Lurch: A Word Processor Built on OpenMath that Can Check Mathematical Reasoning

Nathan C. Carter[1] and Kenneth G. Monks[2]

[1] Bentley University, Waltham, MA, USA
ncarter@bentley.edu
[2] University of Scranton, Scranton, PA, USA
monks@scranton.edu

Abstract

Lurch [5] is a free word processor that can check the mathematical reasoning in a document, including the steps of a mathematical proof, even one not written in a formal style. This paper covers our goals, implementation in terms of OpenMath, and a brief overview of the underlying validation engine.[1]

1 Introduction

1.1 Background

Lurch is a free word processor that can check the steps of a mathematical proof, and we have focused the current version on the needs of introduction-to-proof courses, covering topics such as logic, introductory set theory, and number theory. *Lurch* is built on OpenMath [1, 3] (as well as several other technologies, including Qt [16]) and is a free and open-source desktop application for Windows, Mac, and Linux. Its implementation is discussed in Sections 2 and 3.

1.2 Goals

Our user interface guideline is that users should be able to write math in whatever style or notation they prefer, with exposition inserted where they feel it's helpful. Especially in a pedagogical context it is important for the notation and style used in the software to match that of the textbook and lecture notes, so as not to be an obstacle for the student. Of course, pencil and paper won't tell you whether your proof is correct, but *Lurch* will, in addition to providing the other benefits that any word processor does, such as cut and paste.

Lurch is for student use. We're targeting students in their first proof-based courses, where *Lurch* can give frequent, immediate, clear feedback on the validity of the steps in their proofs. That's the part of the mathematical community we're excited about reaching.

Therefore *Lurch* is an entirely different kind of product than existing proof checkers (e.g., *Mizar* [9] or *Coq* [2]) in several ways. First, we do not require the user to learn a specific language or rules; in *Lurch*, the user (or his or her instructor) defines the rules from scratch. Second, *Lurch* does not automate any proof steps on the user's behalf, nor should it; the student types his or her work in *Lurch*, and *Lurch* grades, encourages, and coaches. Finally, proof checkers often have interfaces suitable for advanced users, sometimes requiring familiarity with the command line or shell scripts. *Lurch* is for the typical student, and its user interface is therefore a familiar one: It is a word processor that gives feedback visually in the document.

[1] *Lurch* was supported from 2008-2012 in part by the National Science Foundation's Division of Undergraduate Education, in the Course, Curriculum, and Laboratory Improvement program (grant #0736644). Views expressed in this document are not necessarily those of the National Science Foundation.

We want the only learning curve students encounter to come from the mathematics itself, not *Lurch*.

Existing research on educational technology suggests that our goals make sense and are achievable. Investigations into the effects of automated assessment systems (AiM [8] and MyMathLab [12]) show the value of computers in giving high-frequency, individualized, and immediate feedback [10, 14, 13, 15]. The value of such feedback is well documented in educational research (reviewed in [7]). This research matches our common sense that more feedback, delivered in immediate response to each of the student's actions, is better for learning.

2 Current Implementation

This section explains how far we have come towards achieving the goals in Section 1.2.[2] For more details than what this section provides, see [4] in this same volume. Sections 3 and 4 discuss details of implementation under the hood.

Consider the screenshot of the Mac version of our application in Figure 1. The red and green thumbs and yellow lights interspersed throughout the document are *Lurch*'s feedback on the correctness of each step of the user's work. (This feedback can be hidden to obtain a clean view for printing.) A green thumbs-up icon follows a correct step of work that is correctly justified, a red thumbs-down icon follows a step that contains an error or is supported by the wrong reason, and yellow lights follow mathematical expressions that are used as premises to justify later steps of work, but are not themselves justified (e.g., the hypotheses in a proof).

Students can hover their mouse over any of these colored icons for a brief explanation of its meaning, or double-click it for a detailed report. In order to provide this kind of feedback, *Lurch* needs to know the *meaning* of the mathematical expressions in the document.

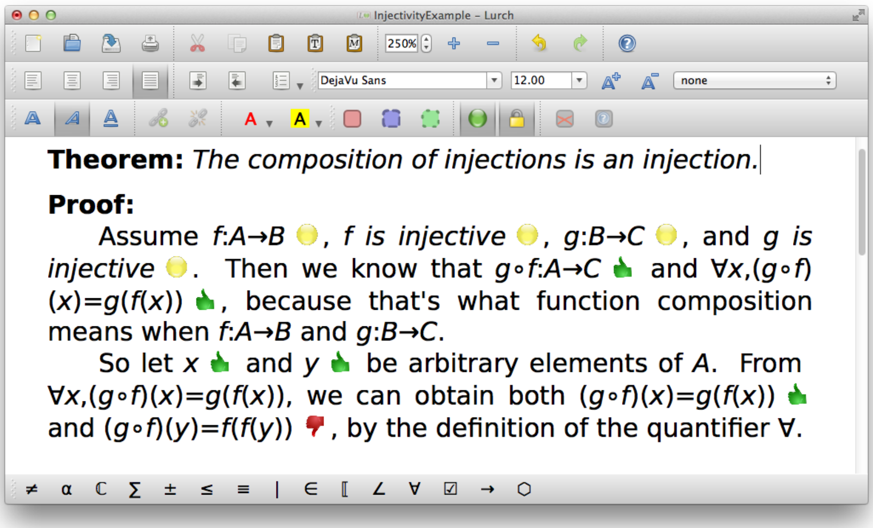
In order for a section of text to be treated as a mathematical expression, the user must mark it as such with a single click of a toolbar button (or the corresponding keyboard shortcut). The user selects a portion of text that he or she wishes *Lurch* to treat as meaningful mathematics, and then clicks the "Meaningful expression" button on the toolbar. *Lurch* then wraps the text in a bubble and reports the understood meaning in a tag atop the bubble, as in Figure 2. *Lurch* draws a bubble around a mathematical expression if and only if the user's cursor is inside it. Both *LyX* and *Word* have somewhat similar interfaces for editing mathematics.

The user will create bubbles often, and thus doing so must be intuitive and take near-zero time. The "Meaningful expression" button appears on the *Lurch* toolbar near formatting options such as bold and italic. It is just as easy to mark text as meaningful as it is to use one of those common formatting commands; a single click or keystroke does it.

This interface enables *Lurch* to *know* which portions of the user's document he or she considers meaningful mathematics. Also, in line with our goals from Section 1.2, the user is stylistically free to structure their document their own way. Because meaning is determined by what sections of text have been bubbled, the user can alternate whenever they like between expository text in any natural language and meaningful mathematics, and arrange their work as informal sentences, formal proof structures, or varying levels in between.

This interface also brings a pedagogical benefit. Student users, just learning what a proof is, are forced to indicate what parts of their own proofs are meaningful and which are not. Experience from testing shows us that students come away from a *Lurch*-integrated course knowing very solidly how a proof is structured, because they had to point out this structure to *Lurch* before it would check their work.

[2]At the time of this writing, the current version of *Lurch* is 0.7992, released on May 9, 2013.



Theorem: *The composition of injections is an injection.*

Proof:

Assume $f{:}A{\to}B$, f *is injective*, $g{:}B{\to}C$, and g *is injective*. Then we know that $g\circ f{:}A{\to}C$ and $\forall x,(g\circ f)(x)=g(f(x))$, because that's what function composition means when $f{:}A{\to}B$ and $g{:}B{\to}C$.

So let x and y be arbitrary elements of A. From $\forall x,(g\circ f)(x)=g(f(x))$, we can obtain both $(g\circ f)(x)=g(f(x))$ and $(g\circ f)(y)=f(f(y))$, by the definition of the quantifier $\forall$.

Figure 1: A *Lurch* screenshot explained in Section 2. Although this screenshot was captured on a Mac, *Lurch* is free and open-source for all three major platforms, Mac, Windows, and Linux.

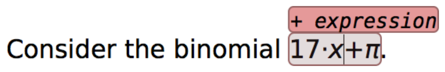
Lurch is not yet at version 1.0, and changes to the current design and interface will come as we evolve our ideas, especially in response to testing. There are many advanced interface features we'd already like to add, including automatic bubbling of statements and reasons, and a visualization of the inference and property graphs. Such features will be able to be enabled or disabled as the user's skills and needs require.

3 OpenMath in *Lurch*

3.1 Types of meaning and document structure

All *Lurch* documents are stored as OpenMath data structures, from the document level (using application of symbols for concepts like "paragraph") down to the atomic level (using OpenMath integers, strings, variables, and so on).[3] In order to see how these structures are formed, we

[3]Although OMDoc [11] may be a better choice for storing documents, the current choice was one of expediency. The word processing technology we're leveraging stores documents as hierarchical data structures, so using OpenMath objects as generic hierarchical containers was an easy way to save, load, and manipulate such



Consider the binomial $17\cdot x+\pi$.

Figure 2: In *Lurch*, bubbles around meaningful expressions appear if and only if the user's cursor is inside the expression. In this example, the bubble tag "+ expression" indicates that the expression is a sum.

must first cover a bit more about meaning in *Lurch* documents.

Lurch makes the following three major categories of bubbles available to users, and provides toolbar buttons for creating each. The full set of rules stipulating which bubble types can contain others is given in [6], but no bubble can contain any other bubble only partially.

Meaningful Expressions (hereafter, MEs): Shown using red bubbles, MEs represent mathematical objects like numbers, polynomials, and statements (as in the previous section).

Properties: Shown using blue bubbles, Properties specify attributes of MEs but are not MEs themselves.

Contexts: Shown using green bubbles, Contexts delimit the scope of declared variables and indicate subproof blocks constructed in the course of writing some kinds of proofs.

Properties can be used to modify neighboring MEs, for the purpose of attaching to them labels (like `\label` in LaTeX), reasons (citations of existing rules), and premises (references to earlier statements in the same proof). Although no bubbles are visible in Figure 1 because the cursor is at the end of the document, it contains many invisible ones. For example, on the third line of the proof, the expression $\forall x, (g \circ f)(x) = g(f(x))$ is an ME and the word "composition" on the next line is a reason property that applies to it and the preceding ME. Properties can apply to any number of adjacent, successive MEs, either to the left or right of the property itself. They do not alter the intrinsic meaning of MEs, but only tell *Lurch* how to work with those MEs (by adding labels, reasons, and so on).

It is possible to see the locations of all bubbles in a document; when a user enables that option, *Lurch* shows bubble boundaries as in Figure 3. For even more information, a user can direct *Lurch* to include the contents of each bubble's tag after its latter boundary. This results in a very cluttered view, but it is one way to see full information quickly.

An ME which is not contained inside any other ME is called a top-level ME. A Lurch document can be thought of as a sequence of top-level MEs $M_1, \ldots, M_n$. Although there is normally text between the M_i, *Lurch* ignores it for the purposes of validation and giving the user feedback. Properties assign attributes to these MEs and Contexts group together collections of MEs that have a common purpose or relevance.

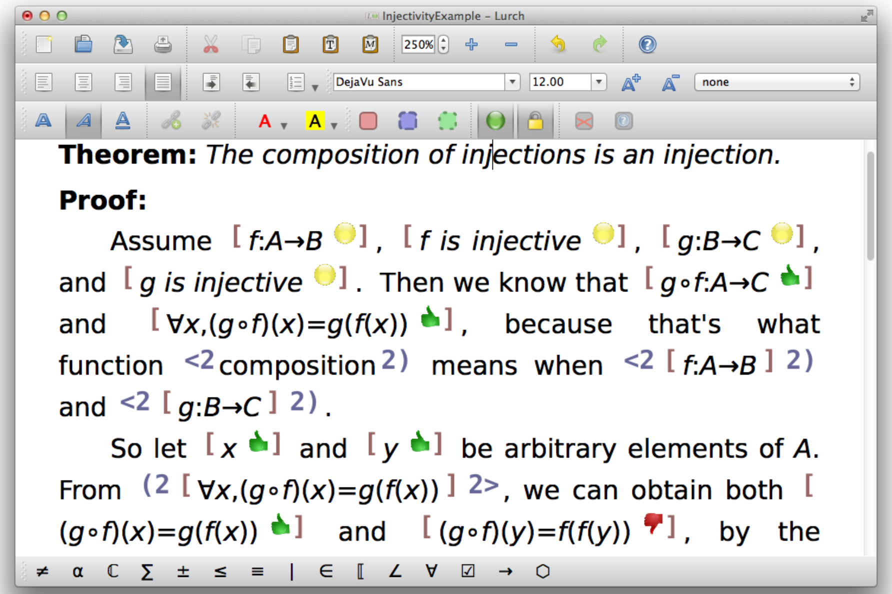
Lurch documents also depend on other *Lurch* documents in the same way that one chapter of a math book might depend on a previous chapter. The rules cited in Figures 1 and 3 are available because their definitions appear in a document on which that one depends. In *Lurch*, an imported document acts as if it appears, invisibly and in a read-only state, in its entirety, at the top of the document that imports it.

3.2 Creating OpenMath objects from MEs

Every ME is either a simple ME (containing no other MEs), of type integer, real, variable, constant, string, operator, or quantifier; or a compound ME (containing other MEs), of type function application or quantification. These correspond roughly to the major types of simple and compound expressions in OpenMath [3], and most are stored in exactly the expected way (e.g., an integer as an OpenMath integer). Every ME bubble lists its content type on its bubble tag.

Lurch infers the meaning of a simple ME from the text in its bubble. Thus a simple ME might contain the text `42`, and be interpreted as an integer, or contain `3.14159` and be interpreted as real. Similarly, `x` would be a variable, `"Life, the Universe and Everything"`

hierarchies.



Theorem: *The composition of injections is an injection.*

Proof:

Assume [*f*:*A*→*B*], [*f is injective*], [*g*:*B*→*C*], and [*g is injective*]. Then we know that [*g*∘*f*:*A*→*C*] and [∀*x*,(*g*∘*f*)(*x*)=*g*(*f*(*x*))], because that's what function <2 composition 2) means when <2 [*f*:*A*→*B*] 2) and <2 [*g*:*B*→*C*] 2).

So let [*x*] and [*y*] be arbitrary elements of *A*. From (2 [∀*x*,(*g*∘*f*)(*x*)=*g*(*f*(*x*))] 2>, we can obtain both [(*g*∘*f*)(*x*)=*g*(*f*(*x*))] and [(*g*∘*f*)(*y*)=*f*(*f*(*y*))], by the

Figure 3: *Lurch* with the option enabled to show all bubble boundaries. While this adds clutter, users can verify at a glance that bubbles are arranged as intended. ME boundaries are red square brackets. Property boundaries are blue (...> or <...) pairs that indicate whether they modify MEs to the left or right, with numbers to indicate how many sequential MEs they modify. No context bubbles appear in this example.

a string, + an operator, and ∀ a quantifier. If *Lurch* cannot guess the meaning of the contents of a simple ME bubble, such as `+++++`, it will classify it as a string, as if it were `"+++++"` (quotes added).

A compound ME can be obtained by nesting simple MEs in a LISP-like way, $[M_1 \dots M_n]$, with each M_i a simple or compound ME. If the head of the list is a quantifier (or other binding symbol) then the list is interpreted as a quantification; otherwise, it is interpreted as function application. It is possible (though obviously rarely desirable!) to enter mathematics into *Lurch* in this prefix-only notation. For example, $3 \cdot f(x+1)$ could be entered as the following expression, with square brackets representing ME boundaries, as in Figure 3.

```
[ [·] [3] [ [f] [ [+] [x] [1] ] ] ]
```

Because most users would rather type the expression in a more standard notation, *Lurch* tests the contents of every simple ME bubble to see if those contents can be understood by a built-in parser. If so, then that ME bubble is treated as the (possibly compound) ME determined by the parsing, while still appearing as a single bubble in the document. This allows a compound ME to be entered with the same ease as a simple ME and usually produces more legible results. For example, $3 \cdot f(x+1)$ can be entered in the typical calculator notation `3*f(x+1)`, or `3·f(x+1)`. Such MEs have the same meaning as the fully “expanded” version shown above.

Users can expand and collapse MEs between these two forms by using the context menu on the ME bubble itself. The parser is a convenience feature that allows users to enter compound MEs in a natural notation, but in every case the actual meaning of an ME is its expanded form, which is indicative of how it is stored as underlying OpenMath data.

Eventually the language that *Lurch* parses will be customizable, but the current version just uses a standard mathematical syntax. Here are some example expressions the parser currently understands.

```
x^2-1=(x-1)·(x+1)          3^2>2^3

∀x, x∈A∩B ⇔ x∈A and x∈B    { x : x∉x }
```

Lurch has a symbol toolbar so that users can click to enter symbols like ∀ and ∉; this is the bottom toolbar in Figure 1, with each symbol a button that opens a palette of related symbols. The software also automatically replaces common LaTeX codes for such symbols with the symbols themselves, so users who prefer keyboard shortcuts can learn those codes.

Although the internal storage for these well-known symbols uses standard OpenMath symbols such as `subset` from content dictionary `set1`, that may need to change because the meaning of the symbol in *Lurch* is whatever the user gives it by means of defined rules, which may or may not match the meaning in the content dictionary.

The current version of *Lurch* does not have mathematical typesetting capabilities, but later versions will include them. In particular, superscripts, subscripts, and fractions must be input using calculator-like shortcuts, such as `x^2`. But our current focus is on introduction to proof courses, and so *Lurch* comes with topics such as logic, set theory, and number theory, which don't really need much more than this plain notation (with a healthy supply of mathematical symbols).

4 Fundamental Theorem of *Lurch*

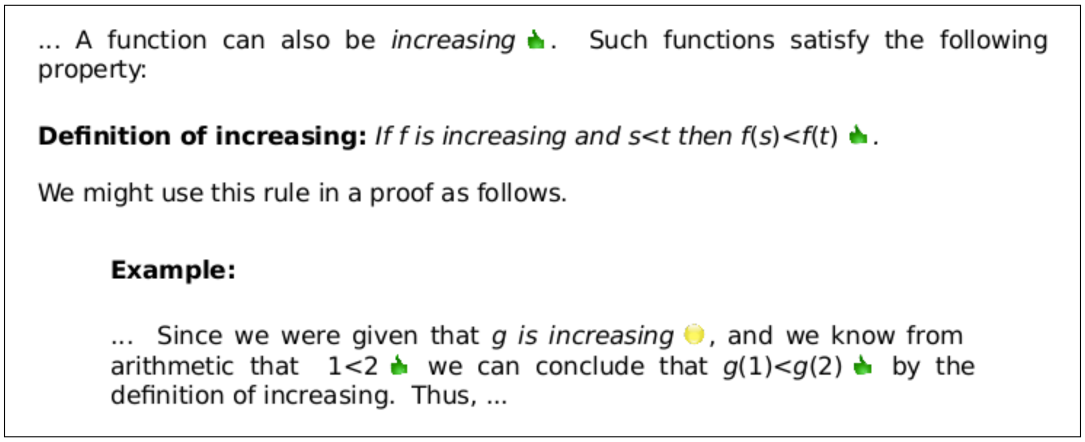
The raison d'être of *Lurch* is validation of student work. For each step of student work, *Lurch* should give feedback about the correctness of that step. In most cases, an ME is a step of work if and only if it is a top-level ME and the user has attached a reason to it; details can be found in the *Lurch* Advanced Users' Guide [6]. That document specifies, for each type of top-level ME, whether it should be validated, and if so, how. Here is one example, the case for MEs representing variable or constant declarations (based on ideas in [17]).[4]

A constant declaration or variable declaration will get a green validation icon if the following conditions are met.

1. It is not already declared, i.e., it is not in the scope of the declaration of the same identifier either as a variable or a constant.
2. It does not have a reason assigned to it.

If either of these conditions are not satisfied, the validation icon for the rule will be red and double clicking or hovering over it with the mouse will give a message indicating which of the conditions was not met, and how you might go about fixing it.

[4]The Advanced Users' Guide is a specification of a system, with no attention given to the motivations behind the design. *Lurch* is not yet at version 1.0, so the design is not set in stone, and thus a motivating document is not yet written.


... A function can also be *increasing* 👍. Such functions satisfy the following property:

Definition of increasing: *If f is increasing and s<t then f(s)<f(t)* 👍.

We might use this rule in a proof as follows.

Example:

... Since we were given that *g is increasing* ●, and we know from arithmetic that 1<2 👍 we can conclude that *g*(1)<*g*(2) 👍 by the definition of increasing. Thus, ...

Figure 4: In this sample *Lurch* document, an instructor specifies a new definition and illustrates how it can be used. *Lurch* validates that the term "increasing" is not already defined as something else, certifies that it understands the instructor's new rule, and validates the statement $g(1) < g(2)$, justified by citing the name of the new definition as a reason and providing the two required premises. The instructor specifies the rule by bubbling it as shown in Figure 5.

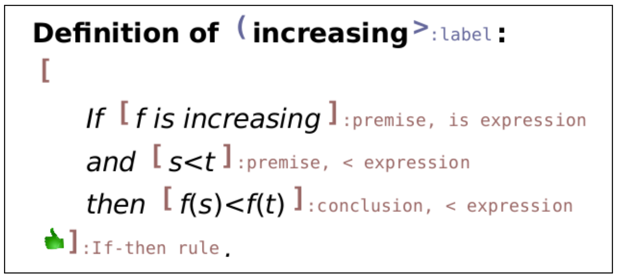
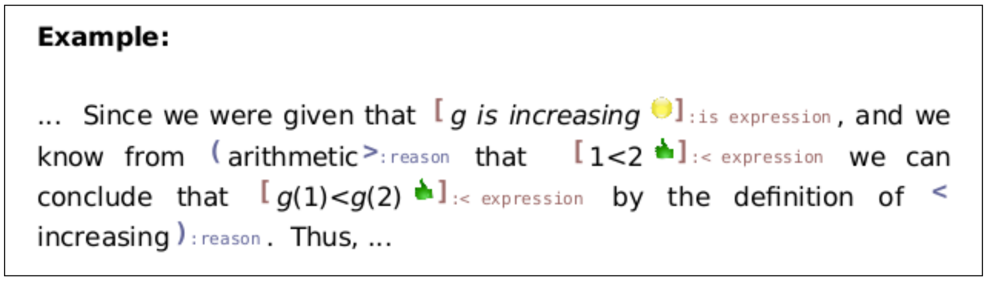

Definition of (**increasing**>:label **:**
[
If [*f is increasing*]:premise, is expression
and [*s<t*]:premise, < expression
then [*f(s)<f(t)*]:conclusion, < expression
👍]:If-then rule.

Figure 5: To alert *Lurch* to a new rule the instructor only has to mark the rule, its label, and any premises and conclusions. Here we see the semantic structure of the rule defined in Figure 4.

Such a specification exists for each type of ME in *Lurch*, but the nontrivial ones are too complex to list here. Instead we illustrate the algorithm with an example.

All of the rules and definitions that are available to students when doing their assignments in *Lurch* are written in *Lurch* documents directly. Users who want to customize the mathematics in *Lurch* can inspect those built-in, foundational documents, copy them, and alter them to suit their own style, or write new ones from scratch. As mentioned above, a document containing rule definitions can be included in another document as a dependency, making the rules in the former available in the latter.

The libraries that ship with *Lurch* define rules that are based on the natural deduction style of proof used in the second author's introduction to proof course. However there is no need to use these rules at all. An instructor can type his or her definitions, rules, and theorems directly into *Lurch* to make new dependency documents that the students can then use when working on their assignments.


Example:

... Since we were given that [*g is increasing*]:is expression, and we know from (arithmetic>:reason that [1<2]:< expression we can conclude that [*g*(1)<*g*(2)]:< expression by the definition of < increasing):reason. Thus, ...

Figure 6: To use the new rule defined in Figure 4 the user cites it by name, "increasing." In this example the user has also cited another rule called "arithmetic" to justify the claim that $1 < 2$.

For example, Figure 4 shows a snippet of lecture notes that an instructor might type into a *Lurch* document to define a new rule. He first declares a new term (*increasing,* bubbled as a constant declaration) which *Lurch* verifies is not already defined. He then specifies his proposed definition in the form and with the level of detail and specificity that he desired. The more advanced bubble structure that the instructor must use to alert *Lurch* to this rule's structure appears in Figure 5, using visible bubble boundaries as introduced in Figure 3.

Finally, when a student uses the rule to justify a statement, he or she does so by supplying the two required premises, and justifies the expression $g(s) < g(t)$ by citing as a reason the name of the rule ("increasing" in this case), as in Figure 6. *Lurch* checks that for the appropriate values of f, s, and t in the rule definition, the required premises are available to justify the desired conclusion.

In addition to the syntactically defined rules *Lurch* currently supports, we would also like to add rules validated by a built-in computer algebra system. Doing so would enable us to support typical algebra or calculus topics as naturally as we currently support proof-related ones.

This paper does not include all the details of the validation algorithm, but interested readers can inspect [6]. With such a specification in hand, we can state and prove the following theorem.

Theorem (One-pass Validation)**.** *Validating each top-level meaningful expression in a* Lurch *document, according to the requirements in [6], in the order they appear in the document, will result in a correctly validated document.*

In other words, the results of validation at point P never change the results of validation at some earlier point Q, and therefore *Lurch* can validate a document quickly, not having to read it more than once. Although mathematicians sometimes write documents for which this strict ordering of ideas and reasons fails to hold, Velleman [17] makes an excellent case that professors usually do not permit students who are still learning proof writing to structure their arguments like that. Therefore in a learning tool like *Lurch*, such a restriction is acceptable, or even desirable.

The proof of such a theorem involves showing that the specification for each type of meaningful expression depends in no way on the results of validation of meaningful expressions later in the document. The efficiency benefits of this theorem have not yet been integrated into *Lurch*, because both the design and implementation are still maturing.

5 Conclusion

Lurch was tested throughout an introduction to formal logic class the first author taught at Bentley University in the fall of 2008, and an introduction to proofs course for mathematics and mathematics education majors the second author taught at the University of Scranton in the spring of 2013. In both cases, student response was very positive, the details of which appear in [4].

Lurch has many opportunities for improvement, including testing in courses not taught by the developers and new features (typeset mathematics, more efficient validation algorithms, more attractive visual representations of premise relationships, and an even less obtrusive bubbling interface). But the current version, despite its opportunities to improve in many ways, shows all the signs of having been very beneficial in the courses in which it has been used.

The *Lurch* team invites additional collaborators for writing new mathematical topics in *Lurch*, doing further classroom testing, and software development.

References

[1] Olivier Arsmac and Stephane Dalmas. OpenMath INRIA C/C++ libraries. [online], Downloaded May 18, 2008. Available at `http://www.openmath.org/software/index.html`.

[2] Yves Bertot and Pierre Castéran. Coq'Art: the calculus of inductive constructions. Springer, XXV, 472 p. ISBN 978-3-540-20854-9, 2004. Available at `http://www.labri.fr/perso/casteran/CoqArt/index.html`.

[3] S. Buswell, O. Caprotti, D.P. Carlisle, M.C. Dewar, M. Gaëtano, and M. Kohlhase, editors. The OpenMath standard, version 2.0. The OpenMath Society, June 2004. Available at `http://www.openmath.org/standard/`.

[4] Nathan Carter and Kenneth G. Monks. *Lurch*: A word processor that can grade students' proofs. In David Aspinall, Jacques Carette, James Davenport, Andrea Kohlhase, Michael Kohlhase, Christoph Lange, Paul Libbrecht, Pedro Quaresma, Florian Rabe, Petr Sojka, Iain Whiteside, and Wolfgang Windsteiger, editors, *Proceedings of the Workshops and Work in Progress at the Conference on Intelligent Computer Mathematics*, number 1010 in CEUR Workshop Proceedings, Aachen, 2013.

[5] Nathan C. Carter and Kenneth G. Monks. *Lurch*: a word processor that can check your math. [online], 2013. A free and open-source software project hosted on SourceForge.net, available at `http://lurch.sf.net`.

[6] Nathan C. Carter and Kenneth G. Monks. Introduction to *Lurch*: advanced users guide, version 1.0. [online], 2013. Available at `http://lurch.sourceforge.net/AUG-v1.html`.

[7] A.W. Chickering and Z.F. Gamson. Seven principles for good practice in undergraduate education. American Association for Higher Education Bulletin, pages 3–7, 1987. Available at `http://aahebulletin.com/public/archive/sevenprinciples1987.asp`.

[8] Gustav Delius and Neil Strickland. AiM—assessment in mathematics. [online], Accessed on April 22, 2013. A free and open-source software project hosted on SourceForge.net, available at `http://sourceforge.net/projects/aimmath/`.

[9] Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz. Mizar in a nutshell. Journal of Formalized Reasoning, 2010. Available at `http://jfr.unibo.it/article/view/1980/1356`.

[10] Saliha Klai, Theodore Kolokolnikov, and Norbert Van den Bergh. Using Maple and the web to grade mathematics tests. In IWALT 2000 Conference Proceedings, 2000. Available at `http://rc.fmf.uni-lj.si/matija/ACDCA2000/Kolokolnikov-Klai-Bergh-pdf.pdf`.

[11] Michael Kohlhase. An open markup format for mathematical documents, OMDoc 1.2. [online], 2009. Available at `http://omdoc.org/pubs/omdoc1.2.pdf`.

[12] Pearson Education. MyMathLab. [online]. A series of mathematics courses on the World Wide Web, available at `http://www.mymathlab.com/`.

[13] Chris J. Sangwin. Assessing higher mathematical skills using computer algebra marking through AiM. In Proceedings of the Engineering Mathematics and Applications Conference (EMAC03, Sydney, Australia), pages 229–234, 2003. Available at `http://web.mat.bham.ac.uk/C.J.Sangwin/Publications/Emac03.pdf`.

[14] Chris J. Sangwin. Assessing mathematics automatically using computer algebra and the internet. Teaching Mathematics and its Applications, 23(1):1–14, 2004. Available at `http://web.mat.bham.ac.uk/C.J.Sangwin/Publications/tma03.pdf`.

[15] Michelle D. Speckler. Making the grade: A report on the success of MyMathLab in higher education math instruction. Pearson Education, Boston, MA, 2005. Available at `http://www.mymathlab.com`.

[16] The Qt Project. Qt 4.8. GNU General Public License (and the lesser) version 3.0, 2012. A cross-platform application and UI framework for developers using C++, QML, CSS, and JavaScript, available at `http://qt-project.org/`.

[17] Daniel J. Velleman. Variable declarations in natural deduction. Annals of Pure and Applied Logic, 144, 133–146, 2006.