Bricklayer: An Authentic Introduction to the Functional Programming Language SML

Victor Winter
Department of Computer Science
University of Nebraska at Omaha
Omaha, USA
vwinter@unomaha.edu

Functional programming languages are seen by many as instrumental to effectively utilizing the computational power of multi-core platforms. As a result, there is growing interest to introduce functional programming and functional thinking as early as possible within the computer science curriculum.

Bricklayer is an API, written in SML, that provides a set of abstractions for creating LEGO® artifacts which can be viewed using LEGO Digital Designer. The goal of *Bricklayer* is to create a problem space (i.e., a set of LEGO artifacts) that is accessible and engaging to programmers (especially novice programmers) while providing an authentic introduction to the functional programming language SML.

1 Motivation

Technological advances are giving rise to computational environments in which concurrent and parallel computation are rapidly gaining in prominence. For example, the Intel Xeon E7-8870 processor, released in April 2011, contains 10 cores and supports 20 threads. On October 2012, AMD released the FX-8300 processor that contains 8 cores and supports 8 threads. Such architectures present significant challenges to traditional "von Neumann trained" software developers: Specifically, how does one best write von Neumann-style programs capable of utilizing the computational resources of such multi-core (soon to be massively multi-core) platforms?

From a conceptual standpoint, there are two models of concurrency: (1) *shared state concurrency*, and (2) *message passing concurrency*. Languages like C, Java, and C++ embrace the notion of a *mutable state* and are thus well-suited for concurrency based on a shared state. In this setting, a mechanism such as a *mutex* or *synchronized method* is needed to prevent simultaneous modification of shared memory. Such mechanisms have the ability to "lock" a specific region of memory thereby enabling sequential modification of the shared program state. Key concerns for such computational models are *deadlock* and *starvation*. Furthermore, catastrophic problems can arise if a program in possession of a lock crashes.

In contrast, functional languages are based on a computational model well-suited to a form of concurrency based on message passing. Unlike imperative languages, whose computational models are abstractly isomorphic to von Neumann architectures, functional programming languages, such as Erlang and SML, trace their origins to mathematics. This difference has long been recognized. In his Turing Award lecture in 1978 [3], John Backus raised the question "Can Programming Be Liberated from the von Neumann Style?" In the world of multi-core programming, the time for such liberation is at hand.

In the context of multi-core computing, the key attribute of computations expressed as functional programs is that they are (essentially) *stateless*. As a result, a vacuous proof is all that is needed to verify that a stateless functional program avoids all problems arising from shared state (e.g., deadlock).

Submitted to:
TFPIE 2014

© Victor Winter
This work is licensed under the
Creative Commons Attribution License.

Statelessness (also referred to as *immutability*) as well as several other properties, such as referential transparency, make functional programs well-suited for distribution over multiple cores. Echoing such sentiment, Joe Armstrong has stated that, subject to minor caveats, "Your Erlang program should just run N times faster on an N core processor".[2]

In applications involving bulk-data running on multi-core environments, the importance of functions is perhaps most prominent. In fact, the need for functions is so central to this growing class of computations that Oracle has responded by incorporating functional concepts into Java. The most significant features of Java 8, the latest release of Java, are *lambda expressions* and *method references*. Such function values enjoy a pseudo-first class status, the most important of which being that they can be passed as parameters to other methods. This computational shift enables a profound change in how iterations over collections can be performed. Specifically, a collection API can now control the parallelization of iterations over the data it stores. This in turn, frees client code dealing with bulk-data from many concerns pertaining to the effective use of the multi-core platforms upon which it is expected to execute.

The suitability of functional programming and functional thinking to multi-core architectures is creating pressure to increase the footprint of functional programming within the educational system. Spearheading the effort to bring about this educational shift is CMU which, in 2011, began teaching functional programming (using SML) to its freshmen. What is perhaps more remarkable is that, at CMU, OO programming is eliminated from its introductory curriculum.

> *Object-oriented programming is eliminated entirely from the introductory curriculum, because it is both anti-modular and anti-parallel by its very nature, and hence unsuitable for a modern CS curriculum. [17]*

CMU did not arrive at this decision lightly [6] and is not alone in its assessment of the importance of functional programming languages. In an article written in 2008 [16], Michael Swaine writes "Knowledge of functional programming (FP) is on the verge of becoming a must-have skill" because it is ideally suited to "solving the multi-core problem". More recently an article was published in the Huffington Post [14] titled "Teach a Kid Functional Programming and You Feed Her for a Lifetime."

A number of universities teach freshmen courses on functional programming. This list includes (but is not limited to): Carnegie Mellon University, University of Edinburgh, University of Nebraska at Omaha, University of North Carolina – Chapel Hill, University of Oxford, Portland State University, Rice University, and Vassar College. Furthermore, many of these courses are introductory requiring minimal or no CS background.

2 Contribution

This paper introduces *Bricklayer*, an API written in SML that provides a set of abstractions for creating LEGO artifacts which can be viewed using LEGO Digital Designer. The goal of *Bricklayer* is to create a problem space (i.e., a set of LEGO artifacts) that is accessible and engaging to programmers (especially novice programmers) while providing an authentic introduction to the functional programming language SML. This paper also describes an approach and philosophy to teaching functional programming and discusses how *Bricklayer* aligns with this approach and philosophy.

The remainder of the paper is as follows: Section 3 discusses related work in developing environments and API's that facilitate teaching programming and computational thinking. Section 4 argues for the use of functional languages to teach first-time programmers and describes key characteristics of computations suitable for programming assignments. Section 5 gives an overview of the *Bricklayer* API and

argues that the LEGO problem space is well-suited for introductory programming assignments. Sections 6 through 9 describe the various SML structures which make up *Bricklayer*, and Section 10 concludes.

3 Related Work

In recent years, US student interest in pursuing computer and information sciences has experienced an alarming decline.

> *Over the past five years[2001-2005], the percentage of ACT-tested students who said they were interested in majoring in computer and information science has dropped steadily from 4.5 percent to 2.9 percent.* [1]

The 2012 ACT Profile Report indicates that planned education majors for students in the fields of computer science and mathematics (combined) is at 2 percent[1]. Juxtaposed to this, the demand for computer science in the workforce continues to grow as does the development and production of multi-core architectures.

Academia has responded to this imbalance by developing languages and programming environments where skills in programming and computational thinking can be developed in ways that are fun and engaging to a broader student population. A fundamental issue confronted in the design of such languages and environments is that of *authenticity*. Authenticity characterizes the degree to which a language realistically combines computational thinking with general-purpose programming. By this metric, syntax-free interfaces associated with languages such as Scratch [11] and Alice [8] are considered to focus more on computational thinking. In an alternate approach, a general-purpose programming language such as Java, Jython[2], or SML can be extended with an API for the purposes of bringing a problem domain within reach of a certain computational thinking skillset. For example, CodeSpells [9] teaches Java programming through first-person immersion in a 3D fantasy role-playing game. In CodeSpells, successful game play requires understanding and adaptation of spells, which are Java methods, belonging to a spellbook. Another example is the Media Computation API [10], developed by Guzdial, that enables non-CS majors to construct Jython programs which manipulate photos in a variety of ways. The approach taken by *Bricklayer* is similar to that of the Media Computation API and brings a world of LEGO® artifacts within reach of authentic SML programming.

4 On Teaching First-time Programmers

Which language is the most appropriate for first-time programmers has been the subject of considerable debate [4][13][12]. With respect to functional programming specifically, studies have been conducted investigating the pros and cons of teaching such languages to freshmen [7][5]. Though arguments have been made from both sides, from the perspective of language design, when viewed in their totality, OO languages such as Java have both a syntax and semantics that is considerably more complex than SML.

In Java, for example, significant complexity is introduced through computational behavior relating to: visibility, nested classes, anonymous classes, class initialization, subtyping, overloading, shadowing, overriding, generic types, static imports, break and continue statements, constructors, *super*, and *this*. In Java 8, lambda expressions and method references also introduce significant complexity. One additional

[1] The chart only displays whole numbers.

[2] Jython is Java-based implementation of Python that has the ability to use Java classes.

area of surprising complexity is Java's *resolution algorithm* which determines the declaration associated with a reference. In its full generality, Java's resolution algorithm is virtually incomprehensible. In support of this claim we offer as evidence that we have not yet discovered a Java IDE or refactoring tool that implements resolution correctly in all cases [15][19]. We have even discovered (and reported) a bug in the Java 1.6.0_26 compiler itself [19].

In spite of Java's overall complexity, its imperative core is relatively well-behaved, both syntactically and semantically. Developing programming facility within this core is the predominant focus of introductory programming based on Java. However, because of the assignment statement (e.g., *x = x + 1*) students are forced to incorporate reasoning about *state* and *time* in their initial mental model of computation.

> *Our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.*
>
> Edsger W. Dijkstra

In languages like C++, the notion of time is even more convoluted and expressions like "x++ + x" are literally undefined, even though they will compile and produce different results with different compilers – oftentimes without warning. Explaining "why this is so" to a first-time programmer borders on the impossible.

As a programming language, SML's simplicity (when compared to languages like Java) makes it an attractive candidate for an introductory course on programming. SML's (stateless) core consists of (1) primitive types, values, and operators, (2) conditional expressions, (3) *val* and *fun* declarations, and (4) let-blocks. The key challenge faced in functional programming lies in understanding the dynamics of recursion. Though intellectually cleaner than invariant-based reasoning about loops, recursion is based on inductive reasoning to which students often have had insufficient prior exposure. For this reason, it is beneficial to introduce recursion in a gradual fashion only after sufficient dexterity has been attained in non-recursive programming.

4.1 Fixed and Variable-length Computations

A basic understanding of any programming language, such as SML, begins with an an introduction to (1) primitive values and types (e.g., integer, Boolean, and string), (2) primitive operations (e.g., $<, =, >$), and (3) core programming constructs (e.g., expressions, declarations, conditions). These language constructs can be composed in a variety of ways to create simple programs (or program fragments) whose execution performs a single *computational step*. For example, an expression that multiplies two numbers is considered to perform a single computational step.

From such fundamentals one can begin the study of how to write a program whose execution will carry out a specific *computation sequence* comprised of one or more computational steps.

The study of computation sequences can be partitioned into the following categories.

1. *fixed-length* – These are computation sequences that do not contain iterators. As a result, fixed-length computations have a constant $\mathcal{O}(1)$ upper-bound on the number of computational steps they perform regardless of the input they are processing. For example, a function that converts

Fahrenheit to Celsius can be implemented using a fixed-length computation (e.g., the conversion from $32°F$ to $0°C$ takes just as many computational steps as the conversion $212°F$ to $100°C$).

2. *variable-length* – These are computation sequences containing iterators, which in the case of functional programming languages, is achieved via recursion. The number of computational steps in variable-length computations can vary according to their input. For example, a simple $\mathcal{O}(n^2)$ sorting function will, roughly speaking, perform $\mathcal{O}(10^2)$ computational steps when given a list of 10 elements as input.

4.2 Suitable Problems

Regardless of language and computation sequence category, introductory programming constrains, in a variety of ways, the kinds of programming problems considered to be *suitable*. Suitable problem statements must be concise and may not make use of advanced data representations or programming language constructs. As a result, teachers often look to mathematical domains to provide the venue for studying both fixed-length as well as variable-length computation. Examples of suitable mathematical problems include: (1) do three integers form a Pythagorean triple, (2) conversions from base 10 to binary, octal or hexadecimal, (3) binary search in the form of 20 questions, (4) tests for primeness (e.g., Sieve of Eratosthenes), (5) perfect squares, (6) magic squares, and (7) termination experiments involving the Collatz conjecture. One concern is that such problems oftentimes do not sufficiently appeal to a broad student population.

4.3 Problems, Examples, and Study

The construction of implementations that solve problems is a key form of feedback employed in programming classes. When viewed along this dimension, a course can be seen as a sequence of *programming assignments* (aka problems) separated by *study* (the development of understanding and computational thinking skills needed to solve a targeted class of problems).

At the heart of study lie germane and compelling *examples* highlighting certain forms of computational thinking. In this context we define an *example* to be a triple consisting of (1) a problem statement, (2) a problem solution, and (3) an instantiated articulation of various forms of computational thinking that can be used to reach the problem solution from the problem statement. A *problem* then can be seen as simply the prefix (i.e., the problem statement) of an example.

In order to be *effective*, a example-problem sequence should have the following attributes:

1. examples and problems are suitably coupled,
2. problems cover a targeted set of learning goals,
3. the change in difficulty between two related elements in the sequence falls within an appropriate tolerance, being neither too difficult nor too easy, and
4. problems and examples are engaging to students.

To achieve this, one needs a domain that is *example rich* and *problem dense*. This enables the construction of effective example-problem sequences that are both accurate and precise with respect to a given set of learning objectives.

When teaching functional programming, a key challenge is the transition from non-recursive to recursive examples and problems.

5 *Bricklayer*

Bricklayer [18] is an API, written in SML, that is being developed at the University of Nebraska at Omaha. *Bricklayer* provides a basic set of abstractions for creating LEGO® artifacts which can be viewed using LEGO Digital Designer (LDD). The rational for *Bricklayer* is premised on the assumption that a LEGO mindset is conducive to the study of programming for the following reasons:

1. *engaging* – Generally speaking, LEGO has a universal appeal. People like LEGOs – both students and their parents.
2. *determined* – Many students have had prior exposure to the construction of physical LEGO artifacts. For example, it is not uncommon for elementary school students to have assembled LEGO artifacts consisting of hundreds (even thousands) of pieces. Thus, the association of extensive instruction sequences with LEGOs has already been established. This association fosters patience and endurance with respect to LEGO-related activities. Such a mindset is beneficial to (*Bricklayer*) programming.
3. *concrete* – A LEGO artifact inhabits a discrete space having a physical manifestation. This provides an environment where students can, in many cases, "play with" and develop a prototype-based physical understanding of a problem before expressing its solution in code.

In the sections that follow, we hope to convince the reader that the artifacts that can be created using *Bricklayer* belong to a domain that is *example rich* and *problem dense* (as discussed in Section 4.3). Users interact with *Bricklayer* through four SML structures named: (I) Predicate, (II) BrickFunction, (III) BasicNavigation, and (IV) AdvancedNavigation. These structures have been ordered according to their computational sophistication and expressive power. A summary of the knowledge of SML that is prerequisite for the *Bricklayer* structures is shown in Table 1.

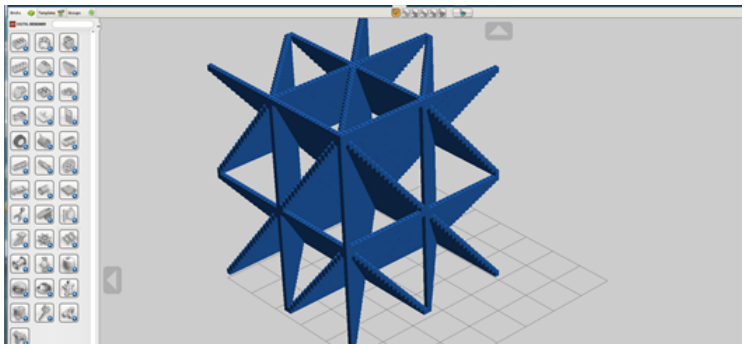
Note that recursion is first introduced in the BasicNavigation structure (III). In Table 1, we designate a function declaration to be *simple-recursive* if it implements the behavior of a loop. In this paper, we will refer to such recursive functions as *simple-recursive loops*.

Examples of the kinds of structures that can be created using the *Bricklayer* structures and displayed using LDD is shown in Figure 1.

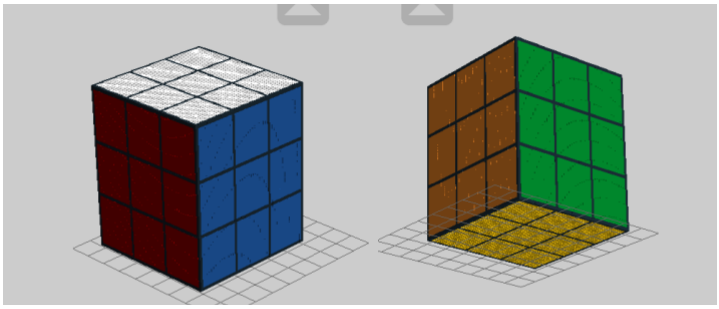
5.1 Separating Computational Concerns

Bricklayer provides some very simple ways to specify *virtual spaces*, which are 3D spaces whose integer coordinates range from $(0,0,0)$ to (x,y,z) where $x,y,z > 0$. In addition, *Bricklayer* provides a number of functions for interacting with virtual spaces. Most notably, *Bricklayer* provides a variety of simple functions for *(generically) traversing* a virtual space (user-defined traversals are also possible).

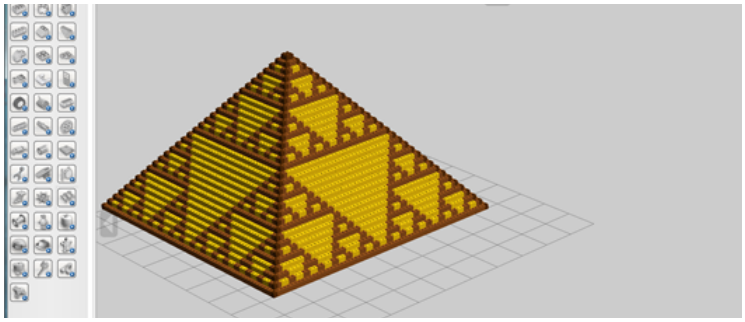
As an aside, it is worth noting that the importance of generic traversal and related computational ideas have been recognized in a variety of areas. In the field of strategic programming, generic traversal enables the separation of conditional rewrite rules from the algorithmic details surrounding their application. In declarative programming and rewriting systems the technical details surrounding search and rule application are handled by the computational model underlying the language. *Bricklayer* introduces such *separation of computational concerns* into the SML core. Through *Bricklayer* traversals is it possible to build a wide variety of three dimensional LEGO artifacts without having to confront the complexity of recursion.



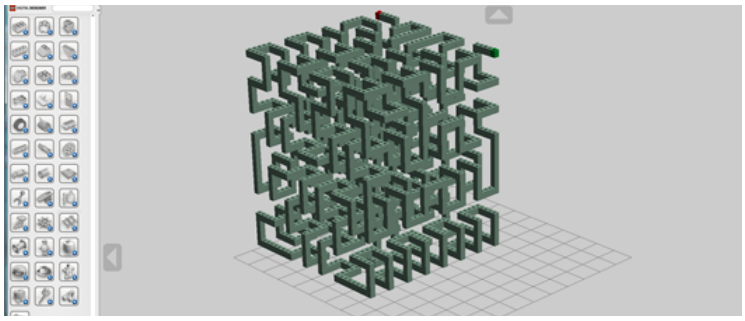
(a) Predicate



(b) BrickFunction



(c) BasicNavigation



(d) AdvancedNavigation

Figure 1: Artifacts created using Bricklayer and displayed using LDD

	I	II	III	IV
primitive types				
integer	√	√	√	√
Boolean	√	√	√	√
string			√	√
...				
variables	√	√	√	√
operators				
arithmetic	√	√	√	√
comparison	√	√	√	√
logical	√	√	√	√
expressions				
arithmetic	√	√	√	√
logical	√	√	√	√
let-blocks	√	√	√	√
conditional expressions		√	√	√
sequence			√	√
output (i.e., print)			√	√
declarations				
val-dec	√	√	√	√
non-recursive fun-dec	√	√	√	√
simple-recursive fun-dec			√	√
recursive fun-dec				√

Table 1: A partial overview of the intersection between SML concepts and *Bricklayer* modules.

6 Predicates

The primary function exported by the Predicate structure is: *show*. This function when called with

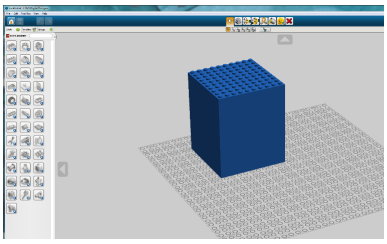
1. an integer n denoting the side of a cube,
2. a function (i.e., a predicate) from (x,y,z) coordinates to Boolean values, and
3. a brick value

will

1. create a virtual cube within Bricklayer whose coordinates range over (0,0,0),...,(n-1,n-1,n-1),
2. traverse the virtual cube's coordinate space and apply the given function to each coordinate, placing the given brick value at every coordinate for which the predicate evaluates to true,
3. output the resulting virtual structure to an lxfml format recognized by LDD, and
4. perform a system call loading LDD with the lxfml file created.

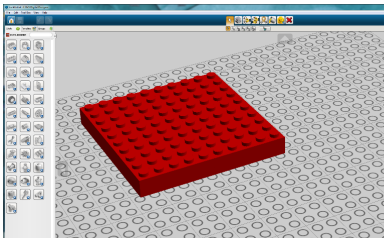
Example: When executed, the following SML code will create and display a $10 \times 10 \times 10$ cube consisting of blue 1-bit LEGO bricks.

```
fun cube(x,y,z) = true;

Predicate.show(10,Pieces.BLUE,cube);
```



Example: When executed, the following SML code will create and display a 10×10 square in the X-Z plane consisting of red 1-bit LEGO bricks.

```
fun square(x,y,z) = y = 0;

Predicate.show(10,Pieces.RED,square);
```



From a technical standpoint, the class of computations that can be studied using the Predicate structure center on expressions involving Boolean and comparison operators. Additional programming constructs, such as helper functions and local variables, may be used to structure code and increase readability. This set computations, which is a subset of the set of *fixed-width computations*, is immense and has incredible expressive power. From a theoretical perspective, this is to be expected.

6.1 Providing Feedback through *Bricklayer* Comparison Functions

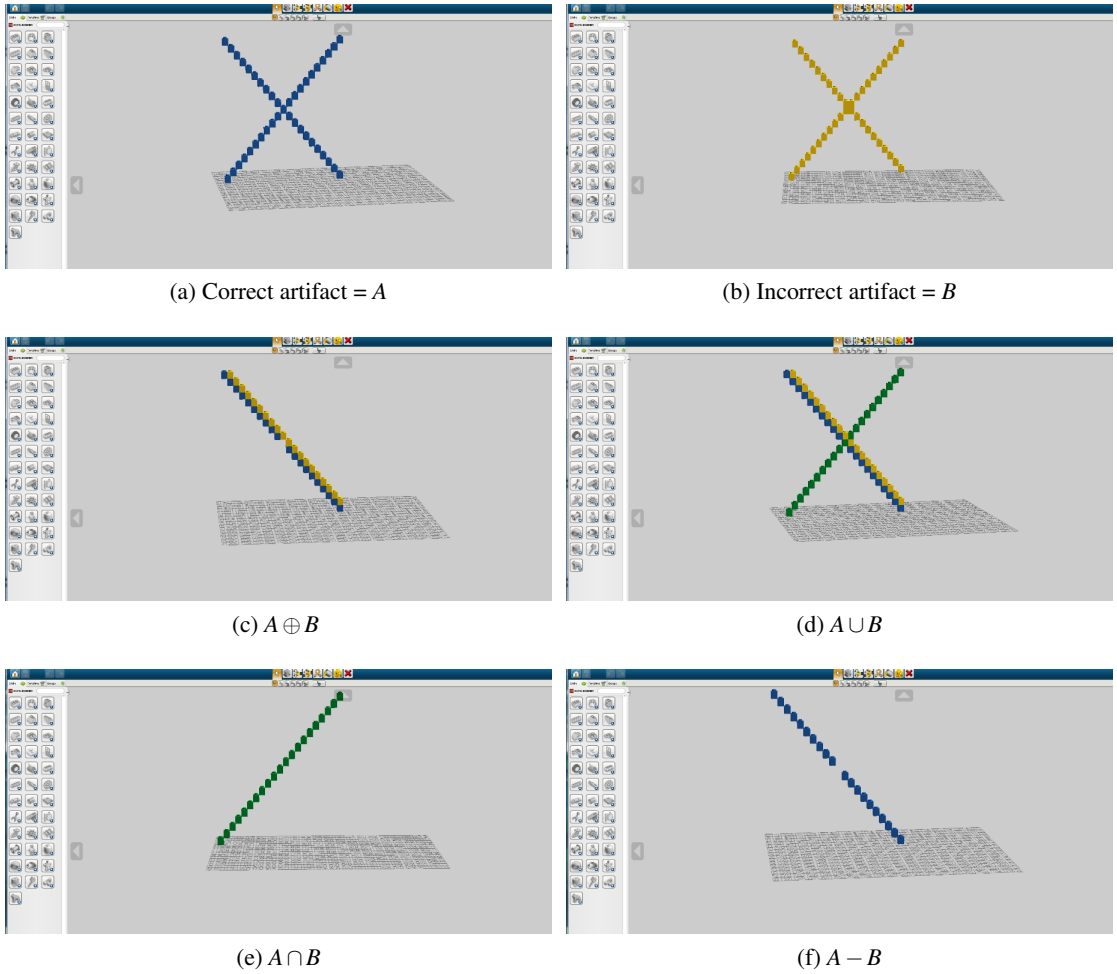
In addition to *show*, the Predicate structure exports the functions *xor*, *union*, *intersection*, and *difference*. These functions enable users to examine differences between two LEGO artifacts (e.g., an artifact in a problem description and the artifact produced by their code). In particular, a student can use these functions to compare (and contrast) their solution to a problem with the instructor's solution.

Figure 2a shows the LEGO artifact that is the solution to a programming assignment. Figure 2b shows the LEGO artifact created by the student's program. In this case, the student's solution contains an off-by-one error. In practice, such errors are common and their discovery without the use of *Bricklayer* comparison functions can involved detailed counting (e.g., counting the number of LEGO bricks in the base of a triangle). Figures 2c through 2f show the various LEGO artifacts resulting from comparing the correct artifact with the student's artifact.

Bricklayer comparison functions provide users with control over the bricks used to create the resulting artifact. In Figure 2, (1) blue bricks are used to denote bricks belonging to the instructor solution, (2) yellow bricks are used to denote bricks belonging to the student's solution, and (3) green bricks are used to denote bricks that the instructor's solution and the student's solution have in common.

6.2 Relationships between Triangles and Loops

The construction of triangles and pyramids, in their various orientations and forms involve computational thinking similar to that underlying nested for-loops. Consider for example, the predicate $p(x,y,z) = x +$



(a) Correct artifact = A (b) Incorrect artifact = B

(c) $A \oplus B$ (d) $A \cup B$

(e) $A \cap B$ (f) $A - B$

Figure 2: Comparing and contrasting student solutions with instructor solutions.

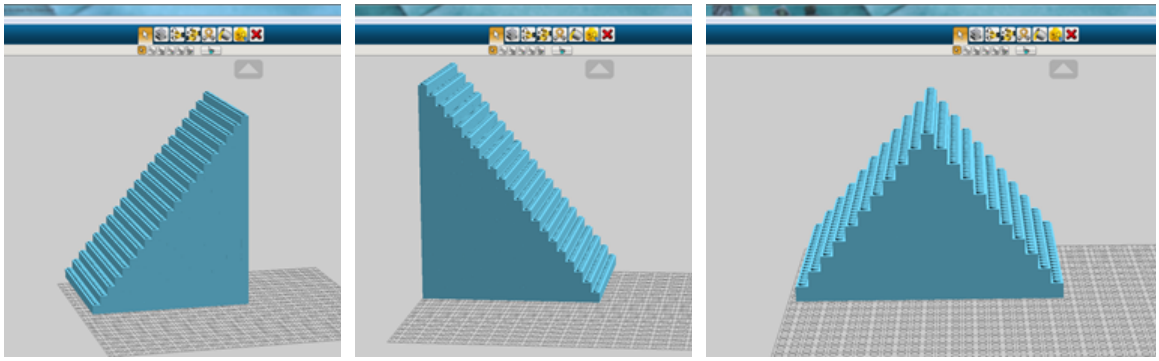
$y < 20$ whose structure is shown in Figure 3b. The computational thinking underlying this construction shares similarities to the computational thinking used to create imperative loops of the form:

```
for i = 0 to 19
    for j = 0 to 19 - i
```

The looping pattern shown above can be used to implement the bubblesort algorithm. Furthermore, counting the number of bricks placed in the triangle in Figure 3b requires computational thinking similar to that which is required to perform a big-$\mathcal{O}$ analysis of the running time of algorithms such as bubblesort.

7 Brick Functions

The BrickFunction structure is similar to the Predicate structure in that its primary export is the function: *show*. The behavior of *BrickFunction.show* is similar to *Predicate.show*. The only difference being that *BrickFunction.show* accepts only two inputs: (1) an integer n denoting the side of a cube, and (2) a total function f from (x, y, z) coordinates to brick values.



(a) x >= y (b) x+y < 20 (c) $a \cap b$

Figure 3: Virtual space: $(0,0,0)\ldots(19,19,19)$

A limitation of the *Predicate.show* is that objects are created using a single type of LEGO brick (e.g., blue, green, etc.). *BrickFunction.show* removes this limitation by processing functions whose return values are LEGO bricks (instead of simple Boolean values). As a result, LEGO artifacts can now be created using multiple brick types (i.e., different colors and shapes).

When using the BrickFunction structure students encounter a compelling problem domain requiring the incorporation of conditional expressions into their programming skill sets. While *conditional expressions* are not essential for the creation of predicate-based LEGO artifacts, conditional expressions are central to the construction of expressions whose evaluation results in a LEGO brick (e.g., brick functions). Conditional expressions provide users with control over what kind of brick is placed in a given cell in the virtual space. However, with this ability comes the responsibility for designating which cells are to remain empty. Bricklayer provides an empty brick for this purpose.

7.1 Periodic Functions and Equivalence Classes

Periodic functions (such as modulus and sine) can be used to create repeating patterns within LEGO artifacts. Periodic functions can be used in all *Bricklayer* levels. For example, the artifact in Figure 1a results from a simple predicate that uses the *mod* operator. It should be noted however that the effects of periodic functions are more compelling when dealing with multi-colored artifacts. The Rubik's cube[3] shown in Figure 1b is created by composing the *mod* operator with conditional coloring. The artifact in figure 4 is created using the sine wave function and mapping different heights of the curve to various shades of blue.

Operations such as *div* as well as "within range" comparisons can be used to group coordinates into equivalence classes. Mappings can then be created from equivalence classes to colors. Figure 5 shows how *div* and *mod* operations can be combined to create a checkerboard where each square of the board consists of 25 LEGO bricks. The example contains a nested conditional expression where the outer conditional serves to define the "domain of discourse" for the inner conditional.

[3]Special thanks to Jared Knust and Colin Kramer for providing this example.

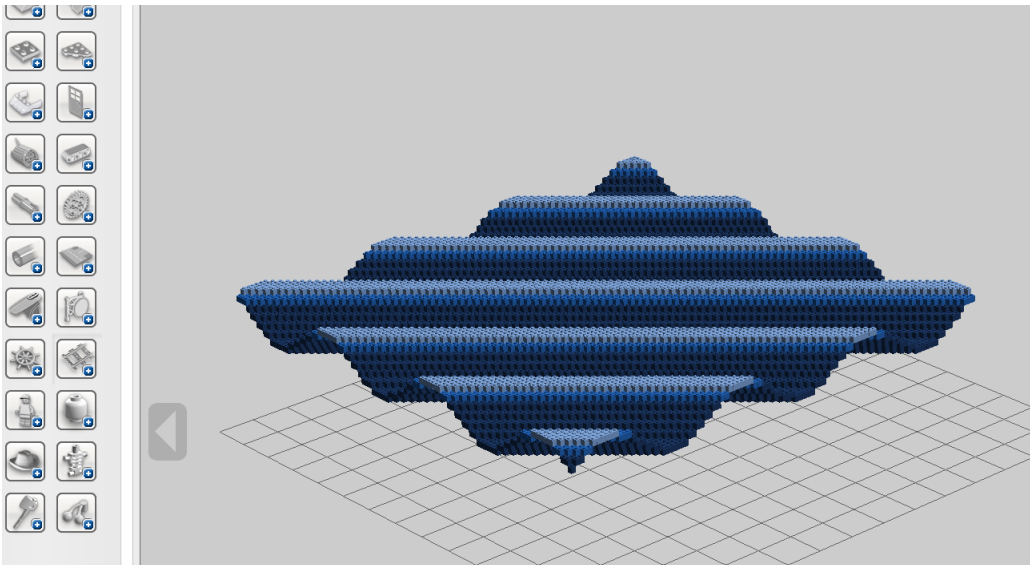


Figure 4: Sine waves with shading.

8 Basic Navigation

It is in the BasicNavigation module where simple recursion is first introduced. The BasicNavigation structure provides a number of traversal functions such as: (1) a function that performs a traversal within a given bounding box, and (2) a set of functions that traverse a two dimensional space for a fixed value of the third dimension. In addition, *access* and *update* functions are provided for cell-level interaction with the virtual space. A function is also provided that draws a "smooth" line between two given points.

Figure 6a shows an artifact that can be created using a *simple-recursive loop* (see Section 5) that shrinks the bounding box (e.g., $(lo, lo, lo) \ldots (hi, hi, hi)$) within which a wire-frame cube is to be created. Figure 6b is a composite artifact in which a simple-recursive loop is used to create the green portion of the Christmas tree which consists of a sequence of increasingly smaller green cones. The remaining portions of the artifact are produced in a non-recursive fashion (e.g., a random function is used to generate falling snow).

The access function provided by the BasicNavigation structure enables the contents of cells in the virtual space to be inspected. This opens the door to computations where the order in which cells are visited (during a traversal) is important. The artifact in Figure 1c, which is based on the Sierpinski gasket, is an example of a computation in which order of traversal is important. There were only two simple-recursive loops used in the construction of this artifact.

9 Advanced Navigation

In addition to all the functionality described thus far, the AdvancedNavigation structure enables users to explicitly create and manipulate virtual space values directly. Also provided are "turtle graphics" capabilities which enable fairly direct implementations of fractals which are specified in terms of Lindenmayer systems (L-systems). The Hilbert cube shown in Figure 1d was created using the capabilities provided by the AdvancedNavigation structure. Depending on the target audience, the AdvancedNavigation structure

```
fun bigCheckerboard boardSize squareSize =
  let
    (* assumes there exists an integer c such that boardSize = c * squareSize *)

    fun equivClass v = (v div squareSize) mod 2

    fun brickFunction(x,y,z) =
      let
        val black = equivClass x = equivClass z
      in
        if y = 0 then
           if black then Pieces.BLACK
           else Pieces.ORANGE
        else Pieces.EMPTY
      end
  in
    BrickFunction.show(boardSize, brickFunction)
  end;
```

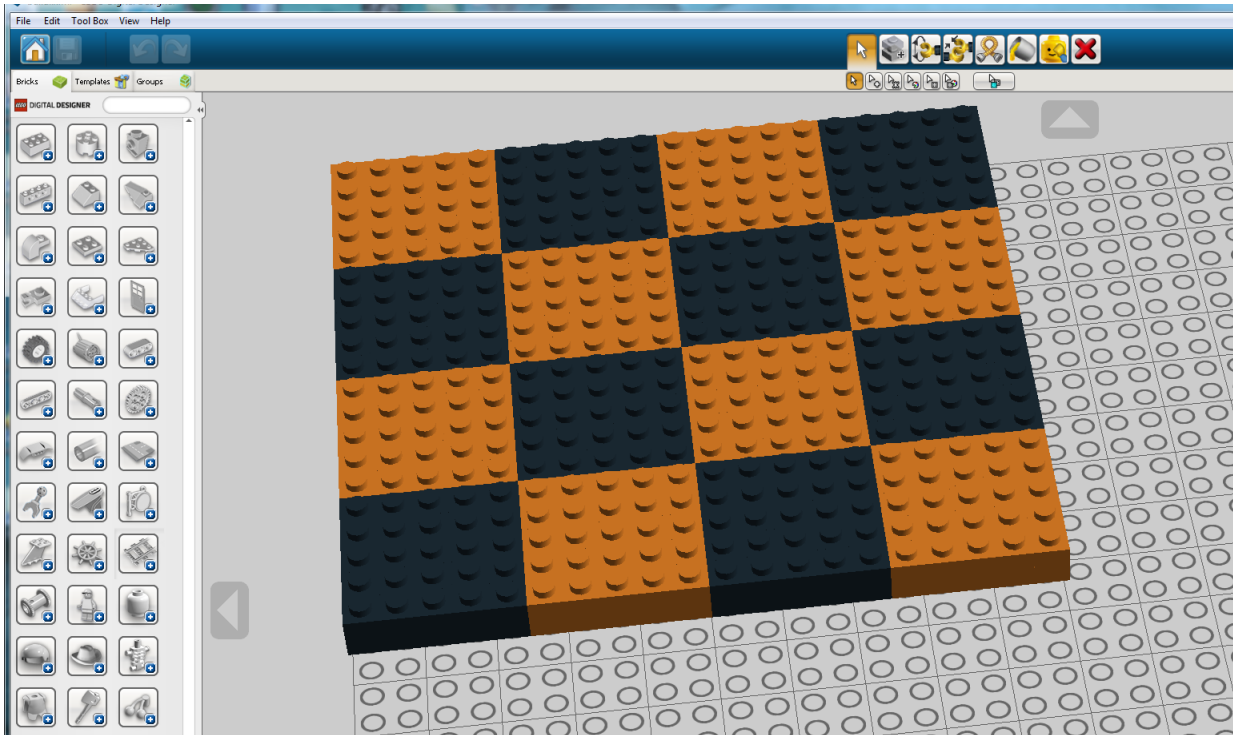
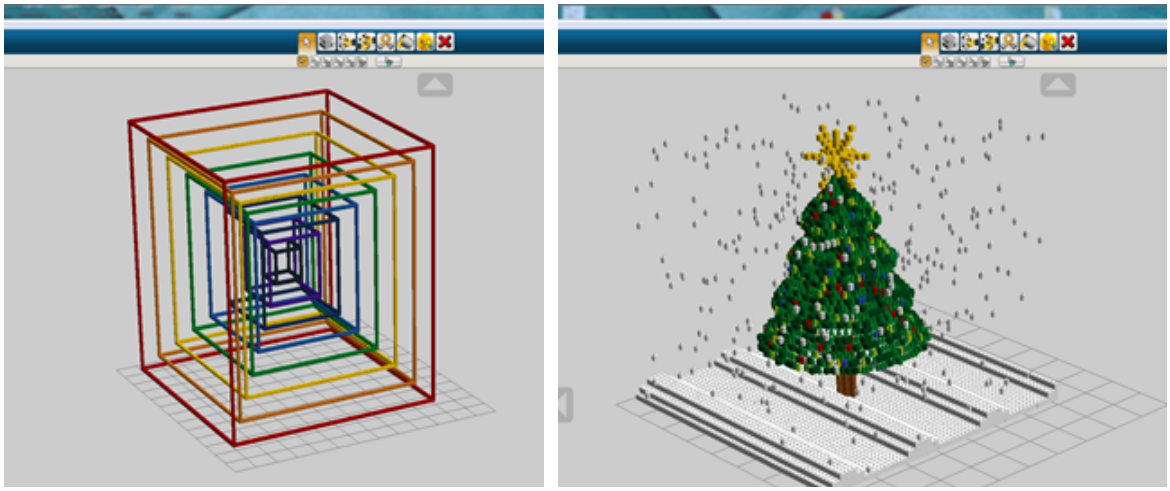


Figure 5: Using equivalence classes to create a checkerboard having big squares.

may (probably should) be omitted from introductory course content.



(a) Nested Cubes (b) Season's Greetings

Figure 6: Applications of simple-recursive functions.

10 Conclusion

The *Bricklayer* API connects SML programming with a rich domain of LEGO® artifacts. In *Bricklayer*, user developed predicates and brick functions characterize a class of fixed-width computations whose combination with *Bricklayer*-provided generic traversals enable the creation of an incredibly broad range of artifacts. Within this realm of fixed-width computations, expressed as *Bricklayer* predicates and brick functions, problems whose solutions requiring sophisticated computational thinking abound. Concepts such as (de)composition, abstraction, generalization, and pattern recognition are inherent to the construction of LEGO artifacts. Furthermore, a number of concepts have direct correspondences with topics taught in discrete math courses. For example, (1) the conjunction of predicates corresponds to set *union*, (2) the outer conditions of nested conditionals can be used to restrict computational thinking to a particular domain of discourse, (3) computations involving division and modulus can be used to create repeating patterns as well as equivalence classes, and (4) triangles – in their many shapes and orientations – have strong conceptual analogies to nested loops. As a result, extensive meaningful study is possible within the realm of fixed-width computation.

In *Bricklayer*, recursive thinking is introduced gradually in the BasicNavigation structure. User-defined recursive functions whose behavior mirrors that of simple for-loops (e.g., a function that iterates over the z dimension of a virtual cube) can be composed with a variety of *Bricklayer* traversal functions (e.g., traverse the XY plane for a given value of z and apply a brick function to every point encountered) to create a broad range of artifacts. Simple-recursive loops can then be composed to form nested loop functions gradually increasing the recursive complexity of an implementation. The end point is reached when *Bricklayer* traversal functions are no longer used to create structures. After such a point is reached, study can continue using the functionality provided by the AdvancedNavigation structure. Key features of the AdvancedNavigation structure include: (1) 3D turtle graphics, and (2) the treatment of the virtual space as a value. Using the functionality provided by the AdvancedNavigation structure it is possible to implement (1) L-systems, and (2) computational models such as cellular automata and even Turing machines.

References

[1] ACT. Developing the STEM Pipeline, 2006.

[2] J. Armstrong. What's all this fuss about Erlang? In *The Pragmatic Bookshelf*, 2008.

[3] J. Backus. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. *Commun. ACM*, 21(8):613–641, Aug. 1978.

[4] J. D. Blake. Language Considerations in the First Year CS Curriculum. *J. Comput. Sci. Coll.*, 26(6):124–129, June 2011.

[5] S. A. Bloch. Scheme and Java in the First Year. *J. Comput. Sci. Coll.*, 15(5):157–165, Apr. 2000.

[6] R. E. Bryant, K. Sutner, and M. J. Stehlik. Introductory Computer Science Education at Carnegie Mellon University: A Dean's Perspective. Technical Report CMU-CS-10-140, School of Computer Science, Carnegie Mellon University, 2010.

[7] M. M. T. Chakravarty and G. Keller. The Risks and Benefits of Teaching Purely Functional Programming in First Year. *J. Funct. Program.*, 14(1):113–123, jan 2004.

[8] W. Dann, S. Cooper, and R. Pausch. *Learning to Program with ALICE*. Pearson Education, 501 Boylston Street, Suite 900, Boston Massachusetts 02116, third edition, 2012.

[9] S. Esper, S. R. Foster, and W. G. Griswold. CodeSpells: Embodying the Metaphor of Wizardry for Programming. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 249–254, New York, NY, USA, 2013. ACM.

[10] M. Guzdial. A Media Computation Course for Non-majors. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '03, pages 104–108, New York, NY, USA, 2003. ACM.

[11] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch Programming Language and Environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, Nov. 2010.

[12] L. Mannila and M. de Raadt. An Objective Comparison of Languages for Teaching Introductory Programming. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, Baltic Sea '06, pages 32–37, New York, NY, USA, 2006. ACM.

[13] L. McIver and D. Conway. Seven Deadly Sins of Introductory Programming Language Design. In *Proceedings of the 1996 International Conference on Software Engineering: Education and Practice (SE:EP '96)*, SEEP '96, pages 309–, Washington, DC, USA, 1996. IEEE Computer Society.

[14] J. Pavley. Teach a Kid Functional Programming and You Feed Her for a Lifetime. In *The Huffington Post*, July 2013.

[15] J. T. Perry, V. Winter, H. Siy, S. Srinivasan, B. D. Farkas, and J. A. McCoy. The Difficulties of Type Resolution Algorithms. Technical Report SAND2010-8745, Sandia National Laboratories, December 2010.

[16] M. Swaine. It's Time to Get Good a Functional Programming. In *Dr Dobbs*, 2008.

[17] E. Type. Teaching FP to Freshmen. `http://existentialtype.wordpress.com/2011/03/15/teaching-fp-to-freshmen/`. Accessed: 2013-12-08.

[18] V. Winter. `http://faculty.ist.unomaha.edu/winter/Bricklayer/index.html`.

[19] V. Winter, C. Reinke, and J. Guerrero. Using Program Transformation, Annotation, and Reflection to Certify a Java Type Resolution Function. In *Proceedings of the* 15^{th} *IEEE International Symposium on High Assurance Systems Engineering (HASE)*, 2014.