Do Values Grow on Trees?
Expression Integrity in Functional Programming

Guillaume Marceau
WPI
100 Institute Road
Worcester, MA, USA
+1 (508) 831-5357
gmarceau@wpi.edu

Kathi Fisler
WPI
100 Institute Road
Worcester, MA, USA
+1 (508) 831-5357
kfisler@cs.wpi.edu

Shriram Krishnamurthi
Brown University
115 Waterman St
Providence, RI, USA
+1 (401) 863-7600
sk@cs.brown.edu

ABSTRACT
We posit that functional programmers employ a notion called expression integrity to understand programs. We attempt to study the extent to which both novices and experts use this notion as they program, discuss the difficulties that arise in measuring this, and offer some observational findings.

Categories and Subject Descriptors
D.1.1 [**Programming Techniques**]: Applicative Programming; H.5.2 [**User Interfaces**]: Evaluation/methodology

General Terms
Design, Human Factors, Languages.

Keywords
Programming for novices. Structured editing.

1. Programming with Values
Some people advocate the use of functional programming in computer science education, especially at the introductory level. Others oppose it on the grounds that it is not merely outside the mainstream but may even be unnatural (perhaps suggesting the latter as an explanation for the former). Unfortunately, this long debate has seen more heat than light. We believe curricula would be better served by rigorous studies that examine the purported advantages and weaknesses claimed by each side.

First, let us review the basic structure of functional programs. The central idea is that programs exist to consume and produce *values*, akin to functions in algebra. Most introductory books completely eschew side-effects in the student's programs (even though they can be described with some effort even in traditionally functional languages). As a result, students do not see *statements*, only *expressions*. A program is simply a series of definitions, each of which has an expression body. Each expression may, recursively, have many nested sub-expressions. The actual computation is triggered by one or more expressions presented either in the program source or in an interactive evaluator (sometimes colloquially called an "interpreter", even though the underlying implementation may employ a compiler, JIT, or other strategy).

ICER'11, August 8–9, 2011, Providence, RI, USA.

The functional style raises many questions (as does its more traditional counterparts). For instance, one might ask how students relate to the decomposition of computation by statements versus expressions; how much the problems caused for program reasoning by side-effects are offset by benefits; at what depth of nesting students start to have difficulties; and so on. A broader set of issues is whether functional programming is indeed "natural" or not. If it is not, then perhaps it is unsuitable in education (though we must also then ask the same question about algebra itself, and understand how the answers relate). If it is, then perhaps some of the complaints about it stem more from instructor prejudice than from student behavior and perception.

To investigate such questions, we feel it is critical to first ask more foundational ones about students' ability to even relate to how functional programs compute. This paper reports on a first such investigation. From extensive discussions with experienced functional programmers, we find (anecdotally) that when reading, reviewing, and editing programs they *understand programs as trees of expressions, not as a sequence of characters*. We use the term **expression integrity** to capture this notion of understanding programs, and examine it in more detail. This concept is especially important in functional programs because everything other than a definition is an expression, even the control operations. We suspect, but don't study here, that this concept also applies to imperative programs that have shallow expressions but do have statements that can nest several levels (e.g., an assignment inside a conditional inside a loop inside a function).

Observe that "understand" has two senses: syntactic and semantic. Syntax is what users write (and hence it is what they have primary, direct control over), while semantics is what happens when the program runs (and hence reflects whether or not it met the user's goal). Computer scientists have long understood the tension between these two aspects of a language, and indeed movements such as *structured programming* grew precisely out of a direct desire to reduce the gap between the two. Thus, our investigation of functional programming should employ both syntactic and semantic angles.

This paper is about questions more than answers. It combines position (the importance of expression integrity) and discussion (how we might measure it).

2. Syntactic Studies
Expression integrity is presented in terms of how programmers understand programs. Of course, we cannot directly observe "understanding"; instead we must operationalize it. At a syntactic level, we postulate that there are certain behaviors we expect from programmers employing expression integrity, such as:

- They finish working with one expression before switching to another (for nested expressions, they may provide the

highest-level syntactic structure first, then fill in the sub-expressions, or they may complete sub-expressions in depth-first order). For languages with parenthetical syntax, they also maintain (relatively) balanced parentheses as they edit programs.

- They move and copy whole expressions.

These statements apply both to writing new code and to modifying existing code. For example, we would expect an expression-aware programmer to modify existing code by making edits as needed to a single (possibly nested) expression and restoring the surrounding expression structure (including parentheses) before moving on to another expression. Additionally, when modifying code, we would expect that

- They prefer reusing expressions to typing new code, even if the number of keystrokes is similar in each case.

Until a programmer has developed a sense of expression integrity, we would expect them to prefer edit paths with fewer keystrokes, and to apply edits without regard to the program's tree structure. (We observe in passing that these criteria are not limited to functional programs. A similar notion of integrity surely applies to statements also, though the rules are more complex.)

In other words, even when their editor permits character-wise editing (as most modern editors do), programmers with a sense of expression integrity conceptualize the syntax in terms of expression trees, and sometimes even avail of editor operations that atomically handle entire expressions at a time. These expression-oriented operations effectively endow programmers with the benefits of *structured* editing even in an unstructured editor, thereby avoiding the notorious inflexibility of fully-structured editors [1] without giving up their benefits.

Naturally, these are just claims (though based heavily on experience and anecdote). We set out to study these ideas in two populations: rank beginners, and experts. We first discuss our setup (Section 2.2), and our results for beginners (Section 2.3); then we step back and ask how our findings compare against expert behavior (Section 2.4). Before we get into these studies, however, it is worth examining the structure of expressions, since this affects what we can measure.

2.1 The Structure of Expressions

Observe that in a handful of languages (unlike Java, C, etc., where a program either parses or doesn't) there are multiple levels of "expression-ness". In XML, for instance, the levels are called *well-formedness* and *validity*. Well-formedness simply means that the basic rules of XML (pointy brackets, appropriate quotation of certain characters) are being followed, and that the opening and closing tags match and nest properly. However, a particular XML language will have additional rules. For instance, in XHTML (a version of HTML built atop XML), only certain tags are allowed, only some tags can nest within others, only certain attributes are legal, and so forth. This latter level of conformance is called validity. Thus all valid documents are well-formed, but not all well-formed documents are valid. Krishnamurthi [2] refers to languages with such a two-level structure as *bicameral*, since they mimic the bicameral legislatures of many countries.

Another language family famous for bicameral rules is Lisp. Well-formed terms (known as *s-expressions*) meet tokenization rules and have balanced parentheses, while valid terms are those that actually parse. In the Racket dialect, for instance, the term `(+ 1 2)` is both well-formed and valid, but `(lambda x)` is well-formed but not valid: it survives the "reader" (which converts a token stream into an s-expression) but fails to parse.

Our studies are conducted with students learning Racket. This gives us freedom to study programs at either level. While our tools support both, we chose to study well-formedness (which largely translates to whether programs are properly parenthesized) for two reasons:

1. It more closely corresponds to the level at which we believe *beginning* students think: they very quickly understand that programs must be properly parenthesized even though they do not understand the finer rules of grammar. Put differently, to beginners, balancing parentheses probably *is* what they imagine to be the primary form of validity.
2. When we study how student performance evolves over the course of the semester, it is valuable to quantify their progress. This requires that we can measure the degree of mis-parsedness of a program. This would presumably be based on the edit-distance to a properly parsed program, which is extremely sensitive to the editing operations available, and thus may be difficult for even experts to agree upon. In contrast, the degree of parenthetical imbalance (which, too, must deal with the fact that Racket programs have both parentheses and brackets) is a much more objective metric.

It is important to observe that, because every valid program is well-formed, a program that is not even well-formed is certainly not valid. Thus, any studies based on well-formedness would yield a lower-bound on what we would find by studying validity.

Of course, we must be careful when interpreting measurements. It is sometimes impossible to make edits without temporarily altering the validity—or even well-formedness—of a program. This can happen when fixing a typographical error. There are also edits whose entire point is to alter the way a program parses; in the process, it may be very difficult (or even impossible, given the structure of the language!) to make sure every intermediate state parses properly. We conjecture that programmers can easily tolerate such mal-formed intermediate programs provided they are not distracted during edits. Just as humans easily cope with poor spelling and grammar, a small amount of mis-parsedness seems entirely reasonable and perhaps cognitively preferable than trying to preserve parsability at all times.

More broadly, because most other programming languages do not have a bicameral structure, to generalize our results we must eventually examine validity also. Naturally, we should also study languages without a bicameral structure, to avoid confounding factors introduced by "lots of irritating, silly parentheses".

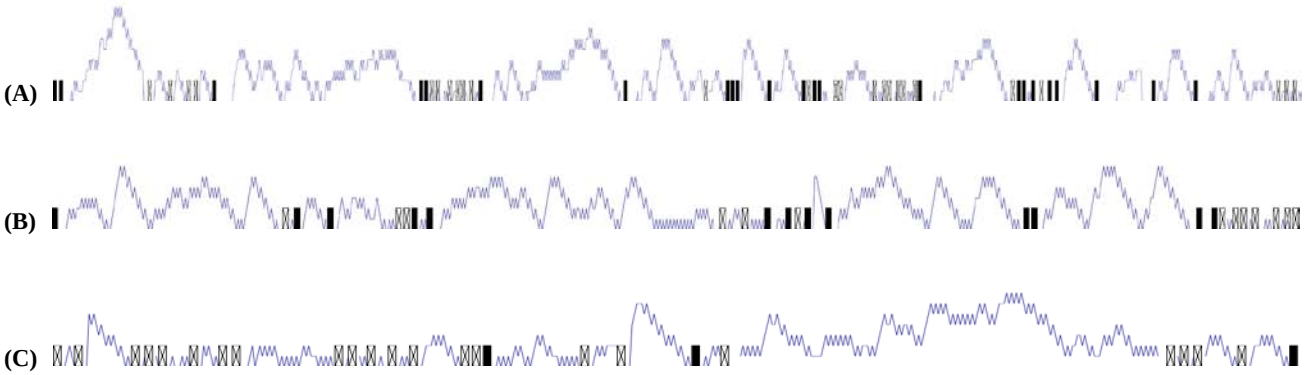


Figure 1. Different behaviors towards the maintenance of well-formedness and validity. The *y*-axes shows the amount of imbalance in the third lab (50 minutes) for an A-student, a B-student, and a C-student, respectively (same scale for all three). The highest peak is 9. Filled rectangles represent successful compilation attempts; x-ed rectangles are compilations that produced an error message.

2.2 Experimental Setup

We conducted our experiments in DrRacket, a modern (but intentionally spartan) interactive development environment. Three characteristics of DrRacket are noteworthy in this paper:

1. DrRacket runs identically (other than default key bindings) and as a native application on Windows, Mac, and Unix. Thus users can apply all the edit operations that they are comfortable with for the platform.
2. Whenever a cursor is placed (by mouse or keyboard) over a parenthesis, the environment highlights the entire parenthetical expression that begins or ends at that parenthesis. (If the parenthesis is mis-matched or dangling, DrRacket instead colors it pink.) This highlighting happens automatically without the need for a mode setting, as is necessary in some other editors.
3. Users can enable keystrokes for manipulating entire s-expressions, imitating those built into Emacs. Most of the advanced programmers described in this paper know these keystrokes, but none of the beginners were shown them (they are not easy to find without reading the manual, so most students never discover them).

Each user in our studies enabled logging software in DrRacket. This software logged all key and mouse events in the environment and could replay them, so we could study the precise evolution of the program source and watch a user's editing as a "movie". (The logger is a DrRacket plugin, so it does not capture any interaction outside the environment such as passwords, etc.).

2.3 Analysis of Student Data

In spring 2010 we collected detailed logs of students' interactions with DrRacket during the lab sessions of an introductory programming course. Each lab ran for 50 minutes. There were six lab sessions (one per week) over the duration of the course. Sixty students out of 120 consented to provide data.

Using these data, we asked three initial questions about students:

1. What is the pattern of well-formedness of their programs? Do they keep their parentheses mostly balanced, or are there wild fluctuations of imbalance?
2. When modifying existing code, as opposed to creating new code, is their behavior any different? For instance, is there a significant difference in their edits after error messages as opposed to edits after the successful passage of tests (which suggests they have completed a task and are moving on to new code)?
3. Do these behaviors change over the course of the semester, as they (a) become familiar with the language syntax, (b) acquire greater skill with the programming environment, and (c) realize (from experts such as professors and TAs) that letting programs become significantly imbalanced is likely to lead to more errors and make it harder to find and fix them?

To observe the extent of unbalancing over the course of a lab, we computed the amount of unbalancing after each change to the parenthetical structure. We used an A* search with alpha-beta pruning to compute the shortest edit distance to a balanced buffer. The search could either add a parenthesis to match a dangling one, remove a dangling parenthesis, or turn a parenthesis into a different kind (from round to bracket or vice versa).

Figure 1 shows the behavior of three students in the third lab. The label (A, B, C) reflects their final course grade. All three start with a balanced buffer (because it's empty), then compose some code. While doing so, the buffer becomes momentarily unbalanced. The students differ in how they manage this imbalance. Student A seems to have developed a habit of frequently returning to a balanced buffer; she also frequently compiles her code, possibly to confirm its well-formedness before going further. Student B also frequently restores the buffer's balance, but does not compile as often. Student C spends almost the entire second half of the lab with a deeply unbalanced buffer. Afterwards, he has to make four compilation attempts before returning to a valid program.

We sought to summarize these behaviors with a metric so that we could make aggregate comparisons across students, across time, and so forth. A natural first metric is:

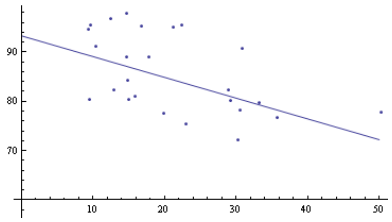
1. Area under the curve: We could simply sum the total extent of imbalance the student's program demonstrates during an editing session.

This metric does not, however, capture whether or not a student regularly brought the buffer back into complete balance. Instead, we might want to measure:

2. Average length of runs between zeroes: On the premise that a long imbalanced edit is worse than a short one (since

(a)

	DF	SS	MS	F Statistic	P-Value
x	1	441.571	441.571	5.16989	0.032214
Error	24	2049.89	85.4121		
Total	25	2491.46			



(b)

	DF	SS	MS	F Statistic	P-Value
x	1	81.5098	81.5098	0.811732	0.376561
Error	24	2409.95	100.415		
Total	25	2491.46			

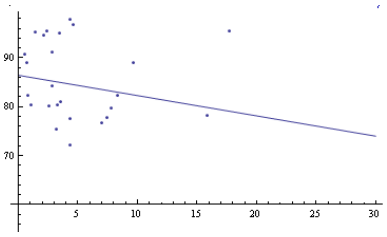


Figure 2. Correlation between the amount of imbalance (*x*-axis) and course grade (*y*-axis) when (a) writing new code (significant), and (b) fixing existing code (not significant).

the student must concentrate longer), we could emphasize the duration of imbalances over their size.

But this fails to capture our intuition that the more imbalanced a buffer is the greater the cognitive burden borne by the student (e.g., having to be conscious of the nesting depth). The summary measure we finally settled on combines the two previous metrics:

3. The mean of the areas under the curve between zeroes: Split at each point where the buffer returns to a balanced state; compute the area of each; average these areas.

This enables us to handle what might appear to be confounding data. For instance, suppose a student begins a program thus (where the arrow represents the cursor position):

```
(define (len l)
  (cond
    [(empty? l↑
```

The program has a nesting depth of four. It is unclear at this point whether the student will eventually balance everything. In contrast, a different student may always close every parenthesis they open right away, writing the above example as follows:

```
(define (len l)
  (cond
    [(empty? l↑)]))
```

By looking at sequences of edits, our metric lets us focus on the entire editing behavior, rather than trying to guess from an intermediate state what the final state will be.

	DF	SS	MS	F Statistic	P-Value
x	1	130.762	130.762	1.32939	0.260269
Error	24	2360.7	98.3625		
Total	25	2491.46			

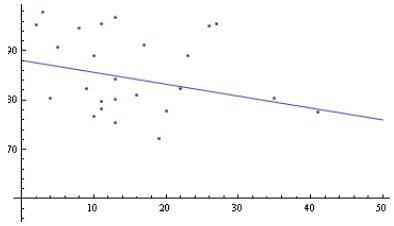


Figure 3. Correlation between the number of parenthesis errors and the final course grade (not significant).

Before analyzing the data, it is worth considering different styles of editing activity. In particular, writing new code is potentially quite different from changing existing code, since the former can much more easily be structural while the latter may necessitate non-structural changes. Thus, we choose to partition edits into those yielding new code versus those that were fixes to errors.

To classify edits, we assume (in line with prior work [3]) that edits after a successful compilation—until the next compilation—consist of new code. The one exception is when a compile returns a parenthesis error; in this case, we continue rather than terminate the edit sequence. Thus, new code sequences begin with a successful compilation, continue through any number of compilation attempts signaling a parenthesis error, and end on either another successful compilation or a non-parenthesis error. We classify all other edit sequences—starting at a non-parenthetical error—as fixing errors. We considered only edit sequences which added or removed at least one parenthesis.

Given this proposed operationalization of expression integrity, and our assumption that successful development of expression integrity is essential for student success in this programming course, we can then ask whether our metric correlates with the final course grade (unfortunately, individual lab assignments were not graded). We thus computed this correlation across the set of students who provided data for at least four labs during the course.

Figure 2 shows the result of the correlation. For episodes of writing new code (in (a)), the linear fit is statistically significant ($p=0.032$ albeit with an R^2 of 0.18). When fixing errors (b) it was not significant.

One might wonder whether this relationship occurs because better students make fewer parenthesis errors overall. But this is not the case. Figure 3 shows there is a slight correlation between the number of parenthesis errors and the final course grade, but this relationship is not statistically significant. We could also ask whether students' score on the metric improves over time. Unfortunately, because the difference in difficulty between labs is neither uniform nor monotonic, the between-task variability would mask any effects of learning. Indeed, we are not able to find any significant improvement effects across the term.

2.4 A Study of Experts

We believe that the buffer imbalance of experts can serve as a useful baseline for our expectations of students. We were able to conduct a small study of experts at the annual Racket user conference. Our study participants build large systems in Racket,

ranging across academia, government, and industry. Some of them indeed use DrRacket on a daily basis.

Due to the limited availability of their time, instead of asking them to write full programs, we instead gave them a small correct, working program, and asked them to refactor it to remove some operations and use others instead; the change did not alter the program's behavior, only its syntax. All 10 participants successfully completed the task.

In designing the task, we ensured that the edit could be performed purely with the s-expression operations, so that it would be possible to never change the parenthetical balance of the program (and, in particular, keep the imbalance at zero). Some experts did strive to keep their buffer's parentheses balanced, and some seemed to use the compiler to confirm well-formedness of their program at intermediate points. Figure 4 shows the distribution of the metric for both students and experts (with experts in (c)). The students do score higher, but the difference could be attributable to the fact that students were creating new (and buggy) programs, not merely refactoring existing code.

In particular, our logging information suggested the experts made almost no use of the s-expression editing commands. As a follow-up, we conducted a survey to ask them whether

1. this was indeed true; if so,
2. whether their behavior in this study was representative of their usual DrRacket programming style; and, if so,
3. whether they mentally still viewed the program as a collection of trees or whether they viewed it as a sequence of characters.

All the participants confirmed that our observation of their behavior was correct. They stated that in general they made little to no use of the s-expression *editing* commands, though a few used the *traversal* commands. Despite this, they confirmed that they absolutely do view the program in terms of trees of expressions independent of which editor they are using, though some participants added that they sometimes also viewed the program in terms of lines (rather than characters), especially when they need to perform block edits. (We conjecture that so do the others, even though they did not say so explicitly.)

3. Semantic Studies

Different languages employ different computational models, which in turn rely on expression integrity in different ways. In functional programming, even control operators are expressions that return values to the surrounding computation. Each node in the "tree of expressions" corresponds to a value which substitutes for the node while reducing a program to a result. In imperative programming, expressions interleave with statements that store values in memory or specify control structure. The "tree of expressions" is insufficient for modeling semantics, in exchange for relying less on the tree model to explain computation.

In our functional programming context, then, we expect that students who lack expression integrity would struggle to understand how programs yield results. Operationally, we expect that students with a basic semantic sense of expression integrity

- Can explain how different expression types (arithmetic, data creation, control operators) reduce to values.
- Can explain what an individual expression contributes to a function's result.

A student with a strong sense of expression integrity should also reflect the following skills for maintaining and editing programs:

- They can identify the expressions whose values might change as a result of editing a particular expression.
- They can identify which sub-expressions need to be edited (and how) in order to change the result of an expression.

The metrics about the impact of edits point to a fundamental difference between functional and imperative programming. In the functional setting, all of our semantic metrics have an implicit *frame condition* [4]: if an expression changes, the values of its sibling expressions do not change. This is not true in an imperative setting, because the changed expression could have a side-effect that causes an unedited expression to produce a different answer. Put differently, in functional programming, the syntactic building block (the expression) corresponds directly to the semantic building block (the value). We would expect this difference to manifest itself in semantic studies of expression integrity within each programming style. However, space precludes us from discussing these issues more.

4. Related Work

Several works have sought to identify the component skills that novice programmers must learn. Mead et al. has an extensive survey of the topic [5], and identifies dependencies between skills. PROUST uses their skill decomposition to automatically detect which subgoals were intended to be met by a student's code [6].

Projects that attempt to operationalize the skills they identified are less common. Anderson and Reiser's LISP Tutor models the fine-grained goal-setting done by novices through 500 production rules [7]. These rules are then used to trigger automatic feedback, to select exercise sequences for the student, and to predict quiz performance.

Expression integrity (thinking of code as a tree) is an instance of *chunking*. Both Adelson [8] and Shneiderman et al. [9] find multiple lines of evidence that novice programmers transition

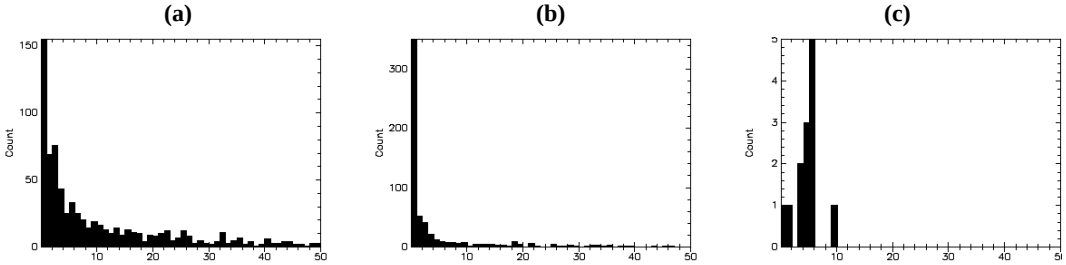


Figure 4. Histograms of the values of the metric of unbalancing for (a) students writing new code, (b) students fixing code, and (c) professionals during a refactoring exercise.

from syntactical chunking to some form of semantic chunking, but neither pinpoints the details. Fix, et al. give empirical support for five characteristics of how experts chunk the relationship between the functions and variables of a program, and to its goal [10]. We are not aware of any previous attempt to operationalize expression integrity specifically.

Some researchers have looked for patterns in beginners' editing behaviors. Rodrigo, et al.'s *error quotient* metric [3] summarizes the kinds and locations of errors a student makes, along with the locations edited in response. Like our metric of unbalancing, the error quotient correlates with final grades, but the two metrics satisfy different goals. The error quotient aims to detect where students struggle regardless of the conceptual cause, whereas we focus on the impact of expression integrity.

Dyke's work counts students' use of IDE features such as the multi-tab interface, breakpoints, and code auto-completion [11], and finds that these too correlate with final grades. In addition, the work finds that usage of more advanced features correlates with the rank difference between two members of a homework pair.

Ko, et al.'s study [12] of the editing behavior of expert Java programmers found patterns similar to those in Section 2.4: experts regularly pass through invalid buffer states while editing, but quickly repair the code and avoid prolonging the invalidity.

5. Perspective and Context

This paper outlines a research agenda centered around expression integrity, which in turn focuses on how students understand the structure of programs. We believe that understanding program structure, both syntactically and semantically, is a core skill in effective programming. We offer one preliminary (and language-sensitive) operationalization of expression integrity at the syntactic level. Naturally, we must address other syntaxes, study this concept at the semantic level, and correlate it with skills in debugging, maintenance, code reviews, etc.

Our preliminary findings, if borne out by more detailed studies, raise questions of what designers of programming environments should do in terms of syntax manipulation operations. On the one hand, fully-structured editors such as that of Scratch [13] are clearly popular and successful; indeed, the Scratch editor makes it impossible to create ill-formed programs. On the other hand, we are not aware of validation of the Scratch editor principle for large programs; any such studies would need to be reconciled with known data on structured editing [14]. What does seem apparent is that, given a free-form text interface, at least some experts employ this over structured commands. Whether it would still be worth teaching these commands to beginners to *instill* a notion of expression integrity needs further study.

Longer-term, expression integrity gives a foundation for studying the relative merits of functional and imperative programming for beginning students. Syntactically, programs correspond to trees in both styles. Semantically, the role of this tree differs significantly. By aligning the syntactic and semantic models through the tree structure, functional programming asks students to work with only a single mental model of programs. Imperative languages reduce semantic reliance on the tree model at the cost of introducing a second model with no framing condition. Understanding the cost-benefit tradeoffs of these models to each of program construction, debugging, and maintenance could provide significant input to the debate about suitable programming methods for novices.

Acknowledgments. We thank the US NSF for grant support, and appreciate comments from the anonymous reviewers and Matthias Felleisen. We thank the students and Racket users who participated in these studies.

6. References

[1] P. Miller, J. Pane, G. Meter, and S. Vorthmann, "Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University," *Interactive Learning Environments*, vol. 4, 1994, pp. 140-158.

[2] S. Krishnamurthi, "Programming Languages: Application and Interpretation," 2007.

[3] M.M.T. Rodrigo, E. Tabanao, M.B.E. Lahoz, and M.C. Jadud, "Analyzing Online Protocols to Characterize Novice Java Programmers," *Philippine Journal of Science*, vol. 138, 2009, p. 177–190.

[4] J. McCarthy and P.J. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," *Machine Intelligence*, vol. 4, 1969, p. 463–502.

[5] J. Mead, S. Gray, J. Hamer, R. James, J. Sorva, C. St. Clair, and L. Thomas, "A cognitive approach to identifying measurable milestones for programming skill acquisition," *ACM SIGCSE Bulletin*, vol. 38, Dec. 2006, p. 182.

[6] W.L. Johnson and E. Soloway, "PROUST: Knowledge-Based Program Understanding," *IEEE Transactions on Software Engineering*, vol. SE-11, Mar. 1985, p. 267–275.

[7] J.R. Anderson, F.G. Conrad, and A.T. Corbett, "Skill acquisition and the LISP tutor," *Cognitive Science*, vol. 13, 1989, p. 467–505.

[8] B. Adelson, "Problem solving and the development of abstract categories in programming languages.," *Memory & cognition*, vol. 9, Jul. 1981, p. 422–433.

[9] B. Shneiderman and R. Mayer, "Syntactic/semantic interactions in programmer behavior: A model and experimental results," *International Journal of Computer & Information Sciences*, vol. 8, Jun. 1979, p. 219–238.

[10] V. Fix, S. Wiedenbeck, and J. Scholtz, "Mental representations of programs by novices and experts," *INTERACT '93 and CHI'93 conference on Human factors in computing systems*, ACM, 1993, p. 74–79.

[11] G. Dyke, "Which Aspects of Novice Programmers' Usage of an IDE Predict Learning Outcomes?," *Special Interest Group on Computer Science Education (SIGCSE)*, 2011.

[12] A.J. Ko and H. Aung, "Design requirements for more flexible structured editors from a study of programmers' text editing," *CHI*, 2005, p. 1557–1560.

[13] D.J. Malan and H.H. Leitner, "Scratch for budding computer scientists," *ACM SIGCSE Bulletin*, vol. 39, 2007, p. 223.

[14] A.J. Ko, "Designing a Flexible and Supportive Direct-Manipulation Programming Environment," *IEEE Symposium on Visual Languages Human Centric Computing*, 2004, p. 277–278.