose syntax that allows more of the text in a program to represent the algorithm the programmer is trying to express. This is beneficial because it allows the programmer to spend less time writing boilerplate code and debugging syntax errors. The time savings could allow instructors to cover important programming concepts more thoroughly and bypass unnecessary concepts (like classes) until later. For example, the time spent differentiating between different styles of loops (for, while, do-while, etc.) and control structures (if/else, switch, etc.) is time that could have been spent solidifying their grasp of the fundamentals. The lessons learned when studying a for-loop can easily carry over to learning a while-loop, but the distinction isn't necessary early on. The same follows for other control structures. If/else structures and switch statements are similar enough that many students have trouble seeing the significance in using one over the other. Learning the differences is important eventually, but usually is not the focus of an introductory programming course.

Most programming languages provide simple I/O capabilities such as reading from standard input

and writing to standard output, but in Java even simple I/O requires an import statement ("`import java.io.*`"). This can be confusing to new programmers who have not learned about packages, name spaces, and encapsulation. File I/O is essential to interactive console computing, and so another goal of this project was to build in simple I/O functionality and make it easily accessible to the programmer.

Likewise, memory management and object-oriented constructs are crucial to modern programming, but their presence can be unhelpful to many introductory students. Things like garbage collection in Java alleviate the memory management issue, but OOP constructs like classes can still get in the way. The above "Hello, world!" program is an example; the program is entirely procedural, but Java does not allow a standalone `main` without a class declaration. However, introductory courses often cover object-oriented programming to some degree, so an entirely procedural language also isn't the answer.

Furthermore, it is commonly believed that an object-oriented approach to a problem is simple and natural for beginning programmers; however, there is some evidence that this is not always the case [1]. One goal of this project is to hide complex OOP constructs from novice programmers until they are better able to appreciate their value. Our solution is to avoid generics, interfaces, abstract classes, and inner classes, which generally go unused in introductory courses. However, we preserve the ability to bind methods to a record type, allowing the language to retain the ability to introduce fundamental concepts in object-oriented programming.

Learning good software engineering practices like unit testing is crucially important when learning introductory programming; good habits learned early pay dividends later on. Unfortunately, there is no native unit testing framework in Java; JUnit is a popular choice, but it is a 3rd-party solution. This means students must either do all of their testing in `main`, or include a jar file in their project. The former is suboptimal from an engineering perspective, and the latter is unintuitive and tedious for new programmers. Inaccessible unit testing discourages students from testing their code at all. One goal of this project is to include native *and* accessible unit testing in the language itself. Our system of in-line unit testing (see Figure 2) allows programmers to write basic unit tests for a function on the lines immediately following the function body, or to intersperse them with functions as desired.

```
add(a, b) {
    return a + b;
}

test add( 1, 2) == 3;
test add(-1, 2) == 1;
```

Figure 2: In-line unit test demonstration

In addition, it is important for beginners to think carefully about the variables they are using and the types of those variables. Strong typing enables the compiler to catch many errors early and before they impact runtime correctness. However, implicitly-typed languages are more user-friendly because

they do not force programmers to declare the type of every variable. One goal of this project was to use implicit typing with type inference [2] to lessen the verbosity of the language while retaining the software engineering benefits of strong static typing.

Less-Java is a programming language designed with the above critiques and goals in mind. More specifically, it is designed to solve a few issues that novice programmers often have when learning Java. These problems include verbose syntax, unintuitive I/O, lack of convenient unit testing, and an expansive, intimidating library that is required for even the simplest data structures like lists and maps. The hope is that students can begin to write correct programs more quickly by simplifying syntax and common tasks. Approaching problems will also become simpler if the language features are in some sense restricted, and the standard library is reduced. Less-Java aims to retain just enough functionality to be useful in teaching introductory computer science material.

2 Background

This section discusses related work in two broad categories of languages: learning and scripting languages. We also include a short description of the general compilation process for readers unfamiliar with it.

2.1 Learning Languages

There are a few programming languages that attempt to address many of the critiques in the previous section and offer an environment more suitable for beginners than the mainstream languages like Java and C++.

Grace is an educational language that aims to "help novices at programming to learn how to write correct and clean code" [3]. Grace even goes as far as to enforce code style in the grammar. Misaligned brackets or improper indentation cause errors when the program is run. This is nice for people that eventually need to read the code, but it is probably a large source of frustration for students. Grace also includes fairly advanced features such as lambdas and exceptions. There are also many different ways to simply assign values to variables (var, def, implicit typing, explicit typing, etc); see Figure 3 for examples. While convenient for many programmers, these features can be confusing for novices.

```
8  // Definitions and Variables
9  def one = 1 // constant
10 def two : Number = 2 // constant with types
11
12 var i : Number := 13  // typed variable - note :=
13 var x := 4 // variable, dynamically typed
14 var z is readable, writable := "Z" // annotated to give public access
15
```

Figure 3: Various assignment statements in Grace

Scratch is another educational language often used in K-12 outreach efforts [4]. See Figure 4 for an example program in Scratch. It is a visual, block-based language designed to eliminate syntax errors entirely so that novice programmers can focus only on the logic of their program. Unfortunately, moving blocks around does not always translate well to college-level programming, as students will eventually need to type text into an editor and therefore deal with syntax errors. The limited vocabulary also makes it difficult to solve complex problems. While very helpful for introducing younger students to programming, it has limited value in a university programming course.

2.2 Scripting Languages

Some universities address the issue of the complexity of Java and C++ by using scripting languages with less strict syntax such as JavaScript, Ruby, and Python to focus on concepts. It may seem intuitive that simpler syntax translates to a lower cognitive load, but that does not appear to be the case. Previously mentioned critiques aside, students may actually struggle more when the syntax abstracts

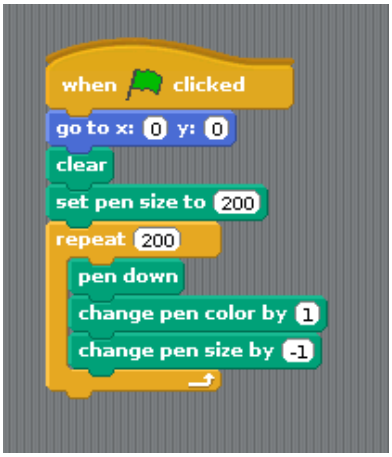

Figure 4: A simple program in Scratch.

away underlying details [5].

These scripting languages are also usually dynamically typed. While that can be a very powerful feature for an experienced developer, it is often confusing and error-prone for novices. Less-Java preserves the expressiveness of scripting languages while also being statically typed and encouraging the programmer to be mindful of what's going on under the hood.

For instance, JavaScript is a popular first language because of its utility in web development, but one that should be avoided when teaching novice programmers. Dynamic, weak typing and lack of standardization across implementations are serious problems that cause JavaScript to be very hostile to students. As a concrete example, consider one JavaScript feature that allows certain non-Boolean values to be used in Boolean expressions. These values are referred to as "truthy" and "falsey" values. Figure 5 demonstrates how this feature can be inconsistent and confusing.

```
> [] == 0
< true = $1
> "\t" == 0
< true = $1
> "0" == 0
< true = $1
> [] == "\t"
< false = $2
> [] == "0"
< false = $2
> "\t" == "0"
< false = $2
```

Figure 5: Unintuitive Boolean evaluations in JavaScript

Python has a more relaxed syntax and has a very large, expressive library. In fact the library is so large that students in search of online Python help often only find posts encouraging them to import a library that performs the task they are attempting to implement. The frequency map assignment is a great example. Constructing a frequency map of some sort is a common beginner programming assignment. Often this involves mapping characters in a string to the number of times they appear in that string. Most languages provide a map implementation either natively or in the standard library, and it's expected that this assignment will get students familiar with how to interface with a map. Should a student get confused and look for help online, they'll encounter numerous online discussions instructing them to use the `Counter` class (see Figure 6). It is a convenient class, but it devalues the assignment. Paired with required indentation in blocks, operator overloading, and other syntax quirks like dunderscores (`__`), it can become difficult for students to focus on the main concept in any given assignment.

```
>>> from collections import Counter
>>> frequencies = Counter(list("abbc"))
>>> frequencies
Counter({'a': 1, 'c': 1, 'b': 2})
>>>
```

Figure 6: A demonstration of the Counter class in Python

Ruby fulfills more of our requirements than most of the other languages discussed in this section, but it is not without its own problems. The main problem with Ruby from the perspective of an introductory class is its writability and the fact that there are so many ways to accomplish the same task. Any assignment could be written imperatively, functionally, or with a library, not to mention the expansive list of differing looping mechanisms (see Figure 7). The writability of Ruby is convenient for experts but potentially confusing and distracting for beginners.

2.3 Compilers

The reference implementation of Less-Java is compiled, and the reference compiler is implemented in Java. The phases of the Less-Java compiler are similar to those of any modern compiler, and include lexing, parsing, analysis, and code generation.

Traditionally, the lexing phase takes source code as input and constructs a queue of meaningful tokens as output. The parsing phase takes the lexer generated queue of tokens as input and outputs a parse tree, which is an arrangement of the tokens in a hierarchical format that more closely matches the semantics of the language. During the analysis phase, the parse tree is first converted to an Abstract Syntax Tree (AST) by eliminating unnecessary syntactic nodes (like semicolons or commas), and then the AST is checked for type errors and other properties (such as the presence of a `main` function). Finally, a code

generation phase converts the AST into target source code. Other traditional phases such as optimization are left out of our compiler due to time restrictions.

```
a = [1,2,3]

sum = a.reduce(:+)                  #reduction
sum = a.inject(:+)                  #also reduction
sum = a.sum                         #native in some versions of ruby

sum = 0
0.upto(a.length-1).each do |i|  #indexed iteration w/ upto
    sum += a[i]
end

sum = 0
(0..a.length-1).each do |i|     #indexed iteration w/ sequence
    sum += a[i]
end

sum = 0                             #for each
for i in a do
    sum += i
end

sum = 0                             #indexed for with sequence
for i in 0..a.length-1 do
    sum += a[i]
end

sum = 0                             #indexed while
i = 0
while i < a.length do
    sum += a[i]
    i+=1
end

sum = 0                             #indexed until
i = 0
until i == a.length do
    sum += a[i]
    i+=1
end

sum = 0                             #indexed 'loop'
i = 0
loop do
    sum += a[i]
    i+=1
    break if i == a.length
end
```

Figure 7: Ten different ways to calculate the sum of an array of integers in Ruby

3 Methods

3.1 Development Environment

The Less-Java reference compiler is written in Java which allows it to interface with the ANTLR lexer and parser, and facilitates the heavily used visitor pattern. Other development tools include Git/Github for version control, the Gradle build tool, and the ANTLR lexer and parser generator.

3.2 Implementation Details

The Less-Java source code borrowed from Decaf [6] source code initially, but it was heavily modified throughout development. Using this code as a starting point saved lots of time, and many of the AST node classes translated well. Some of the visitors were also left almost untouched, such as the visitor that populated the symbol tables.

Other changes were significant. For instance, the lexing and parsing phases were automated with ANTLR [7]. Instead of hand coding the lexer in an ad-hoc manner with regular expressions, we defined tokens and the grammar of the language in the ANTLR format (see Appendix A). ANTLR then reads this file and generates the lexer and parser in Java using LL(*) parsing [8], which allows for arbitrary look ahead without sacrificing performance. This technique allows for tremendous flexibility in defining the grammar. It still can't handle left recursion like other LL(k) algorithms, but can handle the Less-Java grammar.

At the beginning of the analysis phase, we first convert the parse tree to an abstract syntax tree (AST). This is done by a hand-coded visitor that walks the parse tree and constructs a new tree by throwing away insignificant nodes such as those representing commas or parentheses. The latter portion of the analysis phase focuses on creating symbol tables and type inference. This phase is composed of multiple alternating passes over the AST that generate symbols from the nodes in the AST and then infer those nodes' types using the symbols. These inferred types are then reflected in the next symbol building phase. This symbol generation and type inference process is repeated until the types of the symbols stop changing.

Finally, the code generation phase converts the AST into Java source code. This phase is greatly simplified by using a high level language as the target. Low level operations like linear code generation and register allocation are simply handled by Java's compiler. During code generation, non-OOP constructs (top-level functions and unit tests) are translated to Java code in a `Main` class. Regular functions are emitted as public static methods with their Less-Java name and the inferred parameter types. Unit tests are emitted as JUnit test methods with the appropriate assertion. OOP constructs are handled as described in Section 3.7.

While this code generation phase is considered the "final" phase in the Less-Java compiler, the newly

generated Java source code must also be compiled (with javac) before it can be executed. This post-compilation phase is added to the Less-Java compiler, so that the programmer does not need to do it manually.

The compiler source code is generally composed of three parts: AST nodes, AST visitors, and miscellaneous types. The AST nodes encapsulate the nodes defined in the grammar, and are split up into expressions and statements. The AST visitors define operations that are performed on the AST. The visitor pattern is nice in this regard as it decouples the AST nodes from the actual operations performed on them.

As discussed earlier, the parse tree output by ANTLR's generated parser is converted to an AST with a visitor. Once the AST is generated, there are two visitor passes that are done to give more context to each node. The first pass gives every node in the AST a pointer to its parent. This is useful if some context is required for future operations. The constructed symbol tables will be used mostly for type inference. Often, symbol tables are used mainly for static analysis, but the static analysis phase is left out in this iteration due to time limitations.

The final analysis phase implemented in this project is type inference, described in Section 3.5. Once type inference has terminated, there is enough information to begin generating code. This final code generation phase is also encoded as a visitor. This visitor focuses on the statements. When the visitor begins, it emits a standard list of imports that may or may not be used, a class signature, and a main method signature. Then, as nodes are visited, the visitors emit lines of Java code to appropriate lists which are combined in the end to form the final Java source.

3.3 Data Types

Less-Java supports four native data types as well as a handful of native collections (described in Section 3.4). The four native data types are integers, double-precision floating point numbers, booleans, and strings. Currently, there is no way to explicitly cast a data type to another data type. However, operations like addition and subtraction implicitly widen an Integer to a Double when the two types are mixed. The available operations for each of the data types are as follows:

- integer, double
  - addition (`+`)
  - subtraction (`-`)
  - unary-negation (`-`)
  - multiplication (`*`)
  - division (`/`)
  - remainder (`%`)

  - greater than (`>`)
  - greater than or equal to (`>=`)
  - less than (`<`)
  - less than or equal to (`<=`)
- boolean
  - or (`||`)
  - and (`&&`)
  - negation (`!`)
- string
  - There are no native operations for strings (to append strings, use the `format` built-in function)

In addition to the above operators, the equality operators (`==`, `!=`) operate on all of these types.

3.4 Collections

CS1 courses often ask students to use simple collections, while ignoring the implementation details or deferring their discussion to future courses. Rarely do programming languages include these collections natively. Often their use requires an import, and in some cases (like C), there aren't any native collections at all and students are left implementing their own.

In Less-Java, there are three native collections: Lists, Sets, and Maps. They can be constructed with a call to an appropriate constructor, or by using initialization operators (such as brackets for lists). The constructors can be passed another collection to initialize it with values. These collections are instantiated to specifically-defined collection types based on the constructors/operators during type inference. During code emission they are emitted using some hand-written wrapper Java classes. The wrapper classes were written to conveniently translate from the desired Less-Java collection interfaces to the standard Java Collection interfaces.

The following symbols will be used to define the collection interfaces:

- (): the empty set (zero parameters, or void return type)
- $\rightarrow$: read as "returns"
- α, β: type variables

The syntax for an element of an interface will be:

- <element-name> ":" <parameters> $\rightarrow$ <return type>
- <element-name> ":" <owner> $\rightarrow$ <parameters> $\rightarrow$ <return type>

The list collection provides the following interface:

- [] : () → α *list* (constructor)
- [] : $\alpha, \alpha, \alpha...$ → α *list* (constructor)
- list : () → α *list* (constructor)
- list : α *list* → α *list* (constructor)
- list : α *set* → α *list* (constructor)
- add : α *list* → α → ()
- push : α *list* → α → ()
- enqueue : α *list* → α → ()
- remove : α *list* → α → α
- pop : α *list* → α
- dequeue : α *list* → α
- insert : α *list* → (**int**, α) → ()
- removeAt : α *list* → **int** → ()
- get : α *list* → **int** → α
- set : α *list* → (**int**, α) → ()
- size : α *list* → **int**

The set collection provides the following interface:

- {} : () → α *set* (constructor)
- {} : $\alpha, \alpha, \alpha...$ → α *set* (constructor)
- set : () → α *set* (constructor)
- set : α *list* → α *set* (constructor)
- set : α *set* → α *set* (constructor)
- add : α *set* → α → ()
- remove : α *set* → α → α
- contains : α *list* → α → **boolean**
- size : α *set* → **int**

The map collection provides the following interface:

- <>: () → (α, β) *map* (constructor)
- <>: $\alpha \Rightarrow \beta, \alpha \Rightarrow \beta, \alpha \Rightarrow \beta...$ → (α, β) *map* (constructor)
- map : () → (α, β) *map* (constructor)
- map : (α, β) *map* → (α, β) *map* (constructor)
- put : (α, β) *map* → (α, β) → ()
- get : (α, β) *map* → α → β
- remove : (α, β) *map* → α → β

- contains : $(\alpha, \beta)\ map \rightarrow \alpha \rightarrow$ **boolean**
- size : $(\alpha, \beta)\ map \rightarrow$ **int**

Appendix C is a comprehensive example of collections in Less-Java.

3.5 Type Inference

The next significant phase is type inference, which is the process of assigning types to expressions based on context and use as opposed to user declarations. The underlying type system is composed essentially of three kinds of types: variable, base, and object. The inference algorithm is loosely based on Algorithm W for the Hindley-Milner type system [9]. All expressions begin with a variable type (e.g., their type is unknown). Then, iteratively, their types are inferred. After every round of inference, the symbol tables are updated to represent the newly-inferred types of the symbols. There must also be a check to determine if any types changed. If no types change after a round of inference, then iteration stops, and we can assume that no more information can be inferred. Generally, the type rules are that a variable type can be unified to any of the other types, a base type can only be unified to the same base type, an object type can only be unified to the same object type. Literals can be immediately typed as base types, and constructor calls are typed as object (specific to their class). From there, operators and function calls further restrict types.

There must also be a step to add new function nodes to the AST every iteration. This must be done in the case that a function exposes parametric polymorphism. One (admittedly trivial) example would be a function that wraps a check for equality. The `==` operator may be used for comparing expressions with various types. A program like in Figure 8 would not provide enough context to strictly unify a and b to base types or class types because `==` itself only requires that the operands be of the same type, but not any specific type. Their types would remain variable. Their types would only be resolved once `eq` is invoked. An invocation of `eq` would result in a unification between the type arguments and the type parameters. The resulting unified types would be applied to a new instance of the function. This function would then finally be added to the AST. Two invocations of `eq` with different type arguments would result in two new instances of `eq`.

```
eq(a, b) {
        return a == b
}
```

Figure 8: A wrapper around the == operator in Less-Java

3.6 Unit Testing

Less-Java implements elementary unit testing natively. Unit tests are composed of the keyword `test` followed by any Boolean expression. An example can be seen in figure 9. These unit tests can only exist at the top level of the program, and cannot be present inside class definitions. During code generation, these statements are translated into simple JUnit test methods. This facilitated development because the testing framework didn't need to be developed from scratch, and as an added bonus the test output is nicely formatted by the JUnit library. There are also some drawbacks that are discussed along with potential resolutions in Section 5.

```
add(a, b) {
    return a + b
}

test add(1, 2) == 3
```

Figure 9: A simple unit test in Less-Java

3.7 OOP Support

Less-Java is a largely procedural programming language, but it also has some OOP constructs. More specifically, Less-Java supports class definitions that can contain a constructor, attributes, and methods, with single inheritance. In addition to the native int, double, boolean, string, and collection types, new types can be defined through these class definitions. An example can be seen in Figure 10.

OOP code (classes, methods) goes through a translation process to become Java classes. The appropriate boilerplate is generated, the types of attributes are inferred and added to their declarations/instantiations, method return types are inferred and injected into the method signatures, and constructors are generated if they're missing (as might be the case with inheritance).

```
Car {
    public make = ""
    public model = ""

    Car(make, model) {
        this.make = make
        this.model = model
    }

    public toString() {
        return format("%s %s", this.make, this.model)
    }
}

test Car("Toyota", "Camry").toString() == "Toyota Camry"
```

Figure 10: A simple class in Less-Java

4 Results

This section contains many examples of Less-Java code to demonstrate the various features of the language.

4.1 Test Suite

The Less-Java test suite is composed of thirty-two programs (over 640 lines of code) written in Less-Java, using the native unit test feature of Less-Java itself. If all of the test programs compile successfully and all of their unit tests pass, then the compiler test suite passes. It's possible (but unlikely) that there are errors in code generation in such a way that the code for the unit tests is corrupted (e.g., always returning true). Currently, we detect these false positives by manual inspection.

4.2 I/O Examples

A simple Hello, World! program in Less-Java is significantly shorter than the equivalent in Java. It's worth noting that languages like Python can still produce shorter programs. In this case, Less-Java still requires a main function which defines a clear entrance point for the programs execution. In Python, code can be executed from anywhere in the file, and this can be confusing for students that need to deal with multiple files.

```
main() {
    println("Hello, World!")
}
```

Figure 11: A simple Hello, World! program in Less-Java

```
public class HelloWorld
{
    public static void main (String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

Figure 12: A simple Hello, World! program in Java

```
print("Hello, World!")
```

Figure 13: A simple Hello, World! program in Python

In terms of output specifically, Less-Java provides three functions as shown in Figure 14. This small program behaves the same as the program displayed in Figure 15. A static import can be used to omit

the `System.` portion of the output, but explaining that to students eats into class time, so students are instead often burdened by having to write out the fully qualified name multiple times. It's also worth noting that in Less-Java it is currently possible to shadow the native functions. If a student writes their own `print` function, then that one will be used when `print` is invoked. If this turns out to be an issue, this could easily be prohibited during the static analysis phase.

```
main() {
    print("foo\n")
    println("foo")
    printf("%s\n", "foo")
}
```

Figure 14: Write to standard out in Less-Java

```
public class Foo {
    public static void main(String[] args) {
        System.out.print("foo\n");
        System.out.println("foo");
        System.out.printf("%s\n", "foo");
    }
}
```

Figure 15: Write to standard out in Java

Input is also simplified in Less-Java, as shown in Figure 16. This program behaves the same as the program in Figure 17 Notice that simple input in Java still requires an import. Changing `nextX` to `readX` makes what's happening clearer. Less-Java also provides additional utility functions that can read in a character or a word. Input seems to go against the Less-Java pattern of hiding explicit types by having things like "Int" and "Double" in the function names, but this is unavoidable because it's not possible to infer the type of input at compile time since the input doesn't exist until run time. An alternative approach would be to simply have a `read` function that always returns a string type, and then have the programmer cast the type somehow. It's unclear which method is better, but explicitly requiring the type in the function name makes it unlikely that a programmer could read something in and forget to cast it to their desired type. It also allows for input type checks at run-time because the program knows what to expect. This is the approach taken by Haskell.

```
main() {
    i = readInt()
    d = readDouble()
    c = readChar()
    w = readWord()
    l = readLine()
}
```

Figure 16: Read from standard in in Less-Java

4.3 OOP Examples

Object oriented programming (OOP) in Less-Java looks very similar to Java, but is simplified and more restrictive. An example of OOP in Less-Java can be seen in Figure 18. This program behaves the same as the program in Figure 19. Notice that the BullDog class in less Java does not define a constructor. By leaving out a constructor declaration, the BullDog class inherits it's parent's constructor. This is useful when the only thing the specialized class is doing is overriding a method. In Java, a constructor must be defined that explicitly calls `super` to achieve the same effect. Also notice that in Less-Java class attributes must be given a default value. This default value allows type inference for the attributes.

4.4 Unit Testing Examples

Unit testing is one of the most convenient features of Less-Java, where it is native (no library required) and has very simple syntax. It's not as sophisticated as JUnit for Java, but it is plenty for introductory students, and will help facilitate a test-driven development work flow. Inline unit testing in Less-Java can be seen in Figure 20. This program behaves the same as the program in Figure 21. Through a combination of implied boilerplate, simplified unit tests, and type inference, the Less-Java program is significantly shorter than the Java program. Thanks to the type inference, the `add` function doesn't need to be overloaded like it is in Java. Instead, the function is considered generic, and a new one is generated for each function call with a unique parameter list.

4.5 Larger Programs

Figure 22 shows a more complex program: a solution to the well-known $3n + 1$ problem (also called the Collatz conjecture [10]). This problem concerns a sequence where successive terms are obtained from previous terms, beginning with any positive n. If the current term is even, then the next term is $n \div 2$. If the current term is odd, then the next term is $3n + 1$. The Collatz conjecture posits that this sequence

```
import java.util.Scanner;

public class Foo {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        int i = in.nextInt();
        double d = in.nextDouble();
        char c = in.useDelimiter("").next();
        String w = in.useDelimiter("\\s+").next();
        String l = in.nextLine();
    }
}
```

Figure 17: Read from standard input in Java

will always converge to 1. For our example, we wish to calculate the maximum sequence/cycle length within a (low, high) range of integers.

In our implementation, the `cycle_len` routine calculates a single sequence length. We should be confident in our cycle length implementation before proceeding to implement a max function. A suite of unit tests would both guide development and test for regressions. In Java, this would be a tedious chore of finding and adding the required JUnit libraries to your class path, adding the necessary imports, and learning the JUnit API. In Less-Java, it's as simple as using the test operator followed by any expression. In this case, the tests are all equality expressions that compare a function call and an integer literal, but any boolean expression can be tested in this fashion.

Finally, it is important for clarity that all of the "equivalent" Java programs that have been referenced in this section have been manually translated. In actuality, the generated Java source code isn't as clean.

```
main () {
    d = Dog("Dog", 10)
    println(d.bark())
    println(d.run())

    bd = BullDog("Duke␣Dog", 20)
    println(bd.bark())
    println(bd.run())
}

Dog
{
    public name = ""
    private age = 0

    Dog (name, age) {
        this.name = name
        this.age = age
    }

    public bark () {
        return format("%s says BARK!", this.name)
    }

    public run () {
        return format("%s runs!", this.name)
    }
}

BullDog extends Dog
{
    public bark () {
        return format("%s says BARK BARK!", this.name)
    }
}
```

Figure 18: OOP in Less-Java

Appendix B shows the generated Java source code for the $3n + 1$ solution in Figure 22.

```
public class Foo {
    public static void main (String[] args) {
        Dog d = new Dog("Spot");
        System.out.println(d.bark());
        System.out.println(d.run());

        BullDog bd = new BullDog("Duke Dog");
        System.out.println(bd.bark());
        System.out.println(bd.run());
    }

    private static class Dog {
        public String name;

        public Dog (String name) {
            this.name = name;
        }

        public String bark () {
            return String.format("%s says BARK!", this.name);
        }

        public String run () {
            return String.format("%s runs!", this.name);
        }
    }

    private static class BullDog extends Dog {
        public BullDog(String name) {
            super(name);
        }

        public String bark () {
            return String.format("%s says BARK BARK!", this.name);
        }
    }
}
```

Figure 19: OOP in Java

```
add(a, b)
{
    return a + b
}

test add(1, -1) == 0
test add(2.5, 3.5) == 6.0
test add(1000, 1000) == 2000
```

Figure 20: Testing in Less-Java

```
import org.junit.Test;

import static org.junit.Assert.assertTrue;

public class Foo {
    public static int add(int a, int b)
    {
        return a + b
    }

    public static int add(double a, double b)
    {
        return a + b
    }

    @Test
    public void testAdd1() {
        assertTrue(add(1, -1) == 0);
    }

    @Test
    public void testAdd2() {
        assertTrue(add(2.5, 3.5) == 6.0);
    }

    @Test
    public void testAdd3() {
        assertTrue(add(1000, 1000) == 2000);
    }
}
```

Figure 21: Testing in Java

```
cycle_len(n)
{
    len = 1
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2
        } else {
            n = 3*n + 1
        }
        len = len + 1
    }
    return len
}

test cycle_len(1) == 1
test cycle_len(6) == 9

max_3np1_cycle(low, high)
{
    max = 0
    while (low <= high) {
        len = cycle_len(low)
        if (len > max) {
            max = len
        }
        low = low + 1
    }
    return max
}

test max_3np1_cycle(1,   10)   == 20
test max_3np1_cycle(100, 200)  == 125
test max_3np1_cycle(201, 210)  == 89
test max_3np1_cycle(900, 1000) == 174
```

Figure 22: A solution to the 3n + 1 problem in Less-Java

5 Future Work

There are many avenues for future work. Here we describe some of them.

Currently, the compiler error messages are limited to syntax errors because ANTLR handles them automatically. Less-Java static analysis is currently very bare-bones and it's possible that a program can pass Less-Java compilation but throw errors during Java compilation. In this case, the student would see error output from the Java compiler with references to generated variable names, generated line numbers, and Java-native exceptions. This is obviously very confusing to students and makes debugging tedious. Unfortunately, implementing static analysis comprehensive enough to prevent errors during Java compilation was not realistic in the project time frame. Completing this aspect of the compiler is a good project for a future student.

Similarly, I/O in Less-Java is only semi-complete. While a user can write to standard out and read from standard in, there is no way to read or write to files. It's possible to pass everything through to Java, but Java's file management interface isn't very user friendly for beginners. Developing a rich, safe interface for file manipulation is an interesting and challenging project for a future student.

The nature of the Less-Java unit testing also makes the generation of error messages difficult. This is because they compile to JUnit tests and so the test output comes from the JUnit runtime. This means the output and stack traces all refer to objects and line numbers from the generated Java file. It may be worthwhile to capture the JUnit output and map any errors back to the Less-Java source. Another option is to develop a custom, better-integrated testing harness for Less-Java.

Object-oriented programming in Less-Java is also incomplete. While type inference is successful across assignments, it becomes unstable when objects are passed as function parameters. Figure 23 is an example of one of these scenarios. In this case, `foo(x)` should print `‘‘Hello from A!’’`, and `foo(y)` should print `‘‘Hello from B!’’`. Unfortunately, it currently fails to unify the parameter `a` and therefore compilation fails.

A similar issue is faced and solved in Figure 21 for primitive types. Overloading a function to handle the different parameters is insufficient for the object oriented case. Functions with one polymorphic parameter need to be emitted once for each object type. Functions with many polymorphic parameters need to be emitted once for each combination of the parameters in the worst case. This is doable of course, but this cross-product property may generate unreasonably large compiled files for rather small source files. One way to complete this inference is to assign a list of interfaces to objects based on their methods, offloading a lot of the static analysis work to Java's interfaces. In this scheme, function parameters can be typed based on which functions are called on that object in the function. The implementation would also need to be able to handle when multiple methods are called on an object, or when there are multiple parameters. This may require a more extensive implementation of the Hindley-

Milner type inference algorithm, and would make for an interesting project for any student interested in extending the compiler.

```
main() {
    x = A()
    y = B()

    foo(x)
    foo(y)
}

foo(a) {
    a.bar()
}

A
{
    public bar() {
        println("Hello from A!")
    }
}

B
{
    public bar() {
        println("Hello from B!")
    }
}
```

Figure 23: Object oriented ambiguity in Less-Java

Unfortunately, we were unable to observe students using the language in a controlled study because of time constraints. However, we did conduct an informal experiment in the Fall 2017 semester during a competitive programming club meeting. Students were tasked with solving a previously-approved list of problems in Less-Java without having had any prior exposure to the language. There was no formal data collection, but students were able to solve every problem with limited assistance from the language author and advisor. An objective comparison between Less-Java and some of the languages mentioned in Section 2 could let us draw more significant conclusions in regards to language features and their impact on programming education.

Although we provide a Vim plugin for Less-Java syntax highlighting, most common editors and IDEs don't yet support Less-Java syntax highlighting or completion. A custom IDE for Less-Java or integration into a popular platform like Eclipse would greatly increase the user friendliness of the language, especially for beginners.

Finally, the reference Less-Java compiler has no optimization phase, and also hasn't been properly benchmarked. We conjecture that it would perform similarly to Java for most tests since we merely delegate most operations to the corresponding Java constructs, but a comprehensive performance bench-

mark may be able to expose some inefficiencies in the emitted code and address the question of whether an optimization phase in the Less-Java would help.

6 Conclusion

Current programming languages are insufficient for introductory computer science education. They can be verbose, unintuitive, confusing, or some combination of these qualities. Students spend valuable class time learning the quirks of their tools when they should be focusing on core Computer Science concepts. Our contribution is Less-Java, a succinct programming language with simple syntax, along with a reference compiler. It limits the constructs available to the programmer to avoid confusion and complexity while still providing all of the tools necessary to teach a CS1 course, including 1) type inference and static, strong typing, 2) native unit testing to encourage good testing practices, and 3) rudimentary object-oriented programming support. This project also serves as a basis for future expansion as well as analysis of language features and how they impact beginner programmers.

The full software distribution is available at the following URL: https://github.com/Zamua/less-java/

A Decaf Grammar (ANTLR)

```
grammar LJ;

/* Parser Grammar */

program:        (class_ | function | global | test | EOL)*;

class_:          classSignature classBlock;

classSignature: name=ID (EXTENDS superName=ID)?;
classBlock:     (EOL)? LCB (EOL)? (attribute | EOL)* (method | EOL)* RCB (EOL)?;
attribute:      scope=(PUBLIC|PRIVATE) assignment EOL;
method:         (scope=(PUBLIC|PRIVATE))? function;

function:       ID LP (paramList)? RP block;
paramList:      ID (','ID)*;
block:          (EOL)? LCB (EOL)? statement* RCB (EOL)?;

global:         GLOBAL assignment EOL;

test:           TEST expr EOL;

statement:      IF LP expr RP block (ELSE block)?           #Conditional
                | WHILE LP expr RP block                    #While
                | FOR LP var ':' (expr '->')? expr RP block #For
                | RETURN expr EOL                           #Return
                | BREAK EOL                                 #Break
                | CONTINUE EOL                              #Continue
                | funcCall EOL                              #VoidFunctionCall
                | methodCall EOL                            #VoidMethodCall
                | assignment EOL                            #VoidAssignment
                | EOL                                       #Terminator
                ;
```

```
expr:           exprBin;

exprBin:        left=exprBin op=PREC1 right=exprBin
                | left=exprBin op=('+'|'-') right=exprBin
                | left=exprBin op=PREC3 right=exprBin
                | left=exprBin op=PREC4 right=exprBin
                | left=exprBin op=PREC5 right=exprBin
                | left=exprBin op=PREC6 right=exprBin
                | methodCall
                | assignment
                | exprUn
                ;

exprUn:         op=('!'|'-') expression=exprUn
                | exprBase
                ;

exprBase:       funcCall
                | methodCall
                | collection
                | var
                | memberAccess
                | lit
                | LP expr RP
                ;

funcCall:       ID LP (argList)? RP;

methodCall:     (var | funcCall) op=INVOKE funcCall;

collection:     LSB (argList)? RSB       #List
                | LCB (argList)? RCB     #Set
                | PREC3 (argList)? PREC3 #Map
                ;
```

```
assignment:     (var | memberAccess) op=PREC7 expr;

argList:        entry (',' entry)*
                | expr (',' expr)*
                ;

entry:          key=expr ':' value=expr;

var:            name=ID (LSB (index=expr)? RSB )?;
memberAccess:   instance=ID INVOKE var;

lit:            INT | REAL | BOOL | STR;


/* Lexer Rules */

// Key Words
IF:         'if';
ELSE:       'else';
WHILE:      'while';
FOR:        'for';
RETURN:     'return';
BREAK:      'break';
CONTINUE:   'continue';
TEST:       'test';
GLOBAL:     'global';
PUBLIC:     'public';
PRIVATE:    'private';
EXTENDS:    'extends';

PREC1:      MULT
            | DIV
            | MOD
            ;
```

```
// Removed to prevent collisions with unop
//PREC2:       ADD
//             | SUB
//             ;

PREC3:       GT
             | GTE
             | LT
             | LTE
             ;

PREC4:       ET
             | NET
             ;


PREC5:       AND;

PREC6:       OR;

PREC7:       ASGN
             | ADDASGN
             | SUBASGN
             ;

NOT:      '!';
ADD:      '+';
SUB:      '-';
MULT:     '*';
DIV:      '/';
MOD:      '%';
GT:       '>';
GTE:      '>=';
LT:       '<';
LTE:      '<=';
```

```
ET:       '==';
NET:      '!=';
OR:       '||';
AND:      '&&';
ASGN:     '=';
ADDASGN:  '+=';
SUBASGN:  '-=';
INCR:     '++';
DECR:     '--';

LSB: '[';
RSB: ']';
LCB: '{';
RCB: '}';
LP:  '(';
RP:  ')';

INVOKE: '.';


// Literals
INT:         [0-9]+;
REAL:        [0-9]*'.'[0-9]+;
BOOL:        'true'|'false';
STR:         '\"'.*?'\"';

ID:          [a-zA-Z][a-zA-Z0-9_]*;
EOL:         '\r'? '\n';

// Ignore
WHITESPACE:      [ \t]+ -> skip;
BLOCK_COMMENT:   '/*' .*? '*/' -> skip;
LINE_COMMENT:    '//' [\r\n]* -> skip;
```

B Generated Java Code

This section contains the full generated Java code for the $3n + 1$ solution in Figure 22.

```
import static org.junit.Assert.*;
import static wrappers.LJString.*;
import static wrappers.LJIO.*;

import org.junit.Test;
import java.util.*;
import java.io.*;

import wrappers.*;
public class Main
{
    public static void main(String[] args)
    {
        println(cycle_len(Integer.valueOf(1)));
    }
    public static Integer cycle_len(Integer n)
    {
        Integer len;
        len = Integer.valueOf(1);
        while (Boolean.valueOf((!n.equals(Integer.valueOf(1)))))
        {
            if (Boolean.valueOf((Integer.valueOf((n%Integer.valueOf(2))).equals(I
            {
                n = Integer.valueOf((n/Integer.valueOf(2)));
            }
            else
            {
                n = Integer.valueOf((Integer.valueOf((Integer.valueOf(3)*n))+Inte
            }
            len = Integer.valueOf((len+Integer.valueOf(1)));
        }
        return len;
```

```
}
public static Integer max_3np1_cycle(Integer low, Integer high)
{
        Integer len;
    Integer max;
    max = Integer.valueOf(0);
    while (Boolean.valueOf((low<=high)))
    {
        len = cycle_len(low);
        if (Boolean.valueOf((len>max)))
        {
            max = len;
        }
        low = Integer.valueOf((low+Integer.valueOf(1)));
    }
    return max;
}
@Test
public void test0() {
    assertEquals(Integer.valueOf(1), cycle_len(Integer.valueOf(1)));
}
@Test
public void test1() {
    assertEquals(Integer.valueOf(9), cycle_len(Integer.valueOf(6)));
}
@Test
public void test2() {
    assertEquals(Integer.valueOf(20), max_3np1_cycle(Integer.valueOf(1), Inte
}
@Test
public void test3() {
    assertEquals(Integer.valueOf(125), max_3np1_cycle(Integer.valueOf(100), I
}
@Test
public void test4() {
```

```
        assertEquals(Integer.valueOf(89), max_3np1_cycle(Integer.valueOf(201), In
    }
    @Test
    public void test5() {
        assertEquals(Integer.valueOf(174), max_3np1_cycle(Integer.valueOf(900), I
    }
}
```

C Collections

An extensive example of collections in Less-Java.

```
listOperator() {
    list = ["a", "b", "c"]
    list.add("d")

    return list
}

test listOperator().size() == 4
test listOperator().get(0) == "a"
test listOperator().get(listOperator().size() - 1) == "d"

emptyListConstructor() {
    list = List()
    list.add("a")

    return list
}

test emptyListConstructor().size() == 1
test emptyListConstructor().get(0) == "a"
test emptyListConstructor().get(emptyListConstructor().size() - 1) == "a"

listFromList() {
    list = List(listOperator())
    list.add("e")

    return list
}

test listFromList().size() == 5
test listFromList().get(0) == "a"
test listFromList().get(listFromList().size() - 1) == "e"
```

```
listFromSet() {
    list = List(setOperator())
    list.add("a")

    return list
}

test listFromSet().size() == 5

setOperator() {
    set = {"a", "a", "b", "c"}
    set.add("d")
    set.add("e")
    set.remove("e")

    return set
}

test setOperator().size() == 4
test setOperator().contains("a")
test !setOperator().contains("e")

emptySetConstructor() {
    set = Set()
    set.add("a")
    set.add("a")
    set.add("b")
    set.add("c")
    set.remove("c")

    return set
}

test emptySetConstructor().size() == 2
```

```
test emptySetConstructor().contains("a")
test emptySetConstructor().contains("b")
test !emptySetConstructor().contains("c")

setFromSet() {
    set = Set(setOperator())
    set.remove("d")

    return set
}

test setFromSet().size() == 3
test !setFromSet().contains("d")

setFromList() {
    set = Set(listFromSet())

    return set
}

test setFromList().size() == 4

mapOperator() {
    map = <"x" : 10>
    map.put("y", 100)

    return map
}

test mapOperator().contains("x")
test mapOperator().contains("y")
test mapOperator().get("x") == 10
test mapOperator().get("y") == 100

emptyMapConstructor() {
```

```
    map = Map()
    map.put("x", 10)

    return map
}

test emptyMapConstructor().size() == 1
test emptyMapConstructor().contains("x")
test emptyMapConstructor().get("x") == 10

listAsQueue() {
    queue = ["a", "b", "c"]
    queue.enqueue("d")

    return queue
}

test listAsQueue().size() == 4
test listAsQueue().get(listAsQueue().size() - 1) == "d"
```

References

[1] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion," *Computer science education*, vol. 13, no. 2, pp. 137–172, 2003.

[2] B. C. Pierce, *Types and programming languages*. MIT press, 2002.

[3] A. P. Black, K. B. Bruce, M. Homer, J. Noble, A. Ruskin, and R. Yannow, "Seeking grace: A new object-oriented language for novices," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, (New York, NY, USA), pp. 129–134, ACM, 2013.

[4] "Scratch." https://scratch.mit.edu. Accessed: 2018-03-6.

[5] N. Alzahrani, F. Vahid, A. Edgcomb, K. Nguyen, and R. Lysecky, "Python versus c++: An analysis of student struggle on small coding exercises in introductory programming courses," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, (New York, NY, USA), pp. 86–91, ACM, 2018.

[6] "Decaf language reference." https://w3.cs.jmu.edu/lam2mo/cs432_2017_08/files/decaf_ref.pdf. Accessed: 2018-03-6.

[7] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[8] T. Parr and K. Fisher, "Ll (*): the foundation of the antlr parser generator," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 425–436, 2011.

[9] R. Milner, "A theory of type polymorphism in programming," *Journal of computer and system sciences*, vol. 17, no. 3, pp. 348–375, 1978.

[10] W. contributors, "Collatz conjecture — wikipedia, the free encyclopedia," 2018. [Online; accessed 20-March-2018].