ENHANCING AUTOMATIC SPEECH RECOGNITION FOR MATHEMATICAL APPLICATIONS VIA INCREMENTAL PARSING

Marina Isaac School of Computer Science & Mathematics, Kingston University, U.K.
Eckhard Pfluegel School of Computer Science & Mathematics, Kingston University, U.K.
Gordon Hunter School of Computer Science & Mathematics, Kingston University, U.K.
James Denholm-Price School of Computer Science & Mathematics, Kingston University, U.K.

1 INTRODUCTION

Automatic speech recognition and automatic speech understanding systems have, over recent years, improved to the extent that they are used in many practical applications ranging from dictation systems to voice control of household devices (through systems such as Alexa and Amazon Echo) and dialogue systems for telephone shopping and customer services for utility companies. However, advancements in speech input systems for mathematical applications have tended to lag behind those for more general or commercial situations.

Our system *TalkMaths*, which has been under development for several years, is an exception to this, and allows spoken dictation and editing of mathematical text (in standard mathematical notation) using relatively natural spoken language commands. However, up to now, correcting a mistake in a spoken form of a mathematical expression has required a complete re-parse of the spoken input, which is time consuming and potentially frustrating to the user.

In this paper, we discuss ways of improving on this situation using incremental approaches to parsing. These were first devised to make the parsing and compilation of computer program code more efficient, by only reparsing those parts of the program code which had actually changed, and merging the parse trees of the unchanged and modified parts of the code. We adapt this methodology to allow editing of spoken forms of mathematical expressions, and their associated parse trees, and describe and discuss initial experiments to compare the performance of these novel methods with those of more conventional approaches.

2 MOTIVATION AND RELATED WORK

2.1 Speech Technology, Disabled People and Mathematics

As noted above, Speech Technology – Automatic Speech Recognition (ASR), Speech Understanding and Text to Speech (TTS) – systems have become much more sophisticated and reliable over recent years, and they are now finding practical applications in many areas. Many people with disabilities – including those with visual impairments, motor impairments and/or damage to (or loss of the use of) their hands or arms – will often find suing conventional computer interfaces and other devices difficult or even impossible, and Speech Technology can greatly be of benefit to them.

Proficiency in mathematics – at least at an elementary level – is essential for many careers and educational courses. However, Speech Technology applications for Mathematics have not developed at the same rate as for other domains, thus putting people with the types of disability mentioned above at a disadvantage in both their education and career prospects.

However, some systems have been developed which interface automatic speech recognition systems with mathematical software. Examples of these include *MathTalk*™ and *Math Speak & Write.* These have been reviewed in some of our previous papers (Wigmore et al 2008, 2009, Attanayake et al 2014), but are mostly commercial systems or depend on specific commercial software. All of them have their individual limitations, and most support a rather limited mathematical vocabulary.

2.2 Spoken Mathematics

Like more general spoken utterances, the way people tend to describe mathematical equations and formulae when speaking spontaneously tends to be somewhat vague and use a somewhat unpredictable vocabulary (Wigmore et al, 2009). This leads to possible problems – what the speaker has said may be ambiguous or imprecise, resulting in a listener or an automated system misinterpreting the meaning which was intended to a lesser or greater extent.

Hence, there is a conflict between making an appropriate vocabulary and syntax for spoken mathematics precise (to avoid ambiguity) but difficult to learn and use, or much more natural and easy to use but retaining the risks of ambiguity and misinterpretation. Fateman (2013) discussed the issue of ambiguity in some detail. At the simplest level of arithmetic, ambiguity in mathematical content is resolved by rules of operator precedence and associativity. As greater complexity is introduced, the lack of constructs for grouping elements together becomes apparent : for example, without these it would be impossible to dictate the expression $\sqrt{b^2 - a}$ **,** a mechanism is needed to allow the user to specify such delimiters whilst minimising the associated cognitive load. We have decided to follow, but adapt, Fateman's principles, aiming to minimise the potential for ambiguities whilst trying to keep the language used relatively simple and easy to learn and use. Our *TalkMaths* system addresses the issue of grouping elements through by the use of templates, which range from simple bracketing structures ("square root"…"end square root" for the above example) to multiple-part templates to permit specification of elements such as integrals.

2.3 Development and Evolution of *TalkMaths*

Our own speech-based system, *TalkMaths*, for creating and editing mathematical text has been under development since around 2005. It uses a commercial Automatic Speech Recognition (ASR) system *Nuance Dragon Naturally Speaking*[*] to transcribe a user's spoken description of an mathematical expression (using our simple but prescriptive syntax and vocabulary, see Attanayake et al 2014 or Wigmore et al 2009 for details) into text. This is then parsed, converted into a parse tree structure, and thence into *MathML* or *LaTeX* code which can then be rendered into conventional mathematical notation and displayed on a screen or printed page.

The *TalkMaths* system processes mathematical expressions, transcribed from their spoken form, to produce a parse tree that can be further manipulated. An attempt is made to resolve errors in syntactically incorrect expressions. The first generation (Wigmore 2011, Wigmore et al 2008) used an attribute grammar, but this did not support incomplete input. A later version (Attanayake 2014, Attanayake et al 2014) introduced features which allow dictation and processing of incomplete mathematical expressions, tasks that were not feasible using such an attribute grammar, so used an operator precedence (OP) grammar to define the language, with the addition of *mixfix* (also known as *distfix*) operators to implement the templates. Parsing was performed using a version of the XGLR parsing algorithm of Begel and Graham (2006) adapted for OP parsing, incorporating error recovery strategies to handle incomplete input. However, one drawback of that version was that, if an expression were subsequently edited or extended, the revised transcribed input would have to be completed re-parsed in order to convert the new version into *LaTeX* or *MathML*, before it could be rendered on a screen or page. This reparsing was likely to be wasteful of time and computational resources for all but the simplest of mathematical expressions, and could lead to significant delays before the edited version could be viewed – particularly if the user was relying on a mobile device with limited memory and processing power.

With this in mind, we sought a solution which would maintain the benefits and flexibility of earlier versions of *TalkMaths*, but which should be more efficient when reprocessing expressions after they had been edited. This led us into the concept of *incremental parsing*, which had originally been developed for the purpose of efficient parsing and compilation of computer program code after editing. This is described in some detail in the next section.

[*] https://www.nuance.com/en-gb/dragon.html

3 INCREMENTAL PARSING

3.1 Incremental Parsing for the Compilation of Computer Program Code

Incremental parsing was first developed in the 1980s to address similar issues of the need to completely reparse and re-compile computer program code, even after just a very small, simple change had been made to the high level source code. Once the original code has been parsed, the result can be represented as a parse tree, a hierarchical structure of nodes (representing variables and operators acting on them) connected by branches, where the structure indicates the groupings and elements and the order(s) in which operations must be performed. Ideally, rather than reparsing the whole program (or, in our case, mathematical expression) after an edit, only the minimum amount of re-processing which is essential to produce the appropriately modified parse tree should be performed.

An incremental parsing algorithm effectively "performs surgery" on these trees, cutting them at appropriate points and, where necessary, "grafting" one tree onto another in a suitable way. Taking an editor-centric approach, Lalonde & des Rivieres (1981) and Kaiser & Kant (1985) developed a set of node operations to achieve these, allowing for binary operators, brackets, functions and (in concept) unary opartors such as negation or taking a square root. The process was simplifed to just two operations, *split* and *merge,* on a complete parse tree to allow a substantially more economical re-parse, by Heeman (1990), and his approach also made allowances for the possibility of the input being incomplete. Our approach for reparsing mathematical text following editing is an extension of Heeman's method. Most significantly, it handles Attanayake's (2014) composite template nodes by expanding on Heeman's (1990) approach which used simple bracket pair matching.

3.2 Incremental Parsing for Mathematical Text

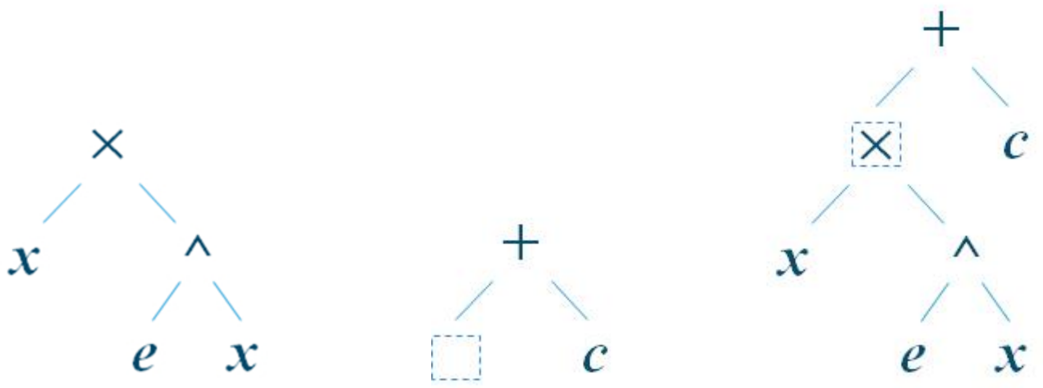
As stated above, we have adapted Heeman's (1990) technique using the *split* and *merge* operations to produce a new parse tree from an old one after a mathematical expression has been edited. We have modified Attanayake's (2014) parser to incorporate these. The approach is best illustrated using some simple examples.

3.2.1 Example 1 : Adding an additional term to an existing expression

Consider the following example, deliberately kept simple. Suppose the user has already correctly dictated expression A, "x-ray times echo to the power of x-ray", to represent the mathematical expression $x \times e^{x}$, and the appropriate parse tree has been produced. He/she then wants to edit this, just to add a constant term on the right, i.e. produce the revised expression $x \times e^{x} + \mathrm{c}$.

Without using incremental parsing, there are two ways in which this could be achieved : **either** the transcribed text version for A could be kept along with the corresponding parse tree, **or** the text for A could be rebuilt from its parse tree. The transcribed text for B would then be appended to the existing text for A. Finally, the entire text for A+B together, "x-ray times echo to the power of x-ray plus charlie", would be re-parsed. This would be time consuming.

The process can be implemented more efficiently using incremental parsing, employing the *merge* operation applied to the original parse tree for A and appending the very simple parse tree for B : Firstly, parse the expression to be added on, namely "plus charlie" (for "+ c"). Then concatenate the tree for A with the tree for B, giving the appropriate parse tree for A+B together "x-ray times echo to the power of x-ray plus charlie", see Figure 1 below.



Parse tree for expression A Parse tree for expression B The merged parse tree for A + B

Figure 1 : Combining two parse trees A and B using the *merge* operation. The empty square represents a “placeholder” for an expected but currently missing entity, eventually filled by the parse tree for expression A.

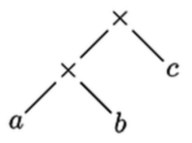
3.2.2 Example 2 : A more complicated change to an existing expression

Consider the following slightly more complicated example. Suppose a user has already dictated the expression “alpha time bravo times charlie”, to represent the mathematical expression “a x b x c”, and that the appropriate parse tree has been produced. The user then wants to modify this to give the result “a + d × c”, i.e. equivalent to a + (d × c) once the correct precedence rules for arithmetical expressions has been followed. It is difficult to see how this could be achieved without incremental parsing, other than by dictating the entire new expression “alpha plus delta times charlie", then parsing the new expression from scratch. This effectively wastes the work done processing the original expression.

If incremental parsing can be used, then the following sequence of operations can be performed on the old tree :

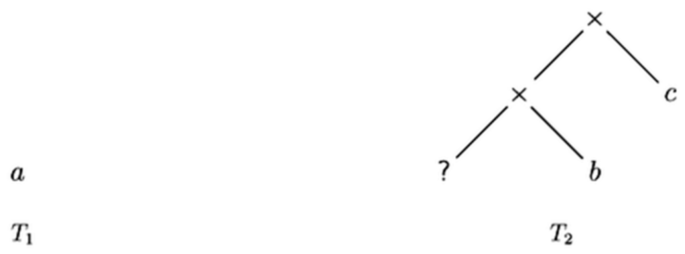
“alpha times bravo times charlie”

$a \times b \times c$



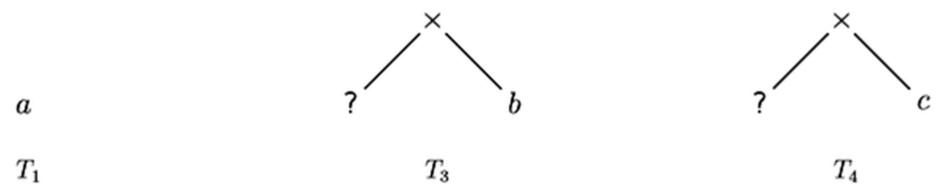
Split the original tree into two trees, T_1 and T_2, after the a :

Step 1 – after user gave some editing command(s) – split after a into trees T_1 and T_2



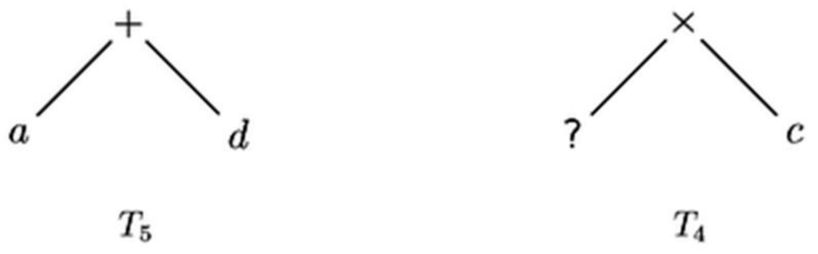
Then divide the larger tree T_2 into two parts, T_3 which (in this case) is discarded and T_4 which needs merging with the tree corresponding to the amendment. In this case, T_3 needs to be replaced by a new parse tree T_3' representing the replacement “plus delta” (“ + d ”).

Step 2 – split T_2 after b into trees T_3 and T_4

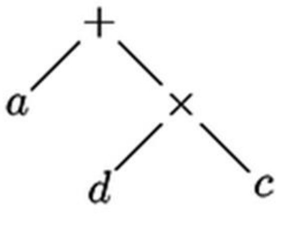


Produce the parse tree T_5 for the amendment, “alpha plus, by editing T_1 if appropriate, and edit T_4 to make it represent “delta times charlie". Finally, *merge* T_5 and T_4 to give the required result, which can be converted to *MathML* or *LaTeX* and then rendered on a page or screen.

Step 3 – merge new tree T_3' for $+d$ with T_1 to give T_5.



Final step: merge T_5 and T_4



4 IMPLEMENTATION

The incremental parsing operations described above have been incorporated into a modified version of Attanayake’s (2014) Operator Precedence Parser. This has been written in Python and runs on a PC under Windows 8. As described in our earlier papers (e.g. Attanayake et al 2014), the system takes transcribed spoken forms of mathematical expressions as input, and produces the corresponding parse tree and *LaTeX* or *MathML* code as outputs.

5 INITIAL EXPERIMENTS AND RESULTS

In order to evaluate whether our incremental parsing approach is more time-efficient than re-parsing the complete expression after it has been edited – and, if so, to what extent – we have devised two experiments for evaluation of our method in comparison with the complete re-parse :

Experiment 1 : Suppose a text string S corresponding to a mathematical expression has already been parsed, and the user wishes to append a newly supplied token string T. For example, the user may wish to add "plus charlie" (representing "+ c") to an existing expression "X-ray times echo to the power X-ray" (representing "$x \times e^x$") – see Example 1 of section 3.2.1 above.

In the conventional "batch parsing" (re-parse after editing) approach, this would require two stages ; firstly, concatenate the text for S and T to give ST, then parse the resulting text ST.

The incremental parsing approach retains the previous parse for S, and parses the new text string T. Finally, the two parse trees for S and T are merged, as described in section 3.2.1 above. This would be expected to be more efficient than the batch approach provided the tree for T is simpler (in some appropriate sense) than that for S.

Experiment 2 : Suppose a mathematical expression, transcribed as a text string XtY (where X and Y are token strings, and t is a single token) has been dictated incorrectly, with the user having missed out token t, and the incorrect version XY has already been parsed. For example, the intended expression should have been "X-ray squared plus charlie" (representing "x^2 + c"), but it has been incorrectly dictated as "X-ray plus charlie" (representing "x + c"). Let us assume that the incorrect version XY has already been parsed, and the corresponding parse tree obtained.

In the conventional "batch parsing" approach, the missing token t (in the example above, "squared") is then dictated and inserted between X and Y, and the text string X, the text token t and the text string Y are concatenated to give XtY. This resulting text string is then parsed and the new corrected tree obtained.

In our incremental parsing approach, the additional token is dictated and transcribed. The old tree, corresponding to XY, is *split* after the last token in X into separate trees T_X and T_Y**.** The additional token t is then parsed, and the corresponding (very simple) tree obtained. The tree for t is *merged* with T_X, and finally the resulting tree is *merged* with T_Y.

Both our experiments were performed on transcribed spoken forms of various common mathematical expressions, plus some taken from a dataset available from the University of Oxford Mathematical Institute (2016). Due to the nature of the tasks, the datasets for the two experiments were not identical.

In Experiment 1 (appending a string), the incremental method outperformed the batch approach in 798 out of 802 cases, and on average only took one third the time of the batch method.

In Experiment 2 (inserting an omitted token), the incremental method outperformed the batch approach in all 657 cases, and on average took one ninth the time required by the batch method.

Since the time required for a parse will increase with the length of the corresponding token string (and also with the complexity of the parse tree), we decided to use a "normalized performance measure" defined for any single expression as :

(Time for batch parse – Time for incremental parse) / (Number of tokens in expression)

Values for this performance measure for our two experiments are given in Table 1 below.

	Experiment 1	Experiment 2
Maximum	0.0366	0.0236
Minimum	-0.00104	0.00693
Mean	0.0155	0.0179
Median	0.0156	0.0178
Standard Deviation	0.00855	0.00231

Table 1 : Values of our “Normalised Performance Measure”, comparing the “Incremental Parsing” and conventional “Batch Parsing” methods across our datasets of spoken mathematical expressions and edits to them. The values correspond to times in seconds, given to 3 s.f.

6 DISCUSSION, CONCLUSIONS AND FUTURE WORK

As can be observed from the results in Table 1, the average (mean or median) values of the normalised performance measure are positive, and substantially in excess of the standard deviation, for both experiments. These values indicate that our incremental method is typically more time efficient than the conventional batch parsing approach when applied to mathematical expressions in this dataset. To quantify this in terms of statistical significance, noting that the distributions of values are not Normally distributed for either experiment, having performed 26 repetitions of parses of the complete dataset to average out random effects in computer performance, we carried out Wilcoxon signed rank and Mann-Whitney U tests (the non-parametric analogues of the t-test) on our results, obtaining high statistical significance ($p < 0.005$) in all cases. This indicates that our method is significantly more time efficient, according to our metric, with respect to this dataset of expressions. Comparisons of the aggregated timings for the two methods for the two experiments are shown in Figure 2 below.

In conclusion, we have succeeded in implementing an incremental parsing approach to processing spoken forms of mathematical expressions, and this appears to be more time efficient with respect to re-processing such expressions after editing compared with traditional “batch parsing” methods. If fully integrated with our *TalkMaths* system, this should lead to our system being more practical for use, particularly if used on relatively low performance devices such as tablet PCs or mobile ‘phones. This should make the system more usable “on the go” and hence more useful to disabled researchers, students, teachers and others.

The next steps in this project will be to incorporate these methods into a new working version of *TalkMaths*, and then to adapt the algorithms to permit creation and editing of computer programming code via spoken input, incorporating the incremental parsing approach.

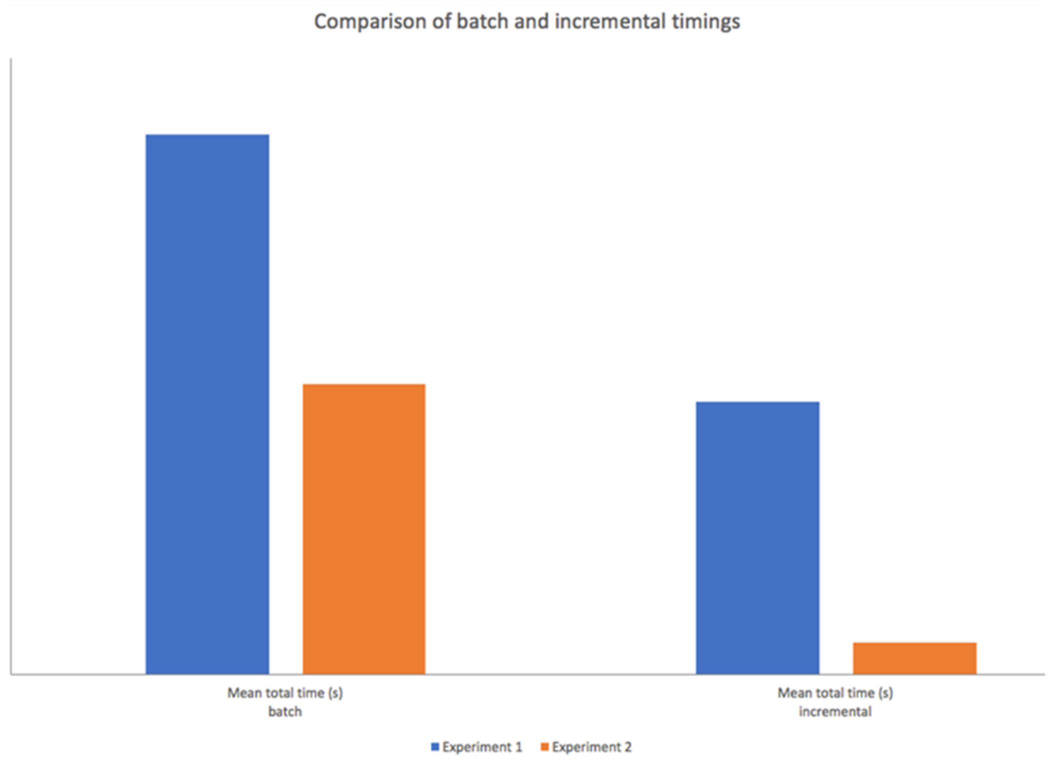


Figure 2 : Comparison of mean total time (average of 26 repetitions, with times aggregated over the whole dataset for each experiment). The pair of bars on the left hand side show the results for conventional batch parsing, whilst the pair of bars on the right hand side show the corresponding results for our incremental parsing method. Experiment 1 (appending of a token string) in blue, Experiment 2 (insertion of a single token) in orange.

7 REFERENCES

1. A.M. Wigmore, G.J.A. Hunter, E. Pflügel & J. Denholm-Price (2009) "*TalkMaths* : A Speech User Interface for Dictating Mathematical Expressions into Electronic Documents", Proceedings of ISCA Workshop on Speech and Language Technology in Education (SLaTE'09), U.K.
2. R. Fateman, "How can we speak math?," (2013) [Online]. Available: http://www.eecs.berkeley.edu/~fateman/papers/speakmath.pdf. [Accessed 18 March 2018]
3. A.M. Wigmore (2011) ""Speech-Based Creation and Editing of Mathematical Content", PhD Thesis, Kingston University, London.
4. A. Wigmore, G. Hunter, E. Pflügel & J. Denholm-Price (2008) "Using Automatic Speech Recognition to Dictate Mathematical Expressions: The Development of the '*TalkMaths*' Application at Kingston University", Journal of Computers in Mathematics and Science Teaching, Vol. 28(2), pp 177-189
5. D. R. Attanayake, J. Denholm-Price, G. Hunter and E. Pfluegel (2014) "*TalkMaths* over the web: a web-based speech interface to assist disabled people with mathematics", *Proceedings of the Institute of Acoustics*, Vol. 36, Part 3, pp 435 – 442.
6. D. Attanayake (2014) "Statistical Language Modelling and Novel Parsing Techniques for Enhanced Creation and Editing of Mathematical E-Content Using Spoken Input", PhD Thesis, Kingston University, London.
7. A. Begel and S. L. Graham (2006) "XGLR - an algorithm for ambiguity in programming languages," *Science of Computer Programming,* vol. 61, no. 3, pp. 211-227.
8. W. R. Lalonde and J. des Rivieres (1981) "Handling Operator Precedence in Arithmetic Expressions with Tree," *ACM Trans. Program. Lang. Syst.,* vol. 3, pp. 83-103.
9. G. E. Kaiser and E. Kant (1985) "Incremental parsing without a parser," *Journal of Systems and Software,* pp. 121-144.
10. F. C. Heeman (1990) "Incremental parsing of expressions," *Journal of Systems and Software, Vol. 13,* pp. 55-69.
11. University of Oxford Mathematical Institute (2016) "Solutions for admissions test in Mathematics, Computer Science and Joint Schools",
URL: https://www.maths.ox.ac.uk/system/files/attachments/websolutions15_1.pdf