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

Zamua O. Nasrawt and Michael O. Lam
Department of Computer Science
James Madison University
701 Carrier Drive, MSC 4103
Harrisonburg, VA 22807
nasrawzo@dukes.jmu.edu, lam2mo@jmu.edu

Abstract

We present Less-Java, a new procedural programming language with a simple and concise syntax, implicit but strong typing via type inference, and built-in unit testing. These features make programming in Less-Java more intuitive for novice programmers than programming in traditional introductory languages. This will allow professors to dedicate more class time to fundamental programming concepts rather than syntax and language-specific quirks.

1 Introduction

1.1 Project Goal

Introductory computer science courses lay the groundwork for future courses by teaching problem-solving techniques and fundamental programming concepts like loops, conditionals, and data structures. Many computer science departments use a mainstream language like Java in these courses, and there are good reasons to do so. Java is ubiquitous in industry, available on many

platforms, and provides an extensive standard library. Unfortunately, there are also drawbacks to using Java as an introductory language[1].

First, even simple Java programs are very verbose and unintuitive to beginners, with required class declarations, long method signatures ("`public static void main (String[] args)`"), and required semicolons. Students must write a lot of boilerplate code before writing program code. The boilerplate code is usually explained by the end of the course, but forcing students to write it without truly understanding it is disingenuous.

Second, Java requires all variables to be declared, forcing students to distinguish between declaration and initialization before they really even understand the concept of a variable itself. Many languages like Ruby and Python avoid this requirement by removing declarations and using dynamic typing. Unfortunately, these languages cannot be type-checked statically, which hurts novice programmers because it allows incorrectly-typed code to compile and run.

Finally, there is no native unit testing framework in Java (JUnit is a popular framework but it is a 3rd-party library). This means students must either do all of their testing in the main method or include a jar file in their project. The former is against software engineering best practices and the latter is unintuitive and tedious for novice programmers. Inaccessible unit testing discourages students from testing their code at all, which is damaging for the student and society at large [10].

To address these drawbacks, we present a new language named Less-Java with 1) a simple and concise syntax, 2) implicit but strong typing via type inference [8], and 3) built-in unit testing. The language also provides simple built-in standard I/O, a core set of built-in data structures (e.g., lists and maps), and basic object-oriented features (e.g., classes and objects). We aimed to retain just enough functionality to be useful in teaching introductory computer science courses without overwhelming novice programmers. To our knowledge, Less-Java is the first language to support all of these features.

1.2 Background

There are programming languages that attempt to address many of the critiques in the previous section. Scratch [9] and Snap [2] are educational languages often used in K-12 outreach efforts. They are visual, block-based languages designed to eliminate syntax errors entirely so that novice programmers can focus only on the logic of their program. Unfortunately, Scratch's limited vocabulary also makes it difficult to solve complex problems. Snap solves this problem by allowing definition of custom blocks, but both languages still re-

[1]To be clear, these drawbacks do not preclude it from being an important and useful language to learn in a more advanced course.

quire students to move blocks around. This does not necessarily translate well to college-level programming where students must use text-based languages.

Grace [4] is a more traditional procedural language that aims to "help novices at programming to learn how to write correct and clean code." Grace even goes as far as to enforce code style in the grammar; misaligned brackets or improper indentation cause errors when the program is compiled. This is nice for people that eventually need to read the code, but it is a source of frustration for students. Grace also includes advanced features (e.g., lambdas) that are unnecessary for most introductory programming courses.

Some universities use scripting languages such as JavaScript, Ruby, and Python, allowing students to begin programming with simpler syntax and less boilerplate. However, there is some evidence that students may actually struggle more when the syntax abstracts away underlying details [3]. In addition, these scripting languages are also usually dynamically typed, which can induce subtle type problems like inadvertently passing a string representation instead of the underlying object. Finally, scripting languages often include large standard libraries and a high degree of language expressivity. These features are valuable for experienced programmers, but for novices they tend to be a source of confusion especially when the students use online search engines to look for help.

2 Less-Java

We propose Less-Java, a simple language for novice-level programming [1]. In this section we describe the language itself and our reference compiler, which is implemented in Java and compiles Less-Java programs to Java code.

2.1 Language Design

Less-Java provides four native data types and three built-in collections. The four native data types are 32-bit signed integers, double-precision floating-point numbers, Booleans, and character strings. There is currently 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. In addition to all the standard arithmetic operators, equality operators (==, !=) operate on all of these types. The three built-in collections are lists, sets, and maps. They can be initialized with a call to an appropriate constructor (optionally providing an existing variable to copy the elements) or by providing hard-coded data using initialization operators (e.g., brackets for lists). These collections are essentially wrappers around standard Java Collection classes.

Less-Java is strongly and statically typed but does not require explicit type declarations. Instead, the compiler assigns types to expressions based on context, a process called *type inference*. The underlying type system is composed of three kinds of types: variable, base, and object. The inference is implemented by a fixed-point process loosely based on Algorithm W for the Hindley-Milner type system [5].

The process begins with all non-literal expressions bound to a variable type (e.g., their type is unknown). These type bindings are refined iteratively using constraints derived from assignments and function calls. The constraints are resolved using a process called *unification*, which attempts to make two inferred types compatible. The type rules are as follows: 1) a variable type can be unified to any of the other types, 2) a base type can only be unified to the same base type, and 3) 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 according to the corresponding object type. If two types cannot be unified, the compiler reports a type error and compilation fails. The process terminates when expression types converge.

As an example, consider the assignment "a = 2.4". Initially, the variable "a" is bound to a variable (unknown) type, but the assignment constrains the type of "a" to be assignable from the right-hand side. Because the right-hand side is a literal and therefore a base type (double-precision number), the unification binds the type of "a" to the same base type. If "a" is later passed to a function, that function parameter will also be unified (and bound) to the same base type. If the assignment "a = true" appears later in the program, the unification will fail because the type of "a" is known to be a different base type than the literal "true" (which is a boolean).

If a function exposes parametric polymorphism (i.e., one or more of its parameters or its return cannot be resolved to a non-variable type), the compiler must generate multiple copies of the the function with concrete types. This is similar to how C++ handles template functions.

Finally, Less-Java includes simple unit testing using a built-in syntax. Unit tests are composed of the keyword "test" followed by any Boolean expression. During code generation, these statements are translated into JUnit test methods. This makes it easy for students to write tests or for an instructor to include tests in a project distribution.

2.2 Implementation

The Less-Java reference compiler is written in Java, which allows it to interface with the ANTLR parser generator [6, 7], facilitating parse tree traversals and code generation. Other development tools include Git/GitHub for version control and the Gradle build tool.

Lexing and parsing is handled by code that is automatically generated from the language grammar using ANTLR. After parsing the source code, the compiler converts the parse tree to an abstract syntax tree (AST) and performs several AST traversals to generate symbol tables and perform type inference. A final AST traversal generates Java target code.

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 their inferred parameter/return types. Unit tests are emitted as JUnit test methods with the appropriate assertion.

After code generation, the generated Java source code must also be compiled with the Java compiler before it can be executed. The Less-Java compiler automates this process so that the user does not need to do it separately.

3 Results

3.1 Examples

The following is a simple example program in Less-Java:

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

main()
{
    printf("4 + 5 is %d", add(4,5))
}

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

Through a combination of implied boilerplate, simplified unit tests, and type inference, the Less-Java program is significantly shorter than equivalent Java code. Thanks to type inference, the add function doesn't need to be overloaded like it would be in Java. Instead, the function is considered generic, and a new copy of the target code is generated for each unique set of types (e.g., integers and doubles) at concrete function calls.

The following excerpt contains a more complex program, which is a computation related to the well-known Collatz conjecture [11]. This conjecture concerns a sequence where successive terms are obtained from previous terms

beginning with any positive integer. If the current term n is even, the next term is $n/2$. If the current term is odd, the next term is $3n + 1$. The Collatz conjecture posits that this sequence will always converge to 1. For our example, we wish to calculate the maximum sequence length within a (low, high) range of initial starting integers. The code demonstrates a range of language features, including functions, conditionals, loops, and unit tests. Additionally, it shows the conciseness and cleanness of the language.

```
// calculate the length of the Collatz sequence beginning at n
seq_len(n)
{
    len = 1
    while (n != 1) {
        if (n % 2 == 0) {         // even
            n = n / 2
        } else {                  // odd
            n = 3*n + 1
        }
        len = len + 1
    }
    return len
}

test seq_len(1) == 1
test seq_len(6) == 9

// find the maximum sequence length for starting values in the given range
max_3np1_seq(low, high)
{
    max = 0
    while (low <= high) {
        len = seq_len(low)
        if (len > max) {
            max = len             // new maximum
        }
        low = low + 1
    }
    return max
}

test max_3np1_seq(1,   10)   == 20
test max_3np1_seq(100, 200)  == 125
test max_3np1_seq(201, 210)  == 89
test max_3np1_seq(900, 1000) == 174
```

3.2 Preliminary Evaluation

We were unable to observe students using the language in a controlled study because of time constraints. However, we did conduct an informal experiment during a competitive programming club meeting. Students were tasked with solving a previously-approved list of problems in Less-Java (including the Collatz conjecture problem described above) without having had any prior exposure to the language. The problems were distributed as Less-Java files with some included unit tests so that the students could check themselves. A brief

demo program was posted on a projector for students to reference. The program contained many of the supported control structures, native functions, and syntax features of Less-Java. There was no formal data collection, but students were able to solve every problem with limited assistance from the language author and the club advisor. The students' solutions were reasonable in both length and complexity, and students seemed to appreciate the ease of testing.

4 Future Work

There are many avenues for future work. Some are mostly cosmetic, such as improving the error messages that the compiler produces, adding file I/O, and implementing IDE support.

More significantly, object-oriented programming in Less-Java is currently incomplete. While type inference is successful across assignments, it becomes unstable when objects are passed as function parameters. Overloading a function to handle the different parameters is insufficient for the object-oriented case; functions with many polymorphic parameters need to be emitted once for each combination of the parameters in the worst case. This cross-product property might generate unreasonably large compiled files, especially with deep inheritance hierarchies. One potential solution is to assign a list of interfaces to objects based on their methods, offloading the static analysis work to Java's interfaces. This would likely require a more extensive implementation of the Hindley-Milner type inference algorithm.

As mentioned in the previous section, an objective comparison study between Less-Java and some of the languages mentioned in the introduction would let us draw more significant conclusions in regards to language features and their impact on programming education.

Finally, the reference compiler for Less-Java has no optimization phase and the generated code hasn't been rigorously benchmarked against other languages. It should perform similarly to Java because we merely delegate most operations to the corresponding Java constructs, but a comprehensive performance benchmark 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.

5 Conclusion

Mainstream programming languages remain suboptimal for introductory computer science education. Many are verbose, unintuitive, confusing, or some

combination thereof. To address this problem, we have presented Less-Java, a new programming language along with a reference compiler. Less-Java limits the constructs available to the programmer to avoid confusion and complexity while still providing all of the tools necessary to teach an introductory programming course, including a simple and concise syntax, implicit but strong typing via type inference, and built-in unit testing. This project also serves as a basis for a wide range of future work.

References

[1] Less-java. `https://github.com/zamua/less-java`. Accessed: 2018-07-24.

[2] Snap. `https://snap.berkeley.edu`. Accessed: 2018-07-24.

[3] Alzahrani, N., Vahid, F., Edgcomb, A., Nguyen, K., and Lysecky, R. 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* (New York, NY, USA, 2018), SIGCSE '18, ACM, pp. 86–91.

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

[5] Milner, R. A theory of type polymorphism in programming. *Journal of computer and system sciences 17*, 3 (1978), 348–375.

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

[7] Parr, T., and Fisher, K. Ll (*): the foundation of the antlr parser generator. *ACM Sigplan Notices 46*, 6 (2011), 425–436.

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

[9] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. Scratch: Programming for all. *Commun. ACM 52*, 11 (Nov. 2009), 60–67.

[10] Somers, J. The coming software apocalypse. `https://www.theatlantic.com/technology/archive/2017/09/saving-the-world-from-code/540393/`, Sep 2017.

[11] Wikipedia contributors. Collatz conjecture — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Collatz_conjecture&oldid=829100536, 2018. [Online; accessed 20-March-2018].