Incremental Parsing Algorithms for Speech-Editing Mathematics and Computer Code

Marina Jennifer Isaac

A thesis submitted in partial fulfilment of the requirements of Kingston University for the award of Doctor of Philosophy in Computer Science.

March 2020

Acknowledgements

I would like to express my gratitude to the following people and entities.

Eckhard Pfluegel provided unwavering support and knowledge throughout this project, and shown great patience over the breaks I needed to take due to personal circumstances.

Gordon Hunter provided advice on statistical matters and helped in preparing material for publication.

James Denholm-Price provided advice on programming languages and helped in preparing papers for publication.

Dilaksha Attanayake explained the workings of *TalkMaths*.

David Wertheim provided valuable support and discussion.

Jackie Deacon and Rosalind Percival helped me navigate the university's administrative procedures.

The research was partially sponsored by Kingston University, London.

Finally, I wish to thank all the species of order Hyracoidea for inspiring me through their beauty. May they continue to exist in perpetuity.

Abstract

The provision of speech control for editing plain language text has existed for a long time, but does not extend to structured content such as mathematics. The requirements of a user interface for a spoken mathematics editor are explored through the lens of an intuitive natural user interface (NUI) for speech control, the desired properties of which are based on a combination of existing literature on NUIs and intuitive user interfaces. An important aspect of an intuitive NUI is timely update of display of the content in response to editing actions. This is not feasible using batch parsing alone, and this issue will be more serious for larger documents such as computer program code. The solution is an incremental parser designed to work with operator precedence (OP) grammars.

The contribution to knowledge provided by this thesis is to improve the efficiency in terms of processing time, of the OP incremental parsing algorithm developed by Heeman, and extend it to handle the distfix (mixfix) operators described by Attanayake to model brackets and mathematical functions. This is implemented successfully for the TalkMaths system and shows a greatly reduced response time compared with using batch scanning and parsing alone. The author is not aware of any other incremental OP parser that handles such operators. Furthermore, a proposal is made for modifications to the data structures produced by Attanayake's parser, along with appropriate adjustments to the incremental parser, that will in the future, facilitate application of OP grammar to program code or other structured content by changing the definition of its content language.

Publications by Author

Some of the work in this thesis has been published in the following peer-reviewed conference papers.

- Isaac, M. J., Pfluegel, E., Hunter, G. and Denholm-Price, J. (2015), Intuitive NUIs for speech editing of structured content (work in progress), *in* ‘26th Annual Workshop of Psychology of Programming Interest Group, PPIG 2015’, pp. 1–5.
- Isaac, M. J., Pfluegel, E., Hunter, G., Denholm-Price, J., Attanayake, D. and Coter, G. (2016), Improving automatic speech recognition for mobile learning of mathematics through incremental parsing, *in* ‘Intelligent Environments 2016’, pp. 217–226.
- Isaac, M., Pfluegel, E., Hunter, G. and Denholm-Price, J. (2018), Enhancing automatic speech recognition for mathematical applications via incremental parsing, *in* ‘Proceedings of the Institute of Acoustics’, Vol. 40, pp. 342–349.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Background

Given the ubiquity of electronic publishing, the means of creating content via computer should be available to anyone who wishes to or is expected to do so as part of their work, study or leisure activity. This includes people who have a disability that prevents them from using traditional input modalities easily, such as keyboard, mouse or touch screen. One alternative method of controlling a computer that does not require expensive additional hardware is via spoken commands.

Automatic speech recognition (ASR) software has been widely available for a number of years, and ranges from digital assistants such as Apple's Siri[1], which require an Internet connection and are designed to respond to simple commands expressed in natural language, to dictation and Windows control software such as Dragon by Nuance[2], developed with plain language document creation and editing in mind. While such facilities may now be commonplace, very little exists to support creation and editing of structured content such as mathematics or computer code. The issues with spoken computer code are well documented (Desilets, 2001; Begel, 2005; Desilets et al., 2006; Pfluegel et al., 2011; Gordon and Luger, 2012), as are those specific to spoken mathematics (Fateman, 2013; Wigmore, 2011; Attanayake, 2014).

One characteristic that is needed to support a natural-feeling interface for any content of this type is timely feedback to the user on the result of their actions. While delay

[1]See `https://www.apple.com/uk/siri/`.

[2]See `https://www.nuance.com/en-gb/dragon.html`.

due to parsing fairly short mathematical expressions might be acceptable to the user, this will not be the case for components of program code. In the latter case, not only will large amounts of content be built up piece by piece (necessarily broken up this way as the user will need to pause to breathe as they dictate), but its display will need to be updated appropriately as the user makes changes. As recognised by Ghezzi and Mandrioli (1979; 1980), to achieve this, a fast incremental parsing algorithm is required.

Given the nature of mathematical expressions, the most appropriate grammar to use for their internal representation is an operator precedence (OP) grammar. Attanayake (2014) developed a batch parser for the *TalkMaths* system (henceforth referred to as *TalkMaths*) which builds up valid abstract syntax trees (ASTs) based on spoken mathematics. These trees include a modelling of bracket structures and functions with more than one operand. The aim of the work described in this thesis is to develop an incremental OP parsing algorithm that will work with such ASTs, thus extending the content that will be handled by previously developed OP parsing algorithms, and to investigate its feasibility for application to any structured content but in particular, computer program code.

1.2 Aim and Objectives

The aim of this project is to contribute to research on speech-driven user interfaces used to edit structured content such as program code, facilitating development of tools to enable maintenance of such content either by speech control only, or in a multimodal fashion that includes speech. These may be used by people with relevant disabilities, or in environments that preclude the use of input devices such as keyboards or tablets.

The objectives are as follows.

- Elicit the characteristics that a user interface should have to provide a natural user experience when using speech control, from a large cross-domain of literature on both user experience and speech interfaces, to inform the design and implementation of a system for creation and editing of mathematical expressions.
- Design and implement a novel incremental parsing algorithm based on manipulating trees representing mathematical expressions and similar content generated by an operator precedence parser.

- Identify and discuss any modifications required to the parsers or data structures in order to handle the creation and editing of spoken computer program code.

1.3 Thesis Contributions

In this thesis, three main contributions to knowledge are made.

First, a number of intuitive natural user interface principles are identified, to be applied when designing speech controlled applications, and in particular, those appropriate for editing mathematical expressions. This is the first time the principles for natural user interfaces and intuitive interfaces have been combined and applied to a speech environment. An important factor in providing such a user interface is that of timely update of displayed content in response to the user's actions. Given that system response time when reparsing an entire expression is unacceptably long, incremental parsing techniques are required to achieve this behaviour.

The second main contribution of this thesis is an extension of the incremental operator precedence parser developed by Heeman (1990) to handle unary operators and the mixfix operators modelled by Attanayake (2014). The parsing algorithms' efficiency is also improved by using an alternative method to identify incomplete composite nodes, and by redesigning the top-level operations using an iterative approach, rather than simply converting the recursive algorithms to their iterative versions. The provision of incremental parsing, along with performance improvements made to the implementation of the Attanayake (2014) batch parser, will enable *TalkMaths* to be used for longer expressions that require multiple utterances to dictate.

Finally, an extension to the parser developed here and the Attanayake (2014) parser are discussed, that will be required to permit them to be used for the modelling of programming constructs, handle editing operations more flexibly, and allow constructs and identifiers to be named by the user in a more natural way.

1.4 Structure of this Thesis

Chapter 2 is an extended version of Isaac et al. (2015), which reviews and evaluates literature on natural user interfaces (NUIs) and intuitive interfaces, combining these to form a list of intuitive natural user interface (INUI) principles. After a discussion on what may be considered to feel natural in a speech interface, and discussing requirements specific to editing structured content such as mathematics or program code, the INUI principles are adapted for this type of speech editing environment.

Chapter 3 reviews the literature on incremental parsing in general, and operator precedence (OP) incremental parsing in particular, including the ways in which OP grammars may be used to represent programming constructs. It finishes by taking a detailed look at the algorithm developed by Heeman (1990), including suggested extensions.

After introducing the notation and terminology used in the remainder of this thesis, Chapter 4 describes in detail the extensions to the Heeman (1990) algorithm, using the mixfix operator model described by Attanayake (2014). The theoretical time complexities of the top-level operations are derived, and compared with their equivalents as would be performed using the Attanayake (2014) batch parser.

Chapter 5 introduces the *TalkMaths* parser developed by Attanayake (2014), and describes how the algorithm developed in this thesis is implemented to work with it. Given the difficulty of direct comparison of the theoretical time complexities of the two parsers, the chapter finishes by presenting a comparison of practical performance for example editing scenarios.

Chapter 6 explores the issues involved with adapting the algorithm for use on programming languages rather than mathematics, and makes recommendations on how these may be tackled to allow *TalkMaths* to evolve into a generalised speech-controlled structured content editor.

Chapter 7 presents conclusions and makes suggestions for future work.

Chapter 2

User Interfaces for Speech

2.1 Introduction

For many years, products such as those offered by Nuance[1] have provided high functionality for speech control in a variety of common spoken languages. The main area that has benefited from spoken dictation facilities is word processing; structured content such as mathematical text or computer program code has been very much left behind in this respect, because of its specialised formatting and punctuation. To make speech control a viable option for editing such content, a more natural style of interaction is required.

This chapter discusses the definitions of natural user interfaces (NUIs) and expands on the notion of intuitivity, in order to derive a list of basic intuitive NUI properties. These are then interpreted for user interfaces that use speech recognition and, taking into account some of the issues encountered when designing for spoken mathematics, principles to which a *TalkMaths* user interface should adhere are proposed.

TalkMaths is the product of an ongoing development programme at Kingston University, aimed at providing a facility for users to create and edit mathematical content using speech control. It was originally created as a result of Wigmore, Hunter, Pfluegel, Denholm-Price and Binelli (2009) and Wigmore (2011), and has undergone various

[1]See `http://www.nuance.com/dragon/index.htm`

changes since then. Currently it consists of a RESTful web service[2], based on the work of Attanayake (2014).

This chapter is an extended version of the paper presented by Isaac et al. (2015) in the 26th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2015, and includes a section on requirements specific to speech-based language-sensitive editors.

2.2 Literature Review

The concept of a NUI has currently been applied mainly to interfaces using touch and gesture (Wigdor and Wixon, 2011). This section traces the concept back to its origins and explores the related notion of intuitive interfaces.

2.2.1 Natural User Interfaces

The concept of the natural user interface was first developed by Fjeld et al. (1998; 1999) as part of a project to develop an augmented reality system designed to aid remotely situated users collaborating in the design of a physical artefact. The interface for such a task would have to minimise the discontinuity between the physical actions required to complete a task, and the user's mental problem solving process. The theoretical basis was that of the activity cycle which, in action regulation theory (Hacker (1994), as cited by Fjeld et al. (1999)), consists of iterations of goal setting, action planning, performance and evaluation. Within these steps, the user performs actions that may be *pragmatic*, which bring a task physically closer to completion, or *epistemic*, which aid the thought process of the user. A good illustration of these is given by Kirsh and Maglio (1994) who describe how the most accomplished Tetris players rotated tiles "physically" rather than try to imagine these transformations before moving a piece. Because these epistemic tile rotations were of benefit to the subjects of the study, the first NUI design guideline developed by Fjeld et al. (1998) was to allow users to perform such exploratory actions. To give users the confidence to behave in this way, the second guideline states that the negative effect of making any mistakes needs to be minimised. The third NUI guideline of Fjeld et al. (1998) is to allow voice as well

[2]A RESTful web service is one that provides access to a resource using representational state transfer, as defined by Roy Fielding in `http://www.ics.uci.edu/ fielding/pubs/dissertation/rest_arch_style.htm`

as any body part to be used for system interactions. Their fourth guideline suggests monitoring of the complete user environment, including interaction with artefacts such as visual projections (Rauterberg, 1999). Given that this is rather ambitious, it was dropped from their updated list of design principles (Fjeld et al., 1999).

Ten years later, as touch screen and gesture-based technology matured, researchers investigated the opportunities in User Interface (UI) design that arose from these (Wigdor et al., 2009). The book *Brave NUI World* (Wigdor and Wixon, 2011) presents a much cited practical guide for designers in these areas, suggesting ways in which many of the ideas of Fjeld et al. (1999) may be implemented in such interfaces.

The idea of NUIs is also addressed by Jetter et al. (2014) in their Blended Interaction framework, which attempts to predict what metaphors in user interface design will make sense by using the ideas of conceptual blending (Fauconnier and Turner, 2008) and image schemas (Hurtienne and Israel, 2007). Asikhia et al. (2015) also use image schemas in their work on intuitive interaction, finding that schemas that may be formed independently by both the user and designer are most likely to be successful.[3] It may be of interest to designers of speech UIs that language seems to be at the core of the ideas developed for direct manipulation: the conceptual blends suggested by Jetter et al. (2014) incorporate the metaphors involved in Hurtienne and Israel's image schemas, which themselves reflect the language used to describe relations between objects and actions to be performed on them (Hurtienne and Israel, 2007).

Ghosh et al. (2017) also explore the related concept of the *natural user experience*, the guidelines for which include the importance of context, (fast) response time and lack of ambiguity, for promoting the perception of "naturalness".

2.2.2 Intuitivity

A requirement that crops up in the original descriptions of NUIs is that the interface should be *intuitive* (Fjeld et al., 1998; 1999). For brevity, this property will be referred to as *intuitivity*[4], and the following definitions of its basic principles will be used.

[3]This relationship seems to be similar to the one between user and developer mental models referred to by Blackler and Hurtienne (2007).

[4]This term is already in use, and is preferable to *intuition*, which is commonly understood as referring to the subconscious human thought process.

Blackler and Hurtienne (2007) describe the concept of intuitivity as the situation where aspects of an interface appear familiar to a user by taking advantage of their existing knowledge; an additional condition is that the user should almost forget that they are having to work via a UI (Naumann et al., 2007). This lack of awareness should result in a lower cognitive load associated with use of the interface (Naumann et al., 2007), particularly when performing low-level actions during the activity cycle. Blackler and Hurtienne (2007) make several recommendations to help produce intuitive designs, the most relevant of which are included in the next section.

2.3 Intuitive NUI Design Principles

The following general principles represent an attempt to blend the definitions of intuitivity and other work on NUIs with the ideas of Fjeld et al. (1999). The list is grouped according to the general objectives of Fjeld et al. (1999).

Encourage epistemic actions and exploratory behaviour.

As proposed by Fjeld et al. (1998), this will help the user complete their task efficiently, exhibiting the exploratory behaviour that will help them progress to expertise in the application (Wigdor and Wixon, 2011; p.55).

1. Users with differing proficiency levels should feel comfortable using the software (Wigdor and Wixon, 2011; p. 13).
2. Provide alternative ways of invoking functionality for different classes of user, as well as employing other types of redundancy such as providing both text and icon to describe controls. (Blackler and Hurtienne, 2007)
3. Interaction with the system should feel *robust* (in the sense of error handling or recovery) to the user, so that they will have the confidence to attempt more advanced operations as they become more skilled.
    - For major changes or destructive actions, confirmation should be required and previews should be provided where appropriate. (Wigdor and Wixon, 2011; p. 55)
    - Minimise the impact of user errors (Fjeld et al., 1998) by allowing the user to reverse them easily (Fjeld et al., 1999).

The user should feel that their interaction with the system is *intuitive*.

The following principles pertain to this area (in which there is considerable overlap with the concepts of "naturalness").

4. Where conventions are already established in the application area or medium (Wigdor and Wixon, 2011; p. 13), adhere to these, otherwise use effective metaphors (Blackler and Hurtienne, 2007).
5. Take advantage (where appropriate) of the user's existing skills, to make their experience feel more familiar (Wigdor and Wixon, 2011; p. 13).
6. Facilitate the planning aspect of the activity cycle by indicating software state and available actions at all times, giving full context to the user (Wigdor and Wixon, 2011; p. 45) and (Fjeld et al., 1999; Ghosh et al., 2017).
7. Show the results of all user actions (Fjeld et al., 1999), with feedback being immediate, appropriate (Blackler and Hurtienne, 2007) and informative. Non-trivial feedback (for example, system messages) should aim to increase user understanding of the system and provide effective help where needed (Wigdor and Wixon, 2011; p. 56).
8. Clear affordance in the design of controls will aid the user in identifying both the function and mode of use of the controls (Fjeld et al., 1999; Blackler and Hurtienne, 2007; Ghosh et al., 2017; Wigdor and Wixon, 2011; p. 55).
9. The interface should reflect the user's mental model[5] of the system (Blackler and Hurtienne, 2007).

Context of use of the system should be taken into account.

The design should reflect:

10. the nature of the user's task rather than the technology of the application (Blackler and Hurtienne, 2007), as well as
11. the physical environment and social context in which the system is to be used (Wigdor and Wixon, 2011; p. 19).

[5] A mental model is the user's perception of the components of the software and how they interact.

2.4 What Feels Natural in a Speech Interface?

2.4.1 Type of Language

While natural language may seem the obvious choice for casual use or novice support, not only is it unclear how the inherent ambiguity of a language such as English could be applied to precise editing operations, but the need to use long sentences to describe repetitive actions is not desirable for the user. Research suggests that users prefer brief commands to natural language for such tasks (Elepfandt and Grund, 2012), and there is even evidence that humans opt for brevity (to the point of failing to convey sufficient information) when speaking to machines if permitted to use their own choice of words (Stedmon et al., 2011), because they expect the machine to be unable to handle more complex sentences.[6] This suggests that a language that provides a natural-feeling experience would be one that contains both simple and more complex versions of commands. Novice users should be reminded of basic commands by the interface, while at the same time they should be made aware of the more powerful versions, which would help them develop their mastery of the command language. This would form part of the "scaffolding"[7] described by Wigdor and Wixon (2011; p.53).

2.4.2 How Can Objects be Manipulated Using Speech?

The most well known discussion of NUIs (Wigdor and Wixon, 2011) deals with the modalities of touch and gesture, reflecting the natural progression from using keyboard and a pointing device to manipulate on-screen objects, to the subjective experience of manipulating them directly. Where automated facilities are not currently provided for certain types of content (such as mathematical expressions written in LaTeX), ASR users commonly use human scribes. Although it would be tempting to develop "assistants" to take the place of these people, they would not give the user any sense of direct manipulation of the objects on screen. The question is, how may systems provide voice commands that do this adequately?

A major challenge in this is how to select and manipulate objects when use of hands is

[6]As speech interfaces become more widespread and sophisticated, it would be interesting to see if such user behaviour changes.

[7]This is the term used to describe the means provided by the UI for allowing the user to become more skilled at using the software.

treated as an optional UI mode. One promising method is to combine speech with eye gaze to indicate the object in question (Elepfandt and Grund, 2012). Kaur et al. (2003) and Maglio et al. (2000) find that users naturally glance towards objects they intend to manipulate just before they do so, suggesting overall efficiency of a system might be improved by using this hybrid modality. When gaze is used as an adjunct to other modalities, the interface is in fact partially meeting the third original NUI guideline of Fjeld et al. (1998) through its use of natural behaviour; Sibert and Jacob (2000) have found that eye gaze can enable faster object selection than use of a mouse. Research in this area is ongoing – see for example Vieira et al. (2015) – and if privacy concerns about eye tracking can be overcome, this may become popular as the technology matures.

Before then, other means are required to refer to on-screen objects, such as the various types of grid described by Wigmore (2011). Most grids (boxes that identify objects that can be selected for manipulation) use numbered labels, that may cause confusion as numbers change due to changes in objects, and increase cognitive load through the user needing to recall label numbers. A better alternative is the semantic grid (Wigmore, 2011) that uses meaningful labels where possible. Additionally, rather than novice users such as children learning mathematics having to recall domain-technical terms such as “numerator”, the labels will remind and reinforce learning of such terms.

2.5 Requirements Specific to Speech-based Language-sensitive Editors

This section considers requirements that are specific to speech based systems, and in particular to language-sensitive editors – that is, those where the content is written in a formal language such as mathematics or computer program code. *TalkMaths* is used to illustrate the issues involved. This is a system for entering mathematical expressions, providing spoken commands for dictating common mathematical symbols and operators, and including specific editing commands that work together with a GUI. For example, the expression $a+b$ might be dictated by saying “alpha plus bravo”. Note that in order to spell out individual letters, most ASRs have difficulties in recognising them correctly unless spoken spelling conventions such as the NATO phonetic alphabet are used (Fateman, 2013). This is not expected to be a major restriction for experienced ASR users, but may pose problems for casual use. Typed input that uses plain letters should of course be permitted, as the system is also expected to be used multimodally for general mathematical input.

The issues that engender additional requirements for this type of software arise not only from controlling the computer using spoken commands, but also the vocabulary and structure of the content and editing commands.

These will be described in further detail in Section 2.5.3, but first some aspects of the content being authored need to be considered.

2.5.1 Spoken Mathematics and ambiguity

Wigmore, Hunter, Pfluegel and Denholm-Price (2009) found that when speaking mathematics naturally to one another, people would use prosody to indicate grouping of parts of an expression. For example, $\sqrt{a} + b$ would be read aloud as "square root of alpha; plus beta" while $\sqrt{a + b}$ would sound more like "square root of: alpha plus beta". This suggests that where there is scope for ambiguity, prosodic elements such as pauses may help identify which possible parse result was intended by the user. Current widely available ASR technology enables people to issue commands consisting of words to the computer but does not take prosody into account. This is a limiting factor in the naturalness with which such expressions may be spoken.

To deal with the problem of ambiguity, Attanayake et al. (2012) proposed the suggestion of alternative words or phrases based on statistical language models as part of their error correction strategy, and investigated the use of predictive models to resolve ambiguities as part of their statistical parsing approach. Given the prevalence of predictive text in mobile devices, an equivalent of this may prove helpful for novice users who are unsure of the forms a command may take: for example uttering just the first word of a command (e.g. "fraction") could result in a "help" area displaying the syntax of this command, while uttering just "select" could display alternatives for the next word that may appear in a selection command used for editing.

2.5.2 Requirements Arising from Speech Control

The requirements described in this section relate to common tasks involved in creating and editing mathematical expressions that may be performed by users with a range of abilities. These may be people with difficulties using a mouse and keyboard but who are expert at mathematical representations such as equations written in LaTeX, as well as the more casual user who would only occasionally need to create or edit mathematical formulae but as noted by Fateman (2013), may find it easier to enter these using some spoken input.

For brevity, a command (or part thereof) that specifies content will be referred to in the same way as an action command. The requirements here can be summarised as follows.

Extensibility : The ability to extend the language by using macros with placeholders.

Concatenability : The facility to issue more than one command in a single utterance.

Cursor placement : Allowing specification of an insertion point as well as selection of parts of content.

The requirement of *extensibility* refers to the ability of the user to invoke a series of commands, or fragment of a command, saved under some "macro" name. Expert users who may frequently need to create similar forms of expression will be able to do this quickly, and certain wordy phrases such as "Greek mike hat" for $\hat{\mu}$ may be shortened to suit the user's taste. This choice of alternative methods to invoke functionality conforms to intuitive speech NUI principle 2.

The utility of this requirement becomes clearer where the system is used by practitioners who work in a variety of areas, with different (or even contradictory) notations. If action commands and content commands are treated as belonging to a single language, one could add another type of macro that in effect extends the content language by allowing placeholders. These would of course need to conform to the grammar of the common language[8]. A mechanism to check that new or altered command phrases do not already exist in the vocabulary would also be required; given that the modality of interest is with speech input, a check for high similarity in sound to other words may also be desirable.

Just as a user may want to use macros to allow them to reduce their speaking time, they will frequently want to be able to issue more than one command in a single utterance. Consider the example of an expression $\frac{a+1}{a+2}$ that should have been $\frac{a+1}{b+2}$. For only a reasonably experienced user, the quickest way of making the change may be to give commands "select denominator" followed by "select alpha" in a single utterance, before dictating the replacement, "bravo". To force the user to pause between the two selection commands would be undesirable as more advanced users will frequently

[8]Use of an operator precedence (OP) grammar makes this less of a daunting prospect via the use of distfix operators, which are discussed in the next chapter.

want to utter commands in a single utterance. This requirement is given the name *concatenability.*

The combination of custom commands and concatenation of these of course raises issues for language design, which will be discussed further in Section 2.5.3.

While Wigmore (2011) describes various grid mechanisms to enable selection of content for editing or deletion, a means of specifying an insertion point is required. This corresponds to the notion of a *cursor* in text editing environments. A means of positioning this other than by hand control is required. It should also be possible to specify multiple cursors (marks) that may be subsequently used as insertion points or as delimiters for selecting a block of text.

2.5.3 Language-related Issues

Language-related issues arise from the special vocabularies used for mathematics or computer programming, as well as operator precedence of mathematical expressions. Requirements pertaining to these may be summarised as follows.

Context-specific vocabulary This implies that when in the context of using the specialist software, the vocabulary needs to be restricted to words that will be analysed lexically as appropriate words within the context of that software, either as commands or the language of the specialised content type.

Error recovery Automatic error recovery strategies, that permit the user to modify the parsed output into the desired form with a minimum number of subsequent steps, should enable the system to handle erroneous inputs gracefully.

Ability to handle commands longer than a single utterance The system must allow a command to be given using more than one utterance, to handle commands that take longer to say in one breath.

Distinguish between a multi-utterance command and sequences of commands The overall effect of a sequence of commands should be the same, no matter where the user had to pause (thus producing a new utterance) while speaking them.

Handle ambiguous commands Where a command is ambiguous, the user should be presented with alternative interpretations (based on possible parses).

A simple example of the need to deal with context-specific vocabulary is where the user wishes to create expression $\frac{\pi}{2}$, so dictates "pi over two" (assuming "pi" is in the vocabulary). A general purpose ASR could recognise this as the string "pie over to", thus providing a string that cannot be parsed.[9]

Although the restriction of vocabulary at least partly addresses the issue of incorrectly chosen homophones, it does not deal with the problem of word misrecognition. There are a number of ways in which input that does not constitute one complete and correct command could arise. If the ASR software fails to interpret part of the user's utterance, the expression could be treated as incomplete. Alternatively, if it formed part of a longer expression, it would need to contain a gap (or "hole") that the user could fill in later. The software will also need to deal with extra words introduced by the user into a sentence in the command language that violate its grammar but form a valid sentence in natural language. For example, a spoken command "edit the numerator", contains the extraneous word "the". This suggests there may be a number of words that, depending on context, should be treated in the same way as filler sounds[10].

As discussed by Fateman (2013), the treatment of commands or structured input that spans more than one utterance is not a trivial issue, and different scenarios need to be considered.[11] The next two illustrations use the example of a user who wishes to construct the expression $\frac{a+b}{c+d}$, which translates (using an example mathematical content language given by Fateman (2013)) into "alpha plus bravo all over quantity Charlie plus delta".[12]

Suppose, having dictated this fragment, the user pauses after "all over", so the remainder of the expression is given in a second utterance. The assumption is made that as described above, the software recognises the form of the expression, and constructs a fraction with a missing denominator. Should the system (a) wait for a new command, which may be one that edits the fragment, or (b) wait for the continuation of the expression? Option (a) may be irritating to an experienced user through interrupting their train of thought, while (b) could cause difficulty if the user deliberately broke

[9]The author accepts this is a very artificial example, given that a modern ASR is likely to recognise the trigram correctly.

[10]Filler sounds are noises such as "um" and "er", that humans typically use to fill in thinking time during speech.

[11]If multimodal input is permitted by a system, users switching between speech and typing may make multi-utterance commands and expressions an even more common occurrence.

[12]The word "quantity" is used in this language as a left delimiter of a non-trivial denominator.

off from dictating the expression. The best approach would be for the software to do both: accept either a new command or the continuation of the input.

If the user paused instead after "alpha plus bravo", the software would interpret this as the complete expression $a + b$, so when the user gives the remainder: "all over quantity Charlie plus delta", this second utterance may be interpreted as a new and incomplete command. This imples that not only may incomplete commands (or expressions) be missing content at the end, or have "gaps" where spoken input was not properly recognised, but an utterance may be the continuation of a previous command or content description.

This implies that a system using the language should be able to accept continuation of input as well as the concatenation of the commands suggested by the concatenability requirement. This may cause problems, as illustrated by the following scenario.

Suppose a user issues consecutive commands "alpha plus bravo", "select alpha" and "plus bravo" as separate utterances (using a system that will accept dictation of new content to overwrite existing content). The sequence of commands will cause the system to replace in the expression $a + b$, the symbol a with $+b$, resulting in $+b + b$. Alternatively, issuing the commands all in one utterance, "alpha plus bravo select alpha plus bravo", would select the entire expression because the utterance would be interpreted as two commands (the second one starting with "select"). This demonstrates that in certain situations, the semantics of commands may change if they are issued in a single utterance, compared with their meaning as two separate utterances in sequence. If one still wants to allow for the time saving that occurs when concatenating commands, one would have to make the command language more flexible.

To state this problem from a mathematical point of view, the translation from command to action is not a homomorphism. That is, if operation $\cdot$ denotes command string concatenation, and $\circ$ denotes the equivalent editing operation (that is $a_1 \circ a_2$ means execute action a_1 followed by a_2) then if f denotes the translation to internal form then the requirement that for any editing commands s_1 and s_2,

$$f(s_1 \cdot s_2) = f(s_1) \circ f(s_2)$$

is not always met.

One solution would be to ensure that every s_1, s_2 is complete individually, which could be achieved by providing a delimiter to the "select" command, making the equivalent of the first version of the command sequence "alpha plus bravo select alpha end select plus bravo". The drawback of this is that extending the "select" command by two words may be undesirable from the user's point of view. To overcome that, a single utterance that contains only the opening part of such a command could be treated as having the closing part (e.g. "end select") at its end. That way, "end select" would only have to be spoken if the "select" command has another command following it in a single utterance. But then what happens if the user is genuinely spreading the command over two utterances? An alternative would be to allow the user to issue commands in two modes:

- *interactive* mode (the default), in which if there is ambiguity when the second command is issued, the user is queried on whether the second command is a continuation of the first, and
- *dictation* mode, in which it is assumed that the user is inputting a sequence of commands that are assumed to be individually complete.

Ambiguity in dictation mode may be handled in a way that is similar to greedy regular expressions[13], in that the above scenario will result in the ambiguous utterances being interpreted as single command "select alpha plus bravo" rather than command "select alpha" followed by content dictation "plus bravo".

The above paragraphs describe a mechanism to handle ambiguity that arises from the user pausing during their interaction. The other type of ambiguity relates either to ambiguities inherent to the language, or those introduced through erroneous input. The way in which such ambiguities are handled, and the order in which any alternatives are presented, needs to be considered. Attanayake et al. (2012) presented a a statistically-based approach for predicting which form will be the most likely, implemented in the *SWIMS* system, as an alternative to deriving a parse forest of all possible parses of the sentence (implemented on an experimental version of *TalkMaths*). Because one of the requirements of *TalkMaths* is to handle incomplete input (whether a truncated command or one that contains gaps), taking this into account when constructing the parse forest can result in a large number of possibilities, even when the command has only one correct interpretation. As well as causing longer response times in practice, it is the belief of the author that the user experience may be impaired by being presented with a large range of choices for what they thought was a simple expression.[14]

[13]Greedy regular expressions will match as long a sequence of input as possible.

[14]In practice, even "alpha minus bravo" produced a two-tree parse forest: one for $a - b$ and the other representing the concatenation of a and $-b$ (where the minus was unary), equivalent to $a(-b)$.

One method of dealing with this may be to have the behaviour controlled by a parameter (for example whether to use statistically-based methods to deal with ambiguity); another may be to use the following approach.

- If the sentence can be parsed without error such that it is unambiguous, accept it.
- If the sentence may be parsed in different ways without error (that is, it is ambiguous, but with each interpretation correct), require the user to choose from the interpretations found.
- If the input could not be parsed without error, list all (or maybe just some) of the possibilities. In this case, it would be desirable to list them either in ascending order of number of "holes" detected during the parse, or by using the statistical prediction method proposed by Attanayake et al. (2012).

2.6 How an Editor for Spoken Mathematics may Reflect INUI Principles for Speech

To avoid over-generalisation, given that speech interfaces may be used in a wide variety of environments, the context of the adaptation has been restricted to content editing environments involving a full sized screen for visual output. The assumption is made that a product such as those offered by Nuance or Microsoft is to be used, and that the user is not visually impaired.

Table 2.1 summarises the intuitive NUI principles described below for speech editing environments. The structure (and numbering) follow that used in Section 2.3. A fuller description of the principles, that are also informed by the experience of working with earlier versions of *TalkMaths*, is given in the following paragraphs.

1. *Users of varying proficiency should feel comfortable using the software.* Command reminders should be given to novice users, but expert or intermediate level users may be given the option to suppress them. The reminders may take the form of a list of most commonly used commands appropriate to the current situation, with an option to show them all. This way, a novice user would not have to resort frequently to use of a help system ("What can I say?"), and so the command list could form context sensitive help. A command history pane could show

Table 2.1: Adaptation of intuitive NUI principles for speech interfaces

	Principle	*Application to speech interface*
	Encouraging epistemic actions and exploratory behaviour	
1	Handle different abilities	Command reminders and history.
2	Alternative routes to functionality, and redundancy	Interactivity and "command line"; explanatory words; illustrative icons.
3	Robust feeling interaction	Preview and confirmation; command history for "undo".
	Intuitivity	
4	Conventions and metaphors	Recognise popular ASR commands, and consider using certain physical metaphors.
5	Use existing skills	Use prior knowledge of conventional interfaces.
6	Show current state and available actions	Indicate progress on command processing, and show only appropriate command reminders.
7	Immediate feedback for all actions	The user must know that they have been heard, and how much of a command has been understood.
8	Affordances	Concept of *sayable*[a] in displayed words.
9	Reflect user's mental model	Application-specific.
	Context	
10	Reflect task rather than technology	Permit use of the software in a way that suits the overall task.
11	Environment	Consider both physical and social environment in design.

[a]The concept of *sayable* is the speech controlled visual interface equivalent of "clickable" in GUIs.

completed commands, thus allowing novice users to learn commands or parts thereof. Experienced users who want to work more quickly should be able to issue multiple commands in a single utterance. (See Section 2.5.2 for a fuller description of this functionality.) Novice users may need to be shielded from the size and complexity of the content language (just as with the command language). To address this requirement, either only the most popular words should be shown by default, or a visual device could be employed to make the popular ones more noticeable (for example display order or emphasis style).

2. *Allow functionality to be invoked in different ways.* Because it may be challenging for novice users to issue an entire command in a single utterance, they may want to build up commands interactively in stages. It should be possible for experienced users to customise commands, perhaps changing specific words to ones that are less likely to be misrecognised given the computing environment, or create commands that replace a frequently used phrase with a single word (Fateman, 2013). Where offered, the facility to create new commands is often appreciated by users of speech interfaces. Controls should default to the display of text (with optional pictorial icons for users who have a preference for those), given that words are central to a speech interface. Explanatory words (that are ignored by the parser) should be included in any command reminders. (If this approach is taken, these "decorative" words should be indicated as being optional, so that as the user grows more proficient, they can drop their use. Also, to help the user learn which words truly are necessary in a command, those should be the only ones shown in the command history.)

   If Incremental Speech Recognition (ISR) is available, command sentences should be shown building up as the user speaks. In any case, sentences should be constructed as utterances are spoken. Where parts are missing, these could be selected by the user and the "holes" filled in[15].

3. *Users should not be afraid of making mistakes.* Bearing in mind the preference for brief utterances from the user, previews should appear at the same time as requests for confirmation. If a large or destructive change is made too frequently, such behaviour may become irritating to the user, so as well as taking care over the decision on whether to show such a dialogue for each change, the option to suppress confirmations may be given to the user, perhaps on a case by case basis,

[15]Note, these holes would correspond to those in the templates and discussed in the literature about incremental parsing dealing with programming languages (Petrone, 1995; Cook and Welsh, 2001).

either via configuration or by offering the choice to suppress further confirmations of this type. It should also be possible to use the command history as a means of rolling back changes. Because the effect of change rollback using the history may be difficult to predict, previews of major rollbacks should be offered, or alternatively the user should be able to "undo" the rollback, suggesting the command history display should not be cleared as soon as the changes are rolled back. The handling of syntax errors in commands should minimise the amount of additional user input required. In the case of input being incomplete, the facility for a command to be spread over more than one utterance will address this issue.

4. *Follow established conventions, and use appropriate metaphors.* Conventions already used by ASR software such as "scratch that" should be followed. Reference to the work of Jetter et al. (2014)[16] on conceptual blends and Hurtienne and Israel (2007) on image schemas may help in this.

5. *Allow users to exercise existing skills.* In addition to facilitating learning to use the software, this may boost the confidence of the user. As well as making use of prior knowledge of interface conventions, vocabulary customisation would allow people to use terms specific to their area of knowledge when using the software.

6. *Always indicate the current state and available actions.* A status pane would enable the user to distinguish between situations that may easily be confused, for example providing missing information for a new command or editing a command from history. Where state is normally indicated using the appearance of the pointer, an alternative such as a status bar could be used. Display only relevant command reminders as being *sayable* (see principle 8 below for explanation of this term).

7. *Give appropriate feedback for all user actions.* To make up for the lack of haptic feedback in a speech interface, the user needs to be notified that their input has been detected even though there is a possible delay in its processing (Wigdor and Wixon, 2011; p. 45). This will be particularly important if technologies are being used that will not begin to process the token stream until the user has completed their utterance. In these circumstances, in addition to the feedback usually provided by ASR software, it may be helpful to indicate progress of the processing of the input.

[16]For example, metaphors on two conceptual blends, even if one of the concepts involved is unfamiliar, have a greater probability of success than those that are based on several familar concepts. (Jetter et al., 2014)

8. *Clear affordances.* This relates to both aspects of affordance: how a control may be activated, and what its function is. As well as indicating the function of a control without requiring the user to invoke its tooltip, adding text to it can indicate what needs to be said to activate the control. This gives rise to the concept of *sayable* – the speech controlled visual interface equivalent of "clickable" in GUIs. The words required to invoke the command should be used as the label of a command control, with optional extra brief information in a lighter emphasis style, while other controls, such as those that indicate objects to be manipulated, should have appropriately named labels. This follows the approach already employed by ASR software for form-filling and web browsing.

9. *Compatibility with the user's mental model.* The way in which objects are presented should enable the user to understand their structure and purpose. For example, if the user has to choose from one of several possible parses of a dictated mathematical expression (and not the result of probabilistic predictions of what they have said), they should be informed that this is their origin.

10. *Reflect the nature of the task rather than the technology.* The software should work with whatever combination of available modalities the user wants to employ, for example move a pointer using the mouse, but then speak instead of click to activate the mouse button. Because a mixture of typing and speech may be used, the user should be able to type in a box as well as utter the words on (or click on) command buttons.

11. *Work within the environment of the user.* Not only should it be possible to use the software with a subset of the modalities provided (for example without speech control in a noisy environment), but the social environment also needs to be considered when choosing appearance and vocabulary. For example, professional physicists and Economics undergraduates are unlikely to both be best served by an identical interface.

2.7 Conclusion

Applying the general principles of intuitive natural user interfaces to the modality of speech control, and considering language-specific issues, provides a list of principles to be followed in designing a new interface for the *TalkMaths* system, and similar speech-driven structured content editing systems.

The rest of this thesis addresses the requirement of providing appropriate feedback in a timely fashion (as per INUI principle 7): as mathematical expressions or other content is created or edited, it needs to be displayed in a meaningful way without long delays. For this, an effective incremental parsing algorithm is required.

Chapter 3

Review of Literature on Incremental Parsing

3.1 Incremental Parsing

The original motivation for the development of incremental parsing approaches was to enable timely feedback on syntactical correctness to be provided to programmers when they compiled their work (Ghezzi and Mandrioli, 1979; 1980). Connected to this idea is the concept of "laziness" – meaning that if only a portion of the output of a parse or compilation is visible to the user then only the code related to this portion needs to be processed (Heering et al., 1994) – which is useful in the context of programming languages because of the formatting (such as typeface or colour) used to signify parts of the grammar in an editor screen (Bernardy, 2009).

The usual approach to incremental parsing is in the context of LL and LR grammars[1], where it is accepted that for the change xyz to $xy'z$, the subtree for x may be left unchanged, and so the goal is to identify a minimal substring of z to be reparsed. This is unlike incremental parsing with OP grammars, in which the entire tree is subject to change, depending on the the removal or insertion of particular operators.

The work of Yeh and Kastens (1988) is based on the idea of recording information with every token, that can be used to reconstruct the state of an LR(1) parser at the moment a token is going to be shifted. If the reconstructed states after parsing xy and xy' are the same, then z would not have to be reparsed. The method is storage

[1]For definitions of LL and LR grammars, see Aho (1972; p336) and Aho (1972; p372) respectively.

intensive and does not seem to have been taken any further.

Earley and Caizergues (1972) proposed the use of a "skeleton" structure that records scope information (and has some correspondence to a parse tree), that can be used to determine what parts of source code need to be re-scanned as a result of a change. Many types of edit operation (line insertion, deletion or change to a different type) require the entire program to be reprocessed, so the extent to which their stated aim – to make recompilation effort "proportional to the size of the change" – will vary according to the structure of the program (Earley and Caizergues, 1972). Although this method placed restrictions on grammar (for example, constructs such as loops and *if* statements need to be designed as "bracket structures"), it was hoped that the overall approach could be extended (Earley and Caizergues, 1972). However, this does not seem to have happened.

Ghezzi and Mandrioli (1979) presented an incremental parser that is not restricted in the kinds of modification allowed, nor does it depend on previously saved parse states for its operation. Properties of LR(k) and RL(k)[2] grammars are used to argue that certain threading (the use of additional pointers to related nodes) in the parse tree may be employed by the parsing algorithm to allow reuse of nodes on either side of the modification when applied to LR $\wedge$ RL languages. It is also suggested that this same approach could be applied to other languages displaying similar "symmetrical" properties (Ghezzi and Mandrioli, 1979). Given that OP languages form a subset of their LR $\wedge$ RL counterparts (Ghezzi and Mandrioli (1977) as cited by Ghezzi and Mandrioli (1979)), they put forward the suggestion that the method of construction of LR tables could be used by parsers for related grammars. Their description of techniques to speed up parsing for LR(0) grammars (Ghezzi and Mandrioli, 1980) removes this requirement for symmetry, and they also suggest these methods could be modified for application to LL grammars[3] (Ghezzi and Mandrioli, 1980). Barenghi et al. (2013) put forward a parallel OP parsing algorithm that makes use of a property of OP grammars in Fischer normal form[4], which they call "local parsability" (Barenghi et al., 2013). The main point is that an input string of a language conforming to such a grammar may be split into substrings at arbitrary points for parallel parsing and then recombined, but an additional suggestion made is the application of the local parsability property for incremental parsing (Barenghi et al., 2013).

[2]A grammar G is RL(k) if the grammar formed by reversing the right-hand sides of G's productions is LR(k) (Ghezzi and Mandrioli, 1979).

[3]See Aho (1972; p336) for definition of LL grammars.

[4]Defined by Reghizzi and Mandrioli (2012).

Larchêveque (1995) presented an approach for LALR(1) (Look Ahead LR) incremental parsing that uses threaded concrete parse trees[5]. For a modified string $xy'z$ (based on xyz), the incremental parse begins at the node representing the last token of x, finds the point in the tree section for $y'z$ where an alternative left hand side of a production is found (so the rule that was used in the original parse no longer applies), or until the first symbol of $y'z$ is placed on the parsing stack. At this point it parses "exhaustively" (Larchêveque, 1995). A method is introduced to minimise the rest of this process by considering where a node may be found that is an ancestor of both the new subtrees being produced and the original nodes representing y (so in effect, its parent's right-hand side siblings may be left unaltered), and suggestions were made on how this process may be optimised by reusing subtrees for non-terminals (Larchêveque, 1995).

Wegman (1980) suggested that the use of additional (to the parse tree) balanced trees[6] of pointers to terminal nodes can speed up incremental parsing due to insertion of strings, provided the order of appearance of terminals is not changed.

Jalili and Gallier (1982) determined what parts of a parse tree for an LR(1) grammar are expected to change as a result of a change to a particular node n, these being either the node m that appears directly to the right of n, or a descendant of m. Ballance et al. (1988; 1992) developed a version of this approach that uses two levels of grammar to describe a language and retains subtrees for reuse in order to improve efficiency, at the cost of holding much additional information within the implementation environment.

Ferro and Dion (1994) also suggested reusing structures from the initial parse but, in this case, using dynamic programming methods to allow recovery from earlier states rather than parse trees. (Their parser creates chains of grammar rules rather than parse trees, and handles ambiguities using AND/OR graphs (Ferro and Dion, 1994).) Wagner (1997) proposed an IGLR (incremental generalised LR) parsing algorithm that was intended to save space by making use of a parse DAG (Directed Acyclic Graph), and to avoid unnecessary reparsing by allowing existing unmodified subtrees to become part of the input stream to the system while applying certain rules to decide whether or not they need to be reparsed based on whether they contain non-deterministic sections (Wagner, 1997). This method is very much concerned with handling of language ambiguities, as the context is incremental compilation. Wagner

[5]A concrete parse tree includes detail for every step in the parse derivations.

[6]A tree is balanced if no subtree within it is taller than another subtree at the same level by more than one node.

and Graham (1998) concentrated more on the parsing aspect, refining the conditions for subtree reuse (from the input stream) to allow this to occur more frequently, and identifying where "top-down reuse" (Wagner and Graham, 1998) may occur, suggesting that a combination of these approaches will yield optimal results for efficiency. Although system state information is used, this is only in the context of the incremental parse (and no state information from the initial parse is held in the tree). The parsing algorithm includes elements from the approaches of Larchêveque (1995) and Jalili and Gallier (1982); one particularly interesting aspect is their handling of sequences of statements, which uses a trick described by Gafter (1990) to allow the tree structure for such a list to be non-deterministic. To reduce the height of the trees that represent the sequences of statements in the default manner (strictly right-descending or left-descending), rather than specify a list of statements in a strictly left or right recursive manner (e.g. $statement\_list$ ::= $statement$ $|statement\_list$ **;** $statement$) it is given by $statement\_list$ ::= $statement$ **;** $|statement\_list$ $statement\_list$ (Gafter, 1990). As part of the parse, a tree for such a repetitive structure is built as a balanced tree; the only drawback of this approach is that it requires a modification to the original grammar (Wagner and Graham, 1998).

Yang (1994) also based his LR(1) incremental parsing algorithm on reuse of subtrees of concrete parse trees, but relaxes the conditions needed for a subtree to be reused (Yang, 1994). Prior to that, Yang (1993) described an incremental LL(1) parsing algorithm that uses a "break-point table" (Yang, 1993) derived from the grammar to determine where subtrees may be reused, along with the idea of a stack of subtrees.

More recently, Sijm (2019) built on the idea of Wagner (1997) for scannerless incremental parsing. Based on the difference between the old and new token streams (where the tokens are individual characters), nodes or subtrees of the parse trees are removed or inserted as required. New subtrees are consolidated where possible, and existing trees with deleted nodes are checked for validity and reparsed where necessary. This algorithm does not, however, reuse as much of the original parse tree as the Wagner (1997) algorithm (Sijm, 2019).

An alternative approach was suggested by Rekers and Koorn (1991) that attempts to determine whether an expression in a small subtree containing a modification may form a substring in the language and, if not, retries this substring parse with progressively larger subtrees. This does result in duplication of work (Rekers, 1992).

Murching et al. (1990) proposed a method for recursive descent incremental parsing, that works on a concrete parse tree, determining which parts of $y'z$ (in the tree for altered input stream $xy'z$ of which the original was xyz) are liable to change. The parse tree created during the initial parse is amended during the incremental parsing process as required. Lindén (1994) presented a version of this that allows for early completion of the incremental parse if certain conditions are met, and attempts to improve handling of some types of production. Kahrs (1979) also presented a top-down incremental parser that starts reparsing at the beginning of the semantic block containing the modified line, and stops at the same level if certain criteria are met. Rather than go to the beginning of the block containing the change, the algorithm presented by Shilling (1993) identifies the nodes in the tree that represent the boundaries of the change, prepares an appropriate section of the tree and reparses the modified region incrementally before resolving any issues caused by this parse. Although Shilling's idea concerning an "editing focus" (that may consist of the token next to the cursor, a subtree or a contiguous string of tokens) is of interest (Shilling, 1993), the incremental parsing process seems rather complex.

Li (1996) employed an augmented LL parse table[7] that is used in determining whether old and new nodes (in the y part of xyz and y' part of $xy'z$) will match, indicating opportunity for subtree reuse. This, along with variations on the reuse possibility and a threaded concrete parse tree, is used to maximise reuse of subtrees of the z section of the string when parsing the y' string. Li suggested this approach is particularly suited for languages with a preponderance of structures (blocks and lists) because of the ease with which their trees may be reused (Li, 1996).

Schwartz et al. (1984) built on an approach introduced by Morris and Schwartz (1981) for LL(1) grammars which involves the use of a sequence of parse trees corresponding to adjacent sections of code, that may be joined or split as necessary.[8] The techniques are used in the context of syntax-directed editors, that use text entry[9] but permit the user to place the cursor before or at any syntax error. The parse trees are built up using the Magpie tool (Schwartz et al., 1984) as the user creates the code, rather than enforcing the use of templates. Degano et al. (1988) propose an editor that is syntax-directed in what the user may do at a high level – for example they would need to specify their

[7]For predictions it included a "distance" (Li, 1996) – minimum number of derivations (steps required) – from non-terminals to terminals.

[8]Here, "joining" does not refer the same operation as in Heeman (1990).

[9]This is unlike the strictly syntax-directed editors such as the one developed by Medina-Mora and Feiler (1981) which requires the user to manipulate the tree directly through code templates, thus obviating any need for re-parsing, or hybrids such as the Cornell Program Synthesizer (Teitelbaum and Reps, 1981).

wish to insert a construct such as a *while* loop, or substitute one construct for another (subject to certain restrictions) – but once they are inside the construct, editing may be performed in text mode. The claim they made is that such amendments would not require reparsing of other parts of the code or access to an original parse stack (Degano et al., 1988). This is achieved by using subtables (one for each construct that may be swapped for another) additional to the LALR(1) parse table – a technique they call "Jump-Shift-Reduce" parsing (Degano et al., 1988). Although an editor with such restrictions is unlikely to provide an acceptable user experience, this kind of mode may be particularly useful as an option for programming by speech, allowing the user to switch between this and text mode as required (as is offered by the PSG system (Bahlke and Snelting, 1986)).

Dubroy and Warth (2017) also propose using data structures created by the initial parse, this time by maintaining the memo table used by packrat parsing (Ford, 2002), updating it in response to the user's edit actions. This does incur considerable cost in space.

Petrone (1995) takes an alternative approach to incremental parsing for hybrid editing (syntax-directed and text), in that he extends the underlying language to include "placeholders" (Petrone, 1995), that are similar to, but not equivalent to, nonterminals in its grammar, and could be thought of as a version of missing tree nodes (see Figure 4.1b) but representing a particular element of the language. (For example an expression in the language of such a grammar may resemble "**while** *condition-placeholder* **do** *statement-block-placeholder*".) Rather than use a specialised incremental parsing algorithm, a standard LR parser for the extended language may be used to create the subtree for the new text (y' in $xy'z$), and with some extra processing (which interestingly does not require state data to be recorded in the tree), the new tree may be inserted if certain conditions are valid (Petrone, 1995). Diekmann and Tratt (2013) presented a variant on the placeholder idea in that a token may represent any kind of object (including a parse tree that uses a different grammar, in which case it is called a 'language box'). Although the editor appears to be textual (from what is displayed to the user), the text is in fact a representation of the complete (concrete) parse tree that is updated using the incremental parser developed by Wagner and Graham (1998). They later refined the idea to permit the user to create language boxes without specifying the change in language, using heuristics to recognise correctly the boundaries of the embedded language in most cases (Diekmann and Tratt, 2019). Cook and Welsh (2001) offer a variation on the placeholder in their incremental parsing algorithm, which is written with error-tolerance in mind.[10] In the case of an error, where a certain reduction (replacement of a sequence of tokens with a non-terminal) has been predicted, its placeholder will be placed in the tree. Like other algorithms, it allows subtrees to be placed on an input stack for reparsing, and completes when a matching condition is met, but this algorithm also uses the stack for prediction of reductions and error handling (Cook and Welsh, 2001).

Of most interest in the context of this project are approaches involving OP grammars (covered by Ghezzi and Mandrioli (1979) and Barenghi et al. (2013)), and the notion of "placeholders" similar to those described by Petrone (1995) for missing parts of a construction (although Petrone does use them in a completely different way by extending the grammar in question to include specific placeholder terminals).

3.2 Incremental Operator Precedence (OP) Parsing

Before describing work on incremental OP parsing, this section explores the use of OP grammars to represent constructs not traditionally thought of as being modelled by OP grammars.

3.2.1 Operator Precedence Grammars to Represent Programming Constructs

Soiffer (1991) suggested the use of "overlays" (a kind of template for valid expressions) to model programming constructs such as conditionals and loops as special types of brackets. These are described in more detail in Section 3.2.4.

Aasa (1995) modelled them using "distfix" (Aasa, 1995) operators, and used their associated precedences[11] to resolve ambiguities in the programming language. For example, construct "**if** E **then** E **else** E" where E stands for any expression would be modelled by a *prefix distfix* operator with production $E \rightarrow$ **if** E **then** E **else** E, as opposed to a

[10]This is due to the editing environment which, by parsing incrementally between keystrokes, will indicate an erroneous input until typing is finished.

[11]Note, Aasa (1995) defined operator precedence (OP) in the opposite way to others; this thesis follows the usual convention: that for operators a and b, $a \lessdot b$ means that a yields to operator b, for example $+ \lessdot \times$.

closed distfix operator that contains both opening and closing terminals[12] (Aasa, 1995). (The bracketing style behaviour of the operators is of course as required, so one could think of the Es as being implicitly enclosed in brackets.) These operators have left and right precedence, so that sentences generated by such a grammar may be parsed using an OP parser (Aasa, 1995). Unlike the overlays described by Soiffer (1991), this scheme requires that every operator be identified uniquely, so the **if**, **then** and **else** operators from the above example would have different identifiers from those used in a conditional without an *else* clause. This would complicate incremental parsing, as an edit action could change an operator from one type to another without making any change to the operator itself, thus necessitating further lexical analysis of enclosing structures.

Danielsson and Norell (2011) relaxed the stipulation that every operator be identified uniquely, with the consequence that the grammar becomes ambiguous. Their way of dealing with this would be to reject ambiguous parses. An obvious drawback of having many distinct operators (as suggested by Aasa (1995)) would be a very large OP table. An alternative to this would be to represent relevant precedence relations only, using DAG (with each node containing a finite set of operators), although if shared identifiers are allowed, ambiguities would not be avoided (Danielsson and Norell, 2011). It would seem that the cost of this simpler scheme would be that it is not as easily handled by OP parsing as the scheme of Aasa (1995), but their ideas may be worth exploring, particularly when one considers that a seemingly local modification to program text parsed with Aasa's style of grammar may result in replacement of one distfix operator with another, that spans a large section of the original code.

The algorithm developed in this project uses the structures employed by Attanayake (2014), known as *templates*. These are tree nodes that behave as the bracket or function nodes described by LaLonde and des Rivieres (1981), but in the case of non-bracket structures, rather than being recognised by the parser as a function name followed by a specific number of arguments, they are recognised during lexical analysis by their delimiters, including start and end words (Attanayake, 2014). It is these delimiters that map onto individual operators when describing distfix operators: for example, a template for a definite integral may be recognised by delimiters “integral from”, “to”, “of”, “end integral”, and correspond to a distfix operator with four parts. Because they act as purely bracketting structures, the problem of representing precedence is

[12]The opening and closing terminals of a distfix operator are the ones that appear on the edges of the right-hand side of the production. For example, in $E \to d_1\ E\ d_2\ E\ d_3$ where the d_n are delimiters, d_1 and d_3 are the opening and closing terminals respectively.

avoided.

3.2.2 Lalonde & des Rivieres – Separating Operator Precedence Handling from the Initial Parse Process

LaLonde and des Rivieres (1981) tackled the issue of parsing languages that contain elements of operator precedence, such as a programming language that includes mathematical expressions. They present a linear-time algorithm that allows parsing to be modularised into an initial less complicated parse according to the underlying grammar without concern for OP, followed by any necessary rearrangement of the syntax tree according to operator precedence (LaLonde and des Rivieres, 1981).

An interesting feature of this work is that syntax trees may contain *function nodes* that, as far as operator precedence is concerned, may be treated as if they were leaf nodes. Using this approach, function nodes may be used to represent brackets as well as other functions (LaLonde and des Rivieres, 1981). All binary operator nodes have a left and right child, while unary operators have a left child *or* a right child, the direction indicating whether it is a prefix or postfix operator.

The algorithm is run on a tree that has already been constructed without concern for operator precedence, and rearranges it using transformations on its nodes. When complete, the tree is correct in terms of operator precedence. Working on the assumption that the tree to be transformed is either left-sided or right-sided (left-sided trees are expanded on their left side, with leaf nodes as right children; right-sided trees are their mirror image), the tree may be traversed in a top-down manner, with a single transformation being used to rearrange nodes that do not follow the hierarchical order specified by the OP scheme (LaLonde and des Rivieres, 1981). If the tree to be processed is right-sided, this transformation (called LEFT-SUBORDINATE) will rearrange adjacent operator nodes as depicted in Figure A.2 in Appendix A. (The mirror-image of this operation for use on left-sided trees is RIGHT-SUBORDINATE.)

3.2.3 Kaiser & Kant – Updating the Syntax Tree in Response to Edits

Kaiser and Kant (1985) also presented an idea that obviates the need to modify the initial parser to handle operator precedence, but this time in the context of editing expressions in hybrid syntax-directed or textual-based editors, and in which the tree is updated in response to what the user typed in the linear representation of the expression (Kaiser and Kant, 1985). Their paper concentrates on simple mathematical expressions that conform to a strict OP grammar, but the method is intended to be generalised to other languages.

Their trees are standard syntax trees for expressions, with special "meta" and "empty operator" nodes used to denote missing leaves and operators respectively if the expression is incomplete (Kaiser and Kant, 1985). Unlike LaLonde and des Rivieres (1981), complete pairs of brackets are represented by bracket nodes, but in addition to these, single bracket nodes (or) may be used where only one bracket is present in the expression. All operators are binary infix (Kaiser and Kant, 1985). Operator precedence (and hence associativity too) is defined in terms of left and right precedence (Kaiser and Kant, 1985), rather than using operator precedence comparison relations.

Tree operations that work at the node level are presented, that respond to a number of user actions such as insertion and deletion. In their original form[13], the operations begin at the insertion point in the tree, after which Lalonde and des Rivieres' LEFT-SUBORDINATE and RIGHT-SUBORDINATE operations (LaLonde and des Rivieres, 1981) (named *twiddle* here) are applied upwards or downwards in the tree to rearrange it as required.

Unmatched parentheses of appropriate type are matched during the tree rearrangement process. Depending on circumstances, a singleton bracket encountering a matching singleton would cause the rearrangement to halt after combination, or "steal" (Kaiser and Kant, 1985) its counterpart from a complete bracket pair, after which rearrangement continues.

Suggested extensions to their algorithm include provision for unary operators and the handling of functions and other programmatic constructs (Kaiser and Kant, 1985). A function would be recognised as an identifier followed immediately by parentheses, which would hold the function's argument. Multiple arguments would be delimited by using a comma operator (Kaiser and Kant, 1985). Kaiser and Kant were slightly vague in their description of possible provision for handling language keywords (special operators, and entities similar to parentheses); a preferable approach may be to combine the function nodes of LaLonde and des Rivieres (1981) with Kaiser and Kant's comma operators, to produce something like the templates used by Attanayake (2014).

[13]Their algorithm is optimised, and includes operations to remove excess empty/meta nodes.

The approach of Kaiser and Kant (1985) has the advantage of handling incomplete expressions, and offers an alternative way of handling brackets. A drawback is that the approach is tightly integrated into the editing environment; for example, most transformations begin at the leaf where a small incremental change has been made.

3.2.4 Soiffer – Language Constructs as "Special" Operators

Whereas Kaiser and Kant (1985) represented unmatched brackets as single bracket operators, Soiffer (1991) extended this idea to programming language constructs, using what he called "overlays with precedence", in which the the words or symbols are referred to as "delimiters". Tree rearrangement following insertion of one of these is similar to Kaiser and Kant's algorithm, in that the unmatched item moves up the tree to be matched with an appropriate unmatched delimiter, or to "steal" a matching delimiter from a matched pair.

All of these delimiters have left or right precedence and are designated *prefix*, *infix* or otherwise (see Soiffer (1991)) when encountered in their unmatched form. What differentiates these delimiters from the brackets encountered so far is that they are also given zero or more possibly matching delimiters. For example, in the case of the *if* statement with allowable form **if** ... **then** ... { **else** ... }, **if** may match **then** to form **if-then**, and **if-then** may match **else**; however, **then** may also match **else**, giving **then-else**, which would be matched on its left by **if**. Although it appears inside the construct (that is, it will never act as a closing delimiter), **else** will never match anything on its right hand side (so in bracketing terms it is similar to a closing bracket); similarly, because the *else* clause is optional, **then** may also be used as a closing delimiter (Soiffer, 1991).

Soiffer (1991) stated that his approach is faster than that of Heeman (1990), partly because he avoided the linear searches involved in matching brackets in the latter's algorithm. (An alternative means of addressing this issue will be presented in Chapter 4.)

It is in this way that Soiffer (1991) attempted to extend the algorithms of Kaiser and Kant (1985) to handle a wider range of syntactical structures by identifying possible left and right matches for language keywords (Soiffer, 1991).

He also optimised the algorithm presented by Jalili and Gallier (1982) for the OP grammars handled by Computer Algebra Systems (Soiffer, 1991), and suggested the use of DAGs instead of trees to improve performance (in terms of storage), where a sub-expression identical to another would be represented by a pointer to the first one encountered.

3.2.5 Heeman – Tree-level Operations

Whereas the other approaches describe modifying or inserting nodes in a tree, followed by any necessary rearrangement using node level operations, the approach of Heeman (1990) is to define basic high level operations that act on entire trees, that may be divided and combined to implement the effect of a change in the token stream that the tree represents. For example, to insert a node, Heeman would *split* the tree at the insertion point, *merge* the new node with one of the results, and then *merge* that with the other part of the originally split tree.

In this context a tree (called "expression tree" (Heeman, 1990)) is similar to an abstract syntax tree (AST) but with leaf nodes for operators (so in this way it resembles a concrete parse tree), and is correct in terms of operator precedence (Heeman, 1990). See Figure 3.1 for an illustration.[14] All operators are binary; associativity is left or right, and precedence is expressed as a numeric value[15].

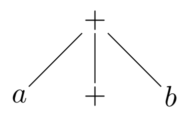


Figure 3.1: Tree notation of Heeman (1990): tree for $a + b$

Brackets are represented using an operator that has three children (Heeman, 1990) (as shown in Figure 3.2). A missing right or left bracket is denoted by replacing the (or) with a "□". Other styles of bracketing characters are permitted, such as "[ ]"and "{}".

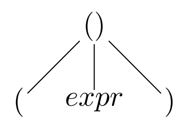


Figure 3.2: Representation of parentheses by Heeman (1990)

Heeman (1990) presented two operations that may be used to split and merge trees, to model splitting and concatenation of token strings. The editor may use these top

[14]Unlike those presented by Heeman, the algorithms in the next chapter do not make use of this particular tree notation.

[15]The higher the value, the higher the precedence of the token. Identifiers have a precedence of ∞.

level operations to reflect whatever changes have been made by the user. The language for the expressions is illustrated using the usual binary mathematical operators, plus a default concatenation operator that permits trees to be merged that do not have an appropriately placed empty node. (For example, whereas expressions "$a + b$" and "$\times d$" could be merged naturally because of the missing operand before the $\times$, unless a merge of trees for expressions "$a + b$" and "$c \times d$" were to produce an error condition (because no operator has been supplied to place between b and c), b and c have to be concatenated to produce the tree for "$a + bc \times d$".)

The basic version of the *merge* algorithm combines two trees from the top down, according to operator precedence; splitting (implemented by the *tear* algorithm) divides the subtree at the split node into two in a top-down fashion, then recombines each into the appropriate parts of the main tree (Heeman, 1990). Having set up the basic versions of these operations, Heeman modified them to deal with brackets: there is no major change in approach in the case of *tear*, but when merging, the sections of the trees representing the right substring and left substring of the left and right trees respectively are searched for matching incomplete opening bracket nodes and incomplete closing bracket nodes respectively. The textually innermost incomplete brackets are combined, either to form complete brackets if they are of the same type, or staggered, the closing bracket placed immediately below the opening one, if they are not (for example a single "**{**" and a single "]") (Heeman, 1990). This is potentially the most time-consuming part of the algorithm, which is improved upon by the algorithm presented in Chapter 4 by identifying all appropriate unmatched bracketing structures in two passes, one for each tree to be merged.

As well as handling incomplete expressions, this approach is of interest because it works with complete trees and would inherently be more loosely coupled with any editor. As stated by Heeman (1990), the algorithms he presented are written for clarity rather than efficiency; possible suggested improvements include the use of iteration rather than recursion in the merging algorithm, and simple node manipulations that will provide the functionality of the merges that are used in the splitting process (Heeman, 1990).

The following extensions were also suggested (Heeman, 1990):

1. Handling of operators taking numbers of arguments other than 2, including single-argument prefix and postfix operators. Although Heeman (1990) stated that the C ternary conditional expression operator has been implemented in the INFORM editor, the next chapter will generalise the algorithm explicitly to handle any pre-

defined (for the operator) number of operands.

2. Application to programming languages. Heeman (1990) suggested that his approach would not be appropriate for constructs that are not readily regarded as expressions, for example statement lists, but it is not clear why Heeman made this assertion.

3. Handling of operators that are ambiguous in terms of arity[16].

4. Handling of functions and arrays – that is, an identifier with a bracketed single argument, which may itself have many components.

Ideas 1, 3 and 4 have been addressed by Attanayake (2014) in terms of data structures and batch parsing, but not for incremental parsing. The idea of the function aspect of item 4 naturally relates to the treatment of brackets by LaLonde and des Rivieres (1981). A primary objective of this project is to build on Heeman's work, improving the efficiency of the search for potentially matching brackets in the *merge* algorithm and investigating the degree to which extension 2 above may be achieved.

3.3 Other References to Incremental Parsing

The literature available on the type of incremental parsing treated in this thesis is scattered fairly sparsely over a number of decades. Other work on incremental parsing shares the motivation of the earliest work on the subject – that of providing timely feedback on a construct as it is processed – but this time almost exclusively in the context of natural language processing (NLP), where the term 'incremental parsing' refers to the updating of a parse tree in response to tokens as they are appended to the token stream. The exception to this is the two-dimensional hand-written mathematics recognition approach of MacLean and Labahn (2013), which also responds to deletion of pen strokes. Most of the recent NLP 'incremental parsing' literature is based on dependency parsing (Cross and Huang, 2016; Kato and Matsubara, 2015; Koehn and Menzel, 2014; Huang, 2010), though the use of semantic roles (Konstas and Keller, 2015) and Incremental Combinatory Categorical Grammar (ICCG) (Hefny et al., 2011) are also discussed. In the case of tree-adjoining grammar (TAG) parsers (Kato and Matsubara, 2015; Konstas and Keller, 2015), one can see certain parallels with the

[16] *Arity* refers to the number of operands required by an operator.

approaches described here, namely the notion of trees being treated as part of a token stream and of an operation that may combine them in a non-trivial way.

3.4 Conclusion

This chapter has provided an overview of the literature on incremental parsing, and focussed on the approaches of most interest in this thesis: incremental parsing of operator precedence grammars as described by LaLonde and des Rivieres (1981), Kaiser and Kant (1985), Soiffer (1991) and Heeman (1990).

The remainder of the thesis will build on Heeman's tree-level operations with a view to their application to spoken computer programming languages. The extensions to the algorithms will have to deal with mixfix operators, and incorporate efficiency improvements to the algorithms.

Chapter 4

An Extended Incremental Parsing Algorithm Based on Tree Operations

4.1 Introduction

This chapter presents the main technical contribution of this thesis, based on an extension of the algorithms described by Heeman (1990) in order to handle expressions containing mixfix operators by manipulating the representation of these constructs in the form presented by Attanayake (2014).

The chapter begins by discussing the representation of non-binary operators, before describing the algorithms for carrying out tree operations. The method for identifying incomplete composite nodes is presented, after which the subject of merging composite nodes is tackled. Section 4.3 finishes by addressing the issues of brackets, composite nodes and associativity, and avoiding violation of operator precedence.

In Section 4.4 the algorithm is evaluated from a theoretical standpoint, and its application is compared with use of the batch parser.

This chapter discusses the algorithms in the context of mathematical expressions; their application to computer program code is covered in Chapter 6.

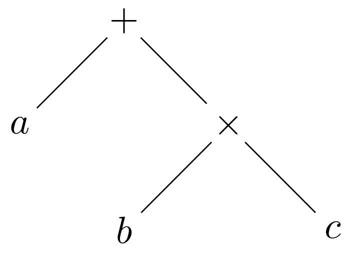
4.2 Notations and Terminology

4.2.1 Trees and Nodes

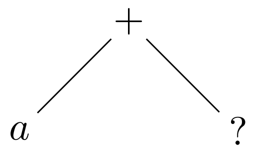
This thesis uses the definition of **abstract syntax trees** (ASTs) as described by Aho et al. (2003; p. 287), as they contain the minimum information needed to describe the language elements of a production of the grammar. ASTs will also be referred to using the shorter forms, **syntax tree** or **tree** (where there is no ambiguity).

An AST may represent an empty node, a leaf node, or any node with children (that themselves may be trees). Typically, operators appear only as non-terminal nodes in trees, while leaf nodes denote operands. A missing operand will be represented by a question mark "?".[1] See Figure 4.1 for examples.

Brackets (such as the ones enclosing this clause) are also referred to as parentheses.



(a) Complete tree: $a + b \times c$



(b) Tree with "missing" node: $a + ?$

Figure 4.1: Example syntax trees

4.2.2 Mixfix Operators

This term describes operators (also known as *distfix* operators (Aasa, 1995)), in which the "holes" between operators act on their content as if they were brackets.[2] The concept of a mixfix operator is the generalisation of operators with respect to arity. As well as binary, prefix and postfix, it includes the concept of operators that may enclose part of an expression – *closed mixfix* or *closed distfix* as described by Aasa (1995), for

[1]This is not simply an empty string – it is a placeholder node that denotes a missing part of the expression.

[2]For example, in a mixfix operator taking form **fraction** *op1* **over** *op2* **end fraction**, **fraction** and **over** would behave additionally as a pair of brackets (opening and closing) around *op1*; likewise, **over** and **end fraction** would behave as a bracket pair.

example a pair of brackets – and any predefined mixture of operators and operands provided they conform to OP grammar rules.

Although programming constructs may be thought of as open as well as closed mixfix operators, in this treatment they are restricted to the closed type, to conform to their modelling by the templates employed by Attanayake (2014).

Where reference is made to a "delimiter", this means any separator that forms part of a composite node, whether it be an internal separator or have an enclosing function. Where a distinction is to be made, they will be referred to as internal delimiters (for separators) and enclosing delimiters, the latter including brackets.

4.3 The Novel Approach to the Algorithm

In this section, Heeman's algorithm is extended to handle mixfix operators, as represented by the structures described by Attanayake (2014). Heeman (1990) states that the transformations are described recursively to aid understanding, and acknowledges that iterative versions would be more efficient. Although this can be viewed as an implementation matter, conversion of the transformations to highly efficient iterative versions is non-trivial, so the loops in the algorithms described here have been designed to be iterative. As observed by Soiffer (1991), in order to improve performance, the issue of the bracket search in the *merge* algorithm needs to be addressed – this being particularly pertinent when the algorithm is used for more substantial documents such as program code for non-trivial programs or tasks. The algorithm presented here includes a solution to this that identifies all enclosing delimiters before the *merge* process begins.

4.3.1 Representation of Non-binary Operators

The representation of a binary operator is well established, with a node to represent the operator with two children, the first (left) child representing the left operand, and the other (right) child representing the right operand. The other kinds of operators are

- unary (prefix or postfix),
- ternary (for example, the conditional operator **? :** found in programming languages),
- grouping or bracketing, and

- lists.

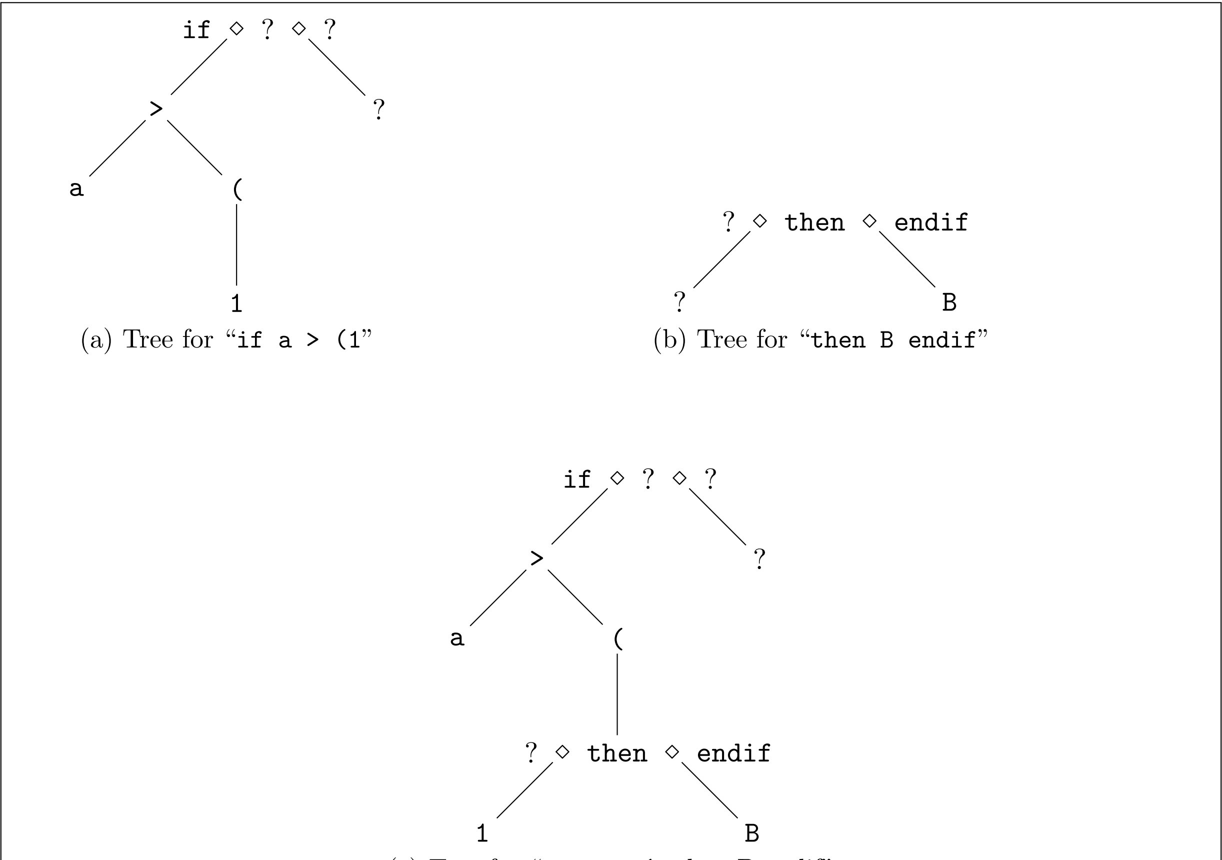
Given that unary operators take precedence over other types, care must be taken not to violate operator precedence when designing the language.

It is difficult to define how the ternary operator may interact with unary or binary operators, except that it is said to have lower precedence than anything but assignment. Given that it is only used in programming situations, in this context the problem of interaction with other true operators is avoided by treating it in the same way as the grouping operators.

Grouping operators cover any special constructs such as programming constructs, mathematical functions and any form of bracket pair. This thesis will use the term employed by Attanayake (2014), *templates*, to refer to this type of construct, and *composite node* or *trunking node* (following the concept of "trunk"[3] described by LaLonde and des Rivieres (1981)) to refer to a node that is of the template or bracket type. A template consists of two delimiters that mark the beginning and end of the structure, with zero or more separators inside. The number of separators for any one particular template is fixed. The question arises on whether grouping operators have precedence that could be used for matching. For example, suppose a simple *if* statement were modelled as a template with delimiters **if**, **then** and **endif**, and a merge were required on fragments "`if a > (1`" and "`then B endif`". With all templates being treated as having equal precedence, an attempt is made to match the "`(`" with the "`then`" (because they are the leftmost and rightmost bracketting constructs respectively), causing the second fragment to appear lower down in the result, within the subtree representing the test clause of the *if* statement. (See Figure 4.2.)) One could argue that it is obvious that the comparison within the first fragment is incomplete, and that the tree representing the result should have a complete *if* statement at its top, with a subtree for "`a > (1`" immediately beneath it. The approach is taken that they should not, as an entire function definition may appear within a pair of round brackets, for example as occurs in JavaScript. This is consistent with the approaches of both Heeman (1990) and Attanayake (2014).

Lists refer to a sequence of similar items, for example program instruction statements or the content of a set extension. They differ from the items between the opening and closing operators in a template in that the number of items in the list is variable and

[3]This is significant in our context because the subtree under the trunk is not subject to being rearranged internally as a result of rearrangements higher up in the tree.



(a) Tree for "`if a > (1`"

(b) Tree for "`then B endif`"

(c) Tree for "`if a > (1` then B endif"

Figure 4.2: Merging trees for "`if a > (1`" and "`then B endif`"

unknown *a priori*. These cannot be represented directly using an operator grammar: for example, a set extension would have to be modelled using a curly bracket pair above a tree of binary comma operators. The modelling of lists is discussed further in Section 6.5.

4.3.2 Tree Operations

As with Heeman (1990), changes in the AST to reflect modifications in the material being represented are achieved using just two operations: splitting and merging.

The basic split operation, *tear* (Heeman, 1990), is generalised to trees that contain unary and grouping operators as well as binary operators. The AST representation presented by Attanayake (2014) is used, which employs a single node to represent a composite mixfix operator, so rather than specify a tree node for the tear point, a token must be used, that will either correspond to a single node, or part of a composite node.

Splitting Trees

The top level operation is described by Algorithm 1. In short, it works approximately as follows.

- Accept as input a tree and the rightmost token (in terms of the linear form of the expression) that is to appear in the original tree after the split.
- Identify the subtree containing the token.
- Remove children to the right of the token, placing them into a new tree.
- Repeat until the root node is reached:
  - Move up one level.
  - Split off nodes to the right of the subtree just processed, into a new tree.
  - Incorporate the split-off tree from the previous iteration into this new tree.
- Return a tuple consisting of what is left of the input tree, along with the latest split-off tree.

Lines 7 and 21 perform actions that should have been included in the original algorithm (Heeman, 1990), as it allowed successive tear operations to produce a tree with an empty root node. For example, tearing a tree for expression "(a)" after the opening bracket, and then tearing the right-hand tree of the result after the "a" would produce a left-hand tree consisting of an empty root node which is effectively a bracket with both sides missing and single child "a". Figure 4.3 illustrates the situation, with 4.3c having an empty root node that may never be removed from the AST. Because this version of the algorithm handles templates with multiple parts, the situation here is not as simple as with empty bracket nodes: in the same way as for the above example, under rare circumstances a sequence of operations may produce a composite node with more than one child but no delimiters at the top. When this arises, all the children of such a node are merged, and the empty composite node replaced with the result of the merge. This is valid because any empty composite node with no delimiters but with more than one child represents the tree structure of the concatenation of the token strings represented by its children.

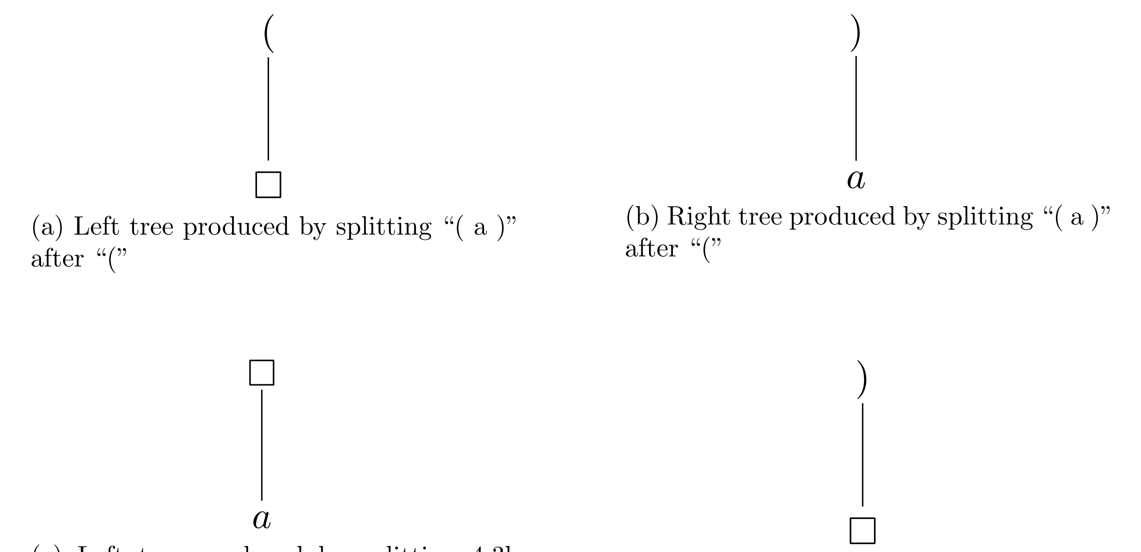
In a similar way, lines 10 and 33 address the case of the default concatenation operator[4] representing the split point. Because this operator is regarded as being required only

[4] In the context of mathematics this is the "invisible" multiplication operator between a and b in ab.

Algorithm 1 $tear(T, k) \to (T_L, T_R)$
Split AST T into two trees T_L and T_R so that the rightmost token of the stream represented by T_L is k.

```
Require: $k$ appears in a node in $T$
 1: set $T_L$ to the node of $T$ containing $k$
 2: if $T_L$ is not the root node then
 3:    define $p$ such that $T_L$ is the $p$th child of its parent
 4: end if
 5: if $T_L$ is composite then
 6:    remove delimiters and children of $T_L$ after $k$, placing them into new tree $T_R$
 7:    replace any of $T_L$, $T_R$ left with no delimiters at the top with its children
 8: else if $T_L$ is binary operator then
 9:    remove second child of $T_L$ as $T_R$
10:    replace $T_L$ with its left child if $T_L$ is the default operator
11: else if $T_L$ is prefix operator then
12:    remove only child of $T_L$ as $T_R$
13: else // postfix operator or identifier
14:    set $T_R$ to empty tree
15: end if
16: while $T_L$ is not root node do
17:    set $T_L$ to its parent
18:    if $T_L$ is composite then
19:       remove delimiters and children of $T_L$ from position $p+1$ onwards, placing them
          into new tree $T'$
20:       replace the $p$th child of $T'$ with $T_R$ and then set $T_R$ to $T'$
21:       replace any of $T_L$, $T_R$ left with no delimiters at the top with its children
22:    else if $T_L$ is binary and $p$ is 0, or $T_L$ is postfix then
23:       remove first child of $T_L$ as $c$
24:       if $T_L$ is the root node then
25:          $T' :=$ duplicate of $T_L$
26:          set $T_L$ to $c$
27:       else
28:          detach $T_L$ from its parent as $T'$
29:          put $c$ in the place previously occupied by $T_L$
30:          set $T_L$ to $c$
31:       end if
32:       place $T_R$ under $T'$ in position 1 and set $T_R$ to $T'$
33:       replace $T_R$ with its right child if $T_R$ is the default operator
34:    end if
35:    if $T_L$ is not the root node then
36:       define $p$ such that $T_L$ is the $p$th child of its parent
37:    end if
38: end while
39: return $(T_L, T_R)$
```

as long as it is needed to avoid violation of the operator grammar, it is removed once it becomes redundant.



(a) Left tree produced by splitting "(a)" after "("

(b) Right tree produced by splitting "(a)" after "("

(c) Left tree produced by splitting 4.3b after "a"

(d) Right tree produced by splitting 4.3b after "a"

Figure 4.3: Empty root node produced by original Heeman (1990) algorithm

The delimiters of composite nodes are numbered 1...n. For example a simple pair of brackets would have delimiter 1 as "(" and delimiter 2 as ")". All such nodes will have children numbered $1, \ldots, (n-1)$, where child 1 will be the part contained between delimiters 1 and 2, child 2 the part between delimiters 2 and 3, and with the numbering of the other children following the same pattern.

At lines 6 and 19, sufficient empty delimiters and children must be inserted into the new tree before those moved from T_L, so that they preserve their original meaning as "arguments" within the template.

Combining Trees

Merging is also based on the approach of Heeman (1990), but with the searches for unclosed or unopened composite nodes (the equivalent of incomplete brackets) using lists of such nodes created at the beginning of the operation in a manner that is more efficient than a linear text search. The top-level algorithm of the *merge* operation is the one that involves combining composite nodes. Although the OP-only aspect of merging can be described simply when using recursion, its non-recursive equivalent does require an equivalent of the “rippling up” of twiddle operations as described by Kaiser and Kant (1985). In brief, the iterative version of merge works in a similar way to the recursive one, except that instead of the recursive calls, “plug points” (the places into which the bottom-most merged subtree will fit) are saved onto a stack. As the subtrees are reassembled into a merged tree, depending on the position and arities of the operators, the twiddle operations (Kaiser and Kant, 1985) may be required, as well as variations of them that work with unary operators. For descriptions of the actions of the various twiddle operations, see Appendix A.

The top-level algorithm for the *merge* operation is described by Algorithm 2, which is given the name *match_merge*, the name used by Heeman (1990). It is an extension of the Heeman (1990) algorithm, which places children of composite nodes into the appropriate gap rather than the single gap provided by bracket pairs. The condition at line 21 of Algorithm 2 refers to this; the topic is discussed more fully in Section 4.3.4. Note that the arrangement of incompatible composite nodes follows the convention adopted explicitly by Attanayake (2014) and implicitly by Heeman (1990) – that they are right associative. (Associativity of composite nodes is discussed more fully in Section 4.3.5.)

The trails of unfinished and unstarted nodes provide “pointers” to incomplete composite nodes. The way in which these are constructed is described in Section 4.3.3.

Heeman’s algorithm name, *opmerge* (Heeman, 1990), is also used for the operation on trees without any unresolved incomplete composite nodes. This is given in Algorithm 3. The main loop describes the recombination of the subtrees into the result, along with any rearrangements needed to preserve OP. It should be noted that it only deals with unary and binary operators. (n-ary operators with $n > 2$ are handled using the templates of Attanayake (2014).)

In the cases of the current left and right nodes being both prefix or both postfix, there is potential for OP to be violated, as described in Section 4.3.6. (See Algorithm fragments 6 and 11 in Appendix B.) It is stated in these fragments how the nodes should be arranged, but the decision on how to deal with the OP violation is left to the implementer. The choice made for this implementation was to enclose the child node in brackets.

Table 4.1 lists the actions taken for the various combinations of the values taken by c_L and c_R in Algorithm 3. The algorithm fragments can be found in Appendix B.

Algorithms 2 and 3, along with Table 4.1 are given on the following pages.

Algorithm 2 *match_merge*$(T_L, T_R) \to T$
Merge ASTs T_L and T_R into a single tree T, preserving validity according to operator precedence.

```
set $u_L$ to $makeTrailOfUnfinished(T_L)$
set $u_R$ to $makeTrailOfUnstarted(T_R)$
while it is possible to pop $l_m$ from $u_L$ or $r_m$ from $u_R$ do
   set $w_R$ to to 0
   if $l_m$ was popped then
      if $l_m$ has 2 delimiters then // simple bracketing structure
         set $w_L$ to 1
      else
         if last delimiter is blank but there is an earlier $k$th non-blank delimiter then
            set $w_L$ to $k$
         else // use default position
            set $w_L$ to position of last child of $l_m$
         end if
      end if
      if $r_m$ was popped then // we want to match these nodes
         if $r_m$ has a parent then
            remove $r_m$ from its parent
         else // $r_m$ is $T_R$
            set $T_R$ to the empty tree
         end if
         if $l_m$ will fit together with $r_m$ then
            merge tokens and children of $r_m$ into those of $l_m$
         else // no match – make one the child of the other
            set $c_A$ to the $w_L$th child of $l_m$, and replace it with an empty subtree
            set $w_R$ to the position left of the first non-blank delimiter of $r_m$
            set $c_B$ to the $w_R$th child of $r_m$
            replace the $w_R$th child of $r_m$ with $opmerge(c_A, c_B)$
            replace the $w_L$th child of $l_m$ with $r_m$
         end if
      else // no potentially matching right bracket found
         set $c$ to the $w_L$th child of $l_m$
         replace the $w_L$th child of $l_m$ with $opmerge(c, T_R)$
         return $T_L$
      end if
   else if $r_m$ found then // no potentially matching left bracket found
      set $c$ to the child to the left of the first non-blank delimiter of $r_m$
      replace this child with $opmerge(T_L, c)$
      return $T_R$
   end if
end while
return $opmerge(T_L, T_R)$
```

Algorithm 3 *opmerge*$(T_L, T_R) \to T$
Merge ASTs T_L and T_R into a single tree T, preserving validity according to OP

```
Require: No unresolved incomplete composite nodes spanning the two trees
if $T_L$ is the empty tree then
    return $T_R$
else if $T_R$ is the empty tree then
    return $T_L$
else
    set $P$ to an empty stack, $c_L$ to $T_L$, $c_R$ to $T_R$, $workingDown$ to true
    while $workingDown$ do
        update $P$ and intermediate tree $s$ according rules laid out in table 4.1
    end while
    while $P$ is not empty do
        pop $(p, k)$ from $P$
        place $s$ under $p$ in position $k$
        if $p$ is prefix and $p \gtrdot s$ then
            if $s$ is binary then
                perform $twiddlePrefixLeft$ on $p$
            else if $s$ is postfix then
                perform $twiddleUnaries$ on $p$
            end if
        else if $p$ is postfix and $p \gtrdot s$ then
            if $s$ is binary then
                perform $twiddlePostfixRight$ on $p$
            else if $s$ is prefix then
                perform $twiddleUnaries$ on $p$
            end if
        else if $p$ is binary then
            if $k = 1$ then
                if $p \gtrdot s$ then
                    if $s$ is binary then
                        perform $twiddleLeft$ on $p$
                    else if $s$ is prefix then
                        perform $twiddleLeftPrefix$ on $p$
                    end if
                end if
            else
                if $p \gtrdot s$ then
                    if $s$ is binary then
                        perform $twiddleRight$ on $p$
                    else if $s$ is postfix then
                        perform $twiddleRightPostfix$ on $p$
                    end if
                end if
            end if
        end if
        set $s$ to $p$
    end while
    return $s$
end if
```

Table 4.1: Actions to take during *opmerge* according to operator type and precedence of the current nodes being visited in the left and right trees by Algorithm 3

		c_R				
		prefix	postfix	binary	id/trunking	null tree
c_L	prefix	$c_LPrefix\_c_RPrefix$	$c_LPrefix\_c_RPostfix$	$c_LPrefix\_c_RBinary$	$c_LPrefix\_c_ROther$	$c_LPrefix\_c_ROther$
	postfix	$c_LPostfix\_c_ROther$	$c_LPostfix\_c_RPostfix$	$c_LPostfix\_c_RBinary$	$c_LPostfix\_c_ROther$	$c_LPostfix\_c_ROther$
	binary	push $(c_L, 2)$ onto P, then set c_L to decoupled right-hand child of c_L	$c_LBinary\_c_RPostfix$	$c_LBinary\_c_RBinary$	push $(c_L, 2)$ onto P, then set c_L to decoupled right-hand child of c_L	set *workingDown* to false, s to c_L
	id/ trunking	set s to a tree with default operator as root and children c_L and c_R	$c_LOther\_c_RPostfix$	$c_LOther\_c_RBinary$	set s to a tree with default operator as root and children c_L and c_R, *workingDown* to false	set *workingDown* to false, s to c_L
	null tree	set *workingDown* to false, s to c_R	N/A			

4.3.3 Locating the Unmatched Composite Nodes in Preparation for *match_merge*

The *match_merge* algorithm published by Heeman (1990) specifies that, for each iteration, incomplete composite nodes should be located in each tree that has no such nodes in its own subtree. Although the way in which this is to be done is not specified, as recognised by Soiffer (1991), it is an aspect that is likely to cause time performance issues. Here, a method is used for locating such nodes before beginning the merge, the time complexity of which is $\mathcal{O}(h)$ where h is the height of the tree, for each tree. The approach is more efficient than a linear search through the yield[5] of a tree because the search only visits nodes along an outer edge. The process creates two stacks of nodes: one of unfinished composite nodes in the left-hand tree and the other of unstarted composite nodes in the right-hand tree. This version of the *match_merge* algorithm simply pops incomplete composite nodes from the stacks until one of the stacks is empty.

The following argument explains why the unfinished node stack construction algorithm will work for the left-hand tree. The situation will be mirrored in the right-hand tree, albeit slightly more simply.

- Any unclosed composite node will reside on the outer right edge of the tree. If that were not the case, there would have to be a subtree that contained somewhere within it an unclosed composite node that would have a right sibling. So the unclosed composite node would be in effect complete, otherwise the parse tree could not have anything that would come logically after that node. (For example, the tree for "$(a \times b + c$" could not have the "+" at the top because the "(" applies to all of the rest of the expression.)
- Because of this, we only need to traverse the right outer edge of the left-hand tree to find unfinished composite nodes.
- The *match_merge* algorithm only combines incomplete composite nodes that have no incomplete composite nodes in their subtrees that are waiting to be matched. (This is by the definition of *match_merge*, as per Heeman (1990).)
- The *match_merge* algorithm always combines subtrees found at the point of the lowest incomplete composite nodes in the hierarchy, moving one of these nodes

[5]The *yield* of a parse tree is the result of "unparsing" it to linear form.

from the left to the right tree or *vice versa*. Because of this bottom-up behaviour, the trails of incomplete nodes on each tree will not be altered by the algorithm (other than by having stack members removed).

- Because of this, the identification of incomplete composite nodes can be completed before the main body of the algorithm.

Although the *match_merge* algorithm does not create nodes of mixed composite types – opening and closing parts are staggered in the same way as in the Heeman (1990) algorithms – when it comes to identifying unfinished opening composite nodes, it turns out that if a closing part of any kind exists underneath it, the opening composite node needs to be treated as closed.[6] The question arises of whether an unfinished composite node could exist, directly below which there is a composite node that is both unstarted and unfinished, below which there is an unstarted composite node with a closing delimiter. Such a situation can in fact occur. Consider three templates A, B and C, with delimiters $d_{a1}, \ldots, d_{ak}$ for A, $d_{b1}, \ldots, d_{bl}$ for B and $d_{c1}, \ldots, d_{cm}$ for C with $k, l, m \in \mathbb{N}$ and take, for example, expression $d_{a1}\ p\ d_{bv}\ q\ d_{cm}$ where $1 < v < l$, and p, q are complete subtrees (not having incomplete composite nodes). In effect it is a closed expression made of three "badly matched" composite nodes. Because all templates are right associative, the conclusion needs to be drawn that $d_{a1} \lessdot d_{bv} \lessdot d_{cm}$, and so the tree shown in Figure 4.4 is built.[7]

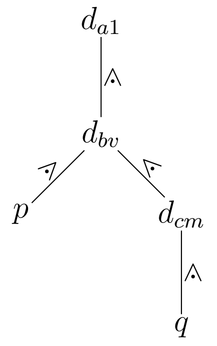


Figure 4.4: Tree for $d_{a1}\ p\ d_{bv}\ q\ d_{cm}$ (with empty nodes and delimiters omitted)

[6]If not, the tree resulting from the merge would not be the same as the tree resulting from parsing the complete expression.

[7]Note, this is a digression from the usual convention that states bracketing operators have precedence $\doteq$ (Aho, 1972). See section 4.3.5 for a longer discussion.

For a concrete example, consider the following situation involving these templates.

- **absolute value** E **end absolute value**
- **integral from** E **to** E **of** E **end integral**
- **log of** E **end log**

Suppose the user has entered the expression `absolute value A to B end log`. As in Figure 4.4, the tree for this expression will have an unclosed composite node for `absolute value` at the top, with an unstarted and unfinished composite node for `to` below that, and an unstarted composite node for `end log` as the right-hand child of the node for `to`. (See Figure 4.5.)

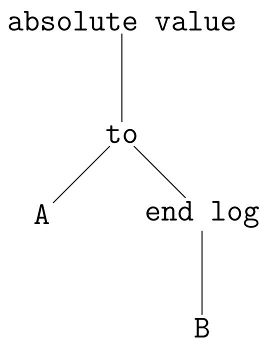


Figure 4.5: Tree for `absolute value A to B end log` (with empty nodes and delimiters omitted)

Because of such a scenario, to check whether or not a badly matched composite node is closed, it is not sufficient to examine just its direct descendant – we have to travel down the right-hand border of the tree to check whether there is a closing delimiter. However, it is already the right-hand border down which we travel while searching for more unclosed nodes, so to avoid the inefficiency of traversing the same section of tree two or more times, the algorithm for *makeTrailOfUnfinished* tracks state (unlike that for *makeTrailOfUnstarted*).

Algorithm 4 describes *makeTrailOfUnfinished* while algorithm 5 describes *makeTrailOfUnstarted*. The latter is simpler because staggered composite nodes have the unstarted node subordinate to their "matched" unfinished node. (When a pair of badly matched composite nodes is discovered, they are merged as illustrated in Figure 4.6. This means that when such a pair arises, the closing half will appear as the last (non-empty) child of the opening half.) Algorithm 4 makes reference to a variable, s, which may take values *searching* (searching for a potentially unfinished composite node), *in_unfinished* (found a potentially unfinished composite node), and *done* (finished the search process).

If an alternative approach to the matching of incomplete composite nodes were to be adopted, for example choosing pairs that maximise the number of successful matches of delimiters belonging to the same template, the validity of using this method to identify incomplete composite nodes would need to be reviewed. Such an approach, while attractive from an error-handling point of view, would necessarily lead to a requirement to reprocess potentially large portions of the document in response to subsequent user edits, thus losing some of the benefits of employing incremental parsing.

4.3.4 Merging Incomplete Composite Nodes

When Heeman (1990) merged expressions in brackets, the composite nodes have at most one child which, because of the order in which the brackets are matched, will have no unmatched composite nodes so, when brackets are successfully matched, the resulting action is simply to execute an operator precedence merge between the single child under the left bracket and the single child under the right. This holds for any composite form that has only one child. For this algorithm the cases of a composite node involved in a merge having more than one child need to be considered. (This is because composite nodes can represent templates, which may have more than one child.) If a child of an incomplete composite node that has been identified for matching is on the "boundary"[8] of the merge, any composite nodes below it will have already been matched, so in this case an OP merge on the child is also appropriate.

In the case of composite forms having multiple children, the algorithm caters for all forms with a fixed number of children.

[8]Here, *boundary* refers to the set of nodes that may interact with nodes from the other tree during the merge.

Algorithm 4 *makeTrailOfUnfinished*$(t) \to r$

Create stack r of unfinished composite nodes in t, having the topmost unfinished node at the bottom

```
set $r$ to empty stack, set $c$ to root of $t$, $s$ to *searching*
while $s$ is not *done* do
   set $f$ to false
   if $s$ is *searching* then
      if $c$ is composite then
         if last delimiter of $c$ is present then
            set $s$ to *done* // subtree closed
         else // we could be at top of staggered closed composite
            empty $r'$ and push $c$ onto $r'$
            set $s$ to *in_unfinished*
         end if
      end if
      if $c$ has any children and $s$ is not *done* then
         set $c$ to right-hand side of binary operator, or last non-empty child, or else empty
         node
         if $c$ is now empty then
            set $f$ to true  if $s$ is now *in_unfinished*
            set $s$ to *done*
         end if
      else // no children
         set $s$ to *done*
      end if
   else // $s$ must be *in_unfinished*
      if $c$ is composite then
         if first first delimiter of $c$ is present then
            set $f$ to true, $s$ to *new_found*
         else
            if last delimiter of $c$ is present then
               set $s$ to *done* // subtree closed
            else // we could be in continuation of a staggered closed composite
               push $c$ onto $r'$
            end if
         end if
      else // $c$ not composite, so opened composite never closed
         set $f$ to true, $s$ to *searching*
      end if// by now, if $c$ not composite, $s$ will have been set to *searching*
      if $s$ is not *done* then
         if $s$ is *new_found* then
            set $s$ to *searching*
         else // we need to move down tree
            if $c$ has any children then
               try to move $c$ to right-hand side of binary operator, or last non-empty child
               set $f$ to true, $s$ to *done* if $c$ is now empty
            else // no children, so opened composite never closed
               set $f$ to true, $s$ to *done*
            end if
         end if
      end if
   end if
   push $r'$ onto $r$ if $f$ is true
end while
return  $r$
```

Algorithm 5 *makeTrailOfUnstarted*$(t) \rightarrow r$
Create stack r of unstarted composite nodes in t, having the topmost unstarted node at the bottom

```
1: set $r$ to empty stack
2: set $c$ to root of $t$
3: set $s$ to true
4: while $s$ do
5:    if $c$ is composite then
6:       if first delimiter of $c$ is present then
7:          set $s$ to false
8:       else
9:          push $c$ onto $r$
10:      end if
11:   end if
12:   if $c$ has any children then
13:      if $c$ is a binary operator then
14:         set $c$ to its left-hand child
15:      else
16:         set $c$ to its first non-empty child, or empty tree if no such child
17:      end if
18:      if $c$ is now empty then
19:         set $s$ to false
20:      end if
21:   else // no children were found
22:      set $s$ to false
23:   end if
24: end while
25: return $r$
```

Testing Whether Composite Nodes Match

Before two composite nodes are combined, a determination has to be made on whether they match. The test on whether a composite node t_L fits together with composite node t_R works as follows:

1. t_L and t_R must represent the same template.
2. We assume that t_L is missing its last delimiter and t_R is missing its first. (If this were not the case, they would not be classified as unfinished and unstarted respectively.)
3. Let $d_1, d_2, \ldots$ represent the delimiters of the template, and let $j > 0$ be the position of the last delimiter of t_L that is present. Let c_i represent a child of a composite node, where a delimiter d_i separates c_i from c_{i+1}.

4. The only child of t_L permitted to be present after d_j is c_{j+1}.
5. The only children that may be present in t_R are c_{j+1} onwards.
6. The only delimiters that may be present in t_R are d_{j+1} onwards.

Informally stated, items 4 and 5 stipulate that nodes with overlapping sets of child nodes are classified as non-matching, with the exception of any child that falls between the two extant delimiter lists.[9] Item 6 states that nodes with overlapping lists of delimiters are classified as non-matching. Item 1 states that nodes that could potentially fit together although they represent different templates are not considered as matching.

Combining Matching Composite Nodes

The approach is as follows. Given two matching composite nodes T_L and T_R, delimiters and children in T_R but not in T_L are moved to T_L. If both T_L and T_R have a child that falls between the last extant delimiter in T_L and the first extant delimiter in T_R, they are merged. (Such children are those that appear on the tree boundary as described above.)

The question needs to be addressed of into which "slot" in the template a child should be placed when sufficient separators are missing to make the meaning of the resulting expression ambiguous. As an illustration, suppose a template takes three arguments, and so has separators $\|_0$ $\|_1$ $\|_2$ $\|_3$, and suppose the (incomplete) expression[10] "$\|_0\ A\ \|_1\ b + c\ \|_3$" is being dictated in two separate utterances "$\|_0\ A\ \|_1\ b$" and "$+\ c\ \|_3$". Although the "b" and the "$+\ c$" clearly belong together, we have the question of where to place the merged content. Given that the batch parser will place the subtree for $b + c$ into the slot between delimiters $\|_1$ and $\|_2$, the incremental parser will follow this convention.

Staggering Badly-matched Composite Nodes

A "badly matched" pair of composite nodes consists of those that failed to match each other, but need to be placed together in the result of the merge.

[9]While incomplete composite nodes with more than one "overlapping" child could be regarded as matching, there is no way in which they could be merged while preserving the yield of the resulting tree. Because of this, they are not considered as matching.

[10]Note, in this example the $\|_2$ is omitted deliberately.

Irrespective of the approach taken to merging composite nodes, "badly matched" nodes will be put together as shown in Figure 4.6. The approach used here will not involve creating more than two composite nodes at one time from a bad match: when the two nodes are created, one will simply appear under the other in the hierarchy, as stipulated by the right-associative convention. Note, reference is made to "unclosed" and "unopened" rather than the "opening" and "closing" parentheses discussed by Heeman (1990), because an internal delimiter may be either unclosed or unopened, or both. (For an example illustration that includes two badly matched composite nodes, see Figure 4.5.)

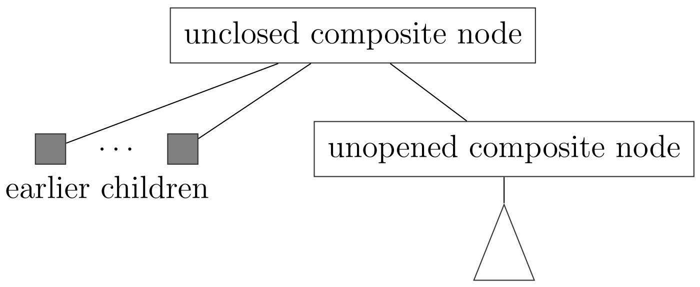


Figure 4.6: Badly matched pair of composite nodes

Observations

This section has described a method for matching and merging composite nodes that designates two incomplete nodes as matching only if they belong to the same template. If this algorithm were to be applied to a programming language defined in a usable way, the method would need to be made more flexible. The following observations will be relevant.

1. Provided the combination of a pair of incomplete composite nodes does not create new incomplete non-staggered composite nodes, the approach to identifying incomplete composite nodes will remain valid.

2. Any single composite node belongs to a single template.

3. Because no variable length templates exist, composite nodes will have the appropriate number of blanks inserted in missing delimiters by the initial (batch) parse and any subsequent tree operations.

4. Because of (3), any composite node will have delimiters in appropriate places, with missing delimiters denoted by blank entries.

5. The algorithm cannot rely on the children being in the places originally intended by the user if they have provided incomplete input.

6. If too many delimiters are missing, it may not be possible to identify a single template without referring to the template originally recorded against the composite node. (The approach presented in this chapter refers only to the original template; this would have to change if fragments of distfix operators were to be combined. See Section 6.6.3.)

The application of the algorithm to programming languages is discussed further in Chapter 6.

4.3.5 Brackets, Composite Nodes and Associativity

In the case of incomplete expressions, the move from simple brackets to composite nodes introduces the question of associativity. The brackets defined by Heeman (1990) are described as having no associativity. But when brackets of different types are matched with each other during a merge, the closing bracket is placed under the opening bracket in the AST, suggesting right associativity, as the structure produced consists of just an opening bracket with a single child which is a closing bracket with a single child the merge of both of their children. (See Figure 4.7.)

$$\begin{tikzcd}
\textit{opening bracket} \arrow[d, no head] \\
\textit{closing bracket} \arrow[d, no head] \\
\textit{merged children}
\end{tikzcd}$$

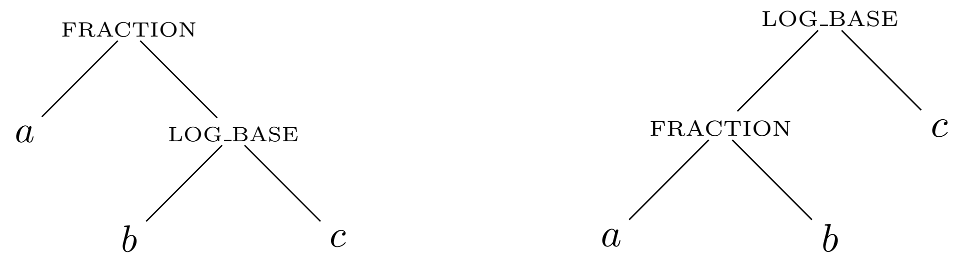
Figure 4.7: Staggered unmatched brackets as per algorithm of Heeman (1990). (The standard AST notation is used here.)

The point at which associativity of composite nodes becomes relevant is when they have more than one child. Consider two templates (from the *TalkMaths* language definition[11]) `FRACTION` and `LOG_BASE`. The `FRACTION` template has delimiters "fraction", "over" and "end fraction", while `LOG_BASE` has delimiters "log base", "of number" and "end log base" (to enable the user to write expressions such as $log_a b$). The tree that will be produced for expression "fraction a over b of number c end log base" where the

[11]The *TalkMaths* language definition forms part of the *TalkMaths* Python code, and is held in the file *MathsDefinition* class.

templates are left associative will differ from that produced if they are right associative, as illustrated in Figure 4.8. (In this illustration, Figure 4.8a represents $\frac{a}{log_b c}$, while Figure 4.8b represents $log_{\frac{a}{b}} c$.)

Attanayake (2014) treated all (closed) mixfix operators as right associative, which happens to be consistent with Heeman (1990). This version adheres to that convention. The convention itself is quite reasonable, but problems would arise upon encountering the left-associative PHP ternary operator (?:) if this parser were to be applied to that programming language. One possible solution would be to record precedences against mixfix operator parts (in other words, against individual template delimiters), but this would mean a precedence matrix between mixfix operators would need to be built, creating a substantial amount of work for the author of any language definition and requiring fundamental changes to the algorithm. It is questionable whether the cost of introducing this extra complexity in language setup would be warranted, given the rarity of such a mixfix operator[12].



(a) "fraction a over b of number c end log base" with current (right) associativity

(b) "fraction a over b of number c end log base" with left associativity (not used in this thesis)

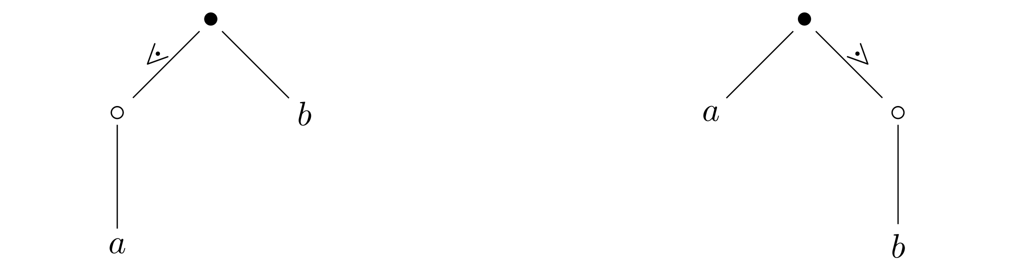
Figure 4.8: Effect of associativity on composite nodes with more than one child

4.3.6 Avoiding Violation of OP by Unary Operators

According to the definition of the operator precedence relations given by Aho (1972; p. 438), if in an expression operator a appears before operator b with $a \lessdot b$ then b is permitted to be binary or prefix, but not postfix, and if operator c appears before operator d with $c \gtrdot d$ then c is permitted to be binary or postfix, but not prefix. If a language definition contains such relations, their combination will violate the conditions for it to conform to OP grammar, and so a parser will not produce well formed parse trees under certain circumstances. The problematic relation combinations are as follows, and are illustrated in Figure 4.9.

[12]The PHP ternary operator is the only example the author has encountered that has left associativity.

- The left operand of a binary operator is a postfix operator with precedence lower than its parent.
- The right operand of a binary operator is a prefix operator with precedence lower than its parent.



(a) Postfix incorrectly under binary (b) Prefix incorrectly under binary

Figure 4.9: Operator precedence violations that cannot be solved by twiddle operations

Provided such relations are not introduced when designing an OP language, violation of OP may be avoided for combinations of binary and unary operators. The author notes that the precedences assigned to mathematical operators do not cause the above problems.

The case is not so simple when two prefix or two postfix operators are combined. For example, consider the dictated expression "a squared factorial". It would appear to describe $(a^2)!$ where the factorial is applied to a^2, but because $! \gtrdot \mathit{power}$, the tree for such an interpretation would violate operator precedence. To avoid such a situation, the batch OP parser places the power at the top of the tree, in effect creating expression $(a!)^2$ but this is inconsistent with what the user dictated. If a situation like this is encountered by the incremental parser implementation, it is handled by placing a complete bracket pair between the operators closest to the operand, creating the tree for $(a^2)!$. Although the encounter of such bracket pairs might seem rather surprising to the user, this would be preferable to changing the order of the operators, and hence their meaning.

4.3.7 Criteria that Must be Met by the Default "Concatenation" Operator

Whenever two ASTs are merged, there is a possibility that an extra operator will need to be introduced to represent the concatenation of the expressions or code. For example if the expressions "$a + b$" and "$c + d$" are to be concatenated, a decision must be made on how to handle the "bc" that appears in the middle. In the case of mathematical content, the operator in question is typically invisible multiplication. If in the context of the application (the content being edited), no such ready-made operator is available, one needs to be chosen that:

(a) does not cause the language to violate the operator precedence conditions, and

(b) makes sense.

In the context of a spoken programming language, a statement separator may be chosen instead by the implementer.

4.4 Theoretical Evaluation

This section discusses the asymptotic time complexity of the algorithms for the incremental parsing operations *tear* and *merge*, and compares these with the batch parser.

4.4.1 *tear*

Examining lines 6 and 7 in Algorithm 1, the only non-trivial actions required would be to move delimiters and child nodes. Because the composite nodes used here have fixed numbers of delimiters (and children), this may be treated as $\mathcal{O}(1)$.[13] Similarly the body of the loop starting at line 16 is also $\mathcal{O}(1)$. The loop itself is repeated for each node in the chain of parent nodes from the node containing token k up the root node. Hence, *tear* is $\mathcal{O}(d + 1)$ where d is the depth of the node containing k. If we wish to express this in more general terms, it is $\mathcal{O}(h)$ where h is the height of the tree.

[13]If the nature of composite nodes were to be modified in order to permit a variable number of arguments, for example to facilitate representation of lists as flat trees, the number of separators and children in the composite node would feature in the complexity calculation.

4.4.2 *merge*

The time complexity for this algorithm is less simple. We begin by defining the following. Subscripts L and R are used to indicate the left-hand and right-hand trees involved in the merge, respectively.

h_L, h_R	Heights of the trees.
e_L	The number of operator nodes on the right boundary of T_L.
e_R	The number of operator nodes on the left boundary of T_R.
c_L	The number of unfinished composite nodes on the right boundary of T_L.
c_R	The number of unstarted composite nodes on the left boundary of T_R.

We also note that the number of nodes (of any type) on the boundary of either tree cannot exceed the height of the tree itself by more than 1, and so a function involving tree height may be treated as specifying an upper bound for a run time expressed in terms of e_L, e_R, c_L or c_R.

The *match_merge* algorithm consists of two phases: (1) constructing the lists of incomplete composite nodes, and (2) performing the merge. For (1), both list construction procedures involve travelling once down the boundary of the tree in question until a completed composite node is encountered, so the worst-case run time for this stage (which would be the case where the trees contain no composite nodes) would be some $f(h_L) + g(h_R)$ with f and g linear functions.

The time for phase (2) is more difficult to predict. Depending on the content of the boundaries of T_L and T_R, the process could consist merely of combining (or staggering[14]) incomplete composite nodes (in the case of the tree boundaries having no operator nodes) at one extreme, to being a pure operator precedence merge operation at the other (for trees containing no incomplete composite nodes).

Within every matched pair of partial composite nodes, operations of *opmerge*[15] will potentially occur at every operator on the "inside boundary" of the subtrees under the matched pair. Because matched pairs from the two sequences are nested, and *opmerge* does not go inside complete ("whole") composite nodes, the net result is that each

[14]See Section 4.3.4 for a description of staggering.

[15]Low-level *opmerge* operations are referred to as *opmerge* operations here, for brevity.

operator on the right boundary of T_L and the left boundary of T_R may be involved 0 or 1 times in one of the low level *opmerge* operations. *opmerge* itself works down once from its starting node and up once as it grafts the tree fragments back together. Twiddle operations may occur at any time during the reassembly of the subtrees.

Boundary nodes are not processed more than once by an *opmerge* operation. The reasoning is as follows. Let us consider the cases of any boundary node:

1. Operator node above all incomplete composite nodes
2. Operator node below all incomplete composite nodes
3. Operator node between incomplete composite nodes
4. Incomplete composite node above all operator nodes
5. Incomplete composite node below all operator nodes
6. Incomplete composite node between operator nodes
7. Identifier node or complete composite node

In case (1) the node will be involved in a single *opmerge* operation after all incomplete composite nodes have been resolved. In case (2) the node will be involved in at most one *opmerge* operation when the bottom-most incomplete composite node is combined. In case (3), all incomplete composite nodes below it will have been resolved, and the node itself will be involved in at most one *opmerge* operation when the incomplete composite node above it is combined. In case (4), the node will not be involved in any *opmerge* operation. In cases (5) and (6) the node will be involved once in any *opmerge* operation for the highest operator node above it. In case (7), the node will be treated as a leaf of the tree and so will not be actively involved in any *opmerge* operation.

Because boundary nodes are processed at most once, and once all the boundary nodes on one of the two trees are "used up" the OP merge will be complete, we can express the run time for all the *opmerge* elements of the merge as some function $f(\min(h_L, h_R)) + g(\min(h_L, h_R))$ for some linear function f for the operations on the way down and up on the nodes involved, and some linear function g to represent any twiddle operations required. We express the function in terms of $\min(h_L, h_R)$ because once the *opmerge* operation runs out of nodes on one of the boundaries involved in its processing, it must stop, and $\min(h_L, h_R) + 1 \geq \min(e_L, e_R)$.

match_merge operations are performed $\min(c_L, c_R)$ times because one cannot match composite nodes that do not have a counterpart. Each *match_merge* operation involves a check on whether the pair of incomplete nodes can be combined into one, followed by the combination of the two nodes, either into a single composite node or into a staggered subtree. While these operations are not trivial (they involve looping through delimiters and children), because the number of these is fixed in the language definition, both the check and the combination can be regarded as $\mathcal{O}(1)$. In both cases, the combination step may involve a top-level *opmerge* operation, for which we have already accounted above. Hence we can express the run time for the *match_merge* aspect of the merge in terms of some linear function $m(\min(h_L, h_R))$ because $\min(h_L, h_R) + 1 \geq \min(c_L, c_R)$.

The above argues that the run time of every aspect of *merge* can be expressed using linear functions of $\min(h_L, h_R)$, and so *merge* is $\mathcal{O}(\min(h_L, h_R))$.

It would be useful to verify this experimentally, but because of the time currently taken to perform a batch parse of fresh text, it is not feasible to collect data on a sufficiently wide range of tree heights.

4.4.3 Comparison with Batch Parser

In this section it is noted it is not feasible to make a generic comparison from a theoretical point of view, so comparisons are made using the two main operations *tear* and *merge*.

The batch parser developed by Attanayake (2014) has two phases: the XGLR parsing algorithm developed by Begel (2006), followed by consolidation of composite nodes. Begel's algorithm may be $\mathcal{O}(m^3)$ for a token string of length m, while the composite node consolidation phase is $\mathcal{O}(n)$ where n is the number of nodes in the parse tree produced by the XGLR parser. $\mathcal{O}(m^3)$ is the generally recognised worst case time complexity for GLR parsers; the case for the XGLR parser (Begel, 2006) is complicated by the fact that multiple alternative parses are spawned on encountering lexical ambiguity. The number of nodes in the parse tree cannot exceed the length of the token string, so the time complexity of the Attanayake batch parser could be said to be $\mathcal{O}(m^3)$ for a token string of length m.

Considering the *tear* operation on a token string of length n and discarding the portion of the string before or after position p, the time complexity for the batch parser's equivalent (re-parsing one or other of the token string portions) will be $\mathcal{O}((n-p)^3)$

or $\mathcal{O}(p^3)$ respectively. The maximum height of a tree representing a token string of length n will be $n-1$ so the time complexity for the incremental *tear* operation will be $\mathcal{O}(n)$, making it asymptotically more efficient for all but very small $|n-p|$.

Considering the *merge* operation on right and left token strings of lengths m and n, the time complexity for the batch parser's handling of their concatenation will be $\mathcal{O}((m+n)^3)$. Estimating the maximum height of a tree as before, the incremental merge operation will have time complexity $\mathcal{O}(\min(m,n))$. From these considerations, the following theorem can be deduced.

Theorem 1 *The time complexity of merging two trees representing right and left token strings of lengths m and n is $\mathcal{O}(\min(m,n))$, which is better than the corresponding batch parsing complexity, which is $\mathcal{O}((m+n)^3)$.*

An experimental comparison is made in Chapter 5, confirming that for the case of y or y' being longer than one token, the incremental method is significantly faster.

Difficulty of the General Case

If we were to compare time complexity of the incremental and batch approaches in the traditional scenario of incremental parsing, we would need to begin by defining the task to be carried out using these two methods. Using the typical incremental parsing scenario of replacing substring y with y' in token string xyz, we would compare the time taken to reparse the entire replacement string $xy'z$ with that taken by the incremental parsing algorithm to perform the replacement using the tree operations. One way of doing this would be (1) split between x and yz, (2) split between y and z, (3) batch parse y', (4) merge x and y', and finally (5) merge xy' and z. (Other ways would involving changing the order of splitting and merging, e.g. split between xy and z first.) The time complexities of the incremental parsing operations in terms of token string length have already been established (for the very worst case), but the lengths of y, y' and xyz may vary independently (notwithstanding the restriction that the length of y is less than length of xyz). Because of this, the time complexity is not amenable to analysis (Howell, 2008).

4.5 Conclusion

This chapter has presented a novel algorithm based on improvements and extensions to the incremental parsing algorithms developed by Heeman (1990), that enable this to be applied to a language that is not limited to simple operators and brackets.

The theoretical time complexities of the two operations are compared with the batch parsed equivalent, and the incremental versions are found to be more efficient.

The next chapter deals with the application of the algorithm to spoken mathematics and revisits the comparison of the parsing approaches from a practical standpoint.

Chapter 5

Application to Spoken Mathematics

This chapter describes the application of the algorithm to *TalkMaths*, and compares practical run times for changes that will need to be made in response to some edit operations, as performed by the batch parser alone and the incremental parser combined with the batch parser as required.

5.1 *TalkMaths*

TalkMaths is the product of research carried out at Kingston University on creation and editing of mathematical content using a speech interface.

The batch parser used for the current version, originally developed by Attanayake (2014), is implemented in Python (and so is platform independent), and has an interface designed to be used via a RESTful web service. The spoken mathematical language is maintained in a class called *MathsDefinition*, which defines operators with precedence relations, identifiers, and the templates designed by Attanayake (2014). Operators and templates may be modified as required by the language user, who has the responsibility to ensure their definitions are consistent and do not violate the requirements of the operator precedence parser. One requirement of this parser which will have significant consequences for further development is that no single lexeme may be used in more than one template. (For example, although it may be desirable for the user to finish the templates for both definite and indefinite integrals with the phrase "end integral", this will not be permitted in the language definition.) The fact that no individual token may appear in more than one distfix operator is consistent with Aasa (1995), who delegates the responsibility for distinguishing between operators that share a lexeme to the lexer (citing binary and unary plus as an example). The Attanayake (2014) parser can handle such ambiguities by producing multiple parse trees, in effect deferring the decision on which meaning the user intended when they dictated the expression, but this functionality is switched off for the purpose of performance evaluation, to minimise the response times of the batch parser.

The ASTs produced by this parser have standard nodes for identifiers and the basic (unary and binary) operators. For anything else, including distfix operators, functions and brackets, the trees contain composite nodes which behave like the function nodes described by LaLonde and des Rivieres (1981). The ASTs produced are valid according to operator precedence, and all nodes contain the required number of children, which may be blank (but not missing) where the user has not supplied all required information. Where the user has omitted delimiters, these are also recorded as blank.

The language definition for *TalkMaths* is summarised in UML in Figure 5.1. This consists of collections of simple operators with precedences, identifiers and templates. Operators are identified by their tokens, which are recorded in the parse tree along with the lexeme that matched the operator's lexeme list at parse time. Operator placement is either prefix, postfix or binary. Every operator will belong to an operator precedence class, and it is these classes that are used to determine the precedence relation between two operators, which will be either $\lessdot$ or $\gtrdot$. This mechanism, designed by Attanayake (2014), reduces the size of the operator precedence table through it not having to contain an entry for every possible combination of operators. For example, both $+$ and $-$ are classified as PLUS_MINUS, while all comparison operators belong to precedence class LOGICAL_OP, so only four entries are required in the operator precedence table to cover all combinations of $+$, $\leq$ and other operators belonging to the same precedence classes. Identifiers consist of single characters (rather than strings of characters), for example h, or 5, that would be recorded along with token identifiers, for example SPOKEN_HOTEL and SPOKEN_NUMBER, had their regular expressions matched lexemes "hotel" and "five". Each template consists of its identifier (for example SINE) along with a sequence of delimiters, each of which will have one or more possible lexemes. (In this example, "sine" or "sine of", followed by "end sine".) Again, both identifier and actual matching lexeme are recorded.

Figure 5.2 shows the essential structure of the data produced by the batch parse. Apart from the *has parse tree* relationship and *child index* attribute (both added to implement the incremental parsing algorithm), the elements are as found in the batch parser at the time work on the incremental parser began. The *Parse Tree* class represents a single

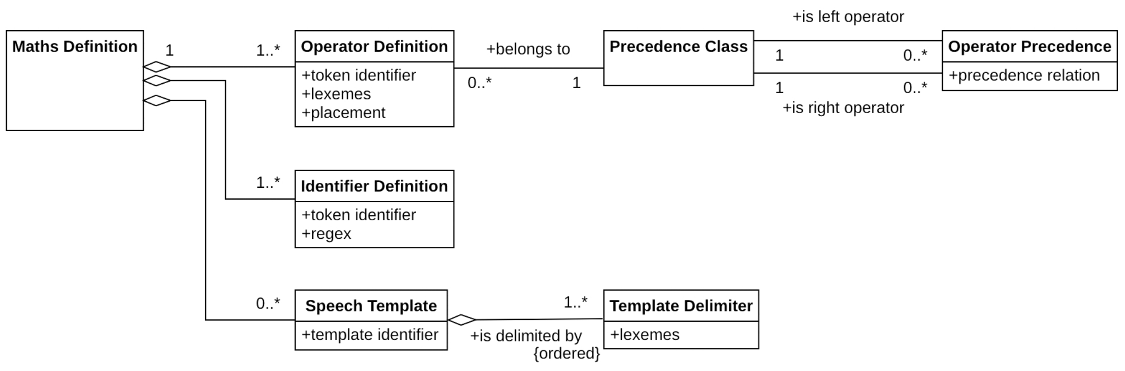


Figure 5.1: Language definition (mathematics)

node of the parse tree, with the tree structure represented by *has child* and *has parent*. Tokens are associated with a parse tree node via the Node class, with bracket type indicating whether the token represents an opening or closing bracket, and *identifier* being the token identifier from the language definition. The lexeme associated with the token is held in a *Symbol Table*[1] object, one of which is associated with each token.

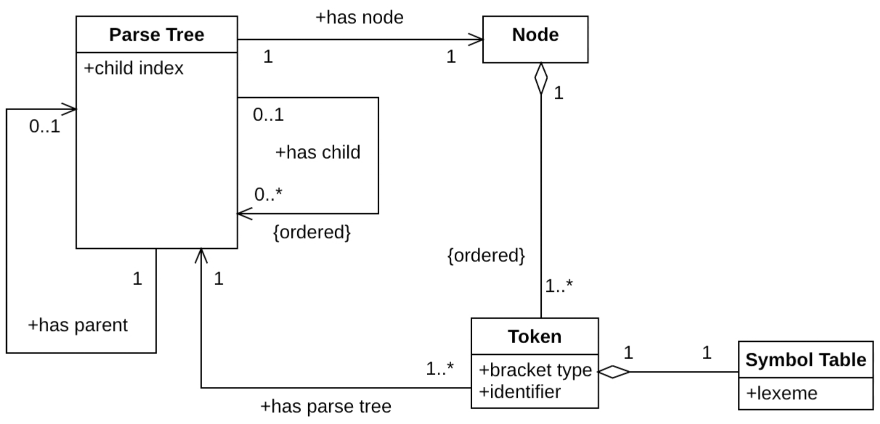


Figure 5.2: Data produced by batch parse

The batch parser is implemented by classes named *OperatorPrecedenceParser* and *Scanner*. Apart from bug fixes and a performance improvement, these classes have been treated as black boxes. To perform a batch parse, one must create a *MathsDefinition* object, and then use that as an argument to instantiate an operator precedence parser object, of which the *parse* method must be invoked on the string representing the recognised speech. The following fragment of Python code creates the tree for $a = \sin B$.

[1]It is thought this is evidence of an earlier intention to use a symbol table as part of the batch parser.

```
mathsDefn = MathsDefinition()
batchParser = OperatorPrecedenceParser(mathsDefn.mySpokenLanguageData)
aTree = batchParser.parse("alpha equals sine of capital bravo end sine")
```

5.2 Implementation of Incremental OP Parsing Algorithm for Use with *TalkMaths*

The algorithms have been implemented in Python 2.7[2] to work with these ASTs. To facilitate this, minor modifications to the data structures produced by the batch parser (but not to the language definition) were required.

5.2.1 Modifications to Existing Classes

As part of the implementation of the incremental parsing algorithm, a number of new methods were required for the *ParseTree* and *Node* classes. These were to implement:

- informational functionality, for example determining whether a *ParseTree* object represents a composite node;
- child retrieval, for example to return a reference to the first non-empty child of a *ParseTree* object;
- adding, inserting, removing or replacing a child;
- all twiddle operations;
- operations to combine or split apart composite nodes, and
- operations to create new parse tree nodes, for example to represent the default operator or insert brackets around an existing subtree to preserve operator precedence validity (see Section 4.3.2).

[2]This is for compatibility with the software used to link the ASR package with *TalkMaths*.

5.2.2 The Incremental Parser

The class *IncrementalParser* implements all major[3] parts of the algorithm described in this thesis. Once an instance of this parser has been created, the *tear* and *merge* methods can be used to perform the top level operations. (*tear* and *merge* simply call *do_tear* and *match_merge* respectively.)

The example code shown in Figure 5.3 models, for illustration purposes, the scenario of a user who having dictated the expression $a = \sin(By)$, replaces (for their own reasons) By with $C^2 + 1$ to produce expression $a = \sin\left(C^2 + 1\right)$. In a production environment, the tokens at which to tear the trees would have been received via some editor, but here we need to identify these tokens by producing the yield of the tree and locating the tokens by lexeme. This is done using a class called *TreeYield*[4]. The final tree is displayed using an original *ParseTree* method. The class that produces the LaTeX translation at the end was developed because it would be of immediate use to the author.

The output is as follows.

```
alpha equals sine of capital bravo yankee end sine
alpha equals sine of capital charlie squared plus 1 end sine
OP[EQUAL](equals)
    ID[SPOKEN_ALPHA](alpha)
    MIXFIX_OPERATOR[SINE](sine of | end sine)
        OP[BINARY_PLUS](plus)
            OP[SQUARED](squared)
                ID[CAPITAL_SPOKEN_CHARLIE](capital charlie)
            ID[SPOKEN_NUMBER](1)
a = \sin{ C ^{2} + 1 }
```

The incremental parsing algorithms were tested by splitting and merging test data obtained from the requirements and code based test cases derived from the algorithms. The practical evaluation also formed a testing process, as the result of the incremental parsing process was compared with that of the batch parse, and any difference flagged.

[3]Operations such as the twiddles are implemented in whichever class is most appropriate, for technical reasons.

[4]This cannot be used as a token stream because it includes blank tokens where parts of an expression are missing

```
from OperatorPrecedenceParser import *
from MathsDefinition import *
from ParseTree import *
from IncrementalParser import *
from TreeYield import *
from LatexMaker import *
mathsDefn = MathsDefinition()
batchParser = OperatorPrecedenceParser(mathsDefn.mySpokenLanguageData)
theIncrementalParser = IncrementalParser()
startText = "alpha equals sine of capital bravo yankee end sine"
print startText
aTree = batchParser.parse(startText)
# user isolates capital bravo yankee
aYield = TreeYield(aTree)
lastLeftTreeToken = aYield.getTokenAtIndex(aYield.findLexemeIndex("sine of"))
lastIsolatedToken = aYield.getTokenAtIndex(aYield.findLexemeIndex("yankee"))
# split
[tempTree, rightTree] = theIncrementalParser.tear(aTree, lastIsolatedToken)
[leftTree, isolated] = theIncrementalParser.tear(tempTree, lastLeftTreeToken)
replacementTree = batchParser.parse("capital charlie squared plus one")
interimTree = theIncrementalParser.merge(leftTree, replacementTree)
finalTree = theIncrementalParser.merge(interimTree, rightTree)
print TreeYield(finalTree)
finalTree.printTree()
lm = LatexMaker()
print lm.translateToLatex(finalTree)
```

Figure 5.3: Example use of incremental parser

5.3 Evaluation: Practical Performance

In this section, the elapsed times taken to carry out some simple editing and parsing operations are compared, using the batch parser alone and the incremental parser together with the batch parser where required. The timings were carried out using the version of the batch parser after the performance improvement was implemented.

Three scenarios are considered:

1. Simple concatenation, which will involve merging only.

2. Insertion of a missing part of an expression, which will involve both splits and merges.

3. Simple deletion, which will involve splitting only.

5.3.1 Scenario 1 – Simple Concatenation

In this scenario, an expression is split into two parts, corresponding to two utterances, that together make up the complete expression. Expressions range from very short ones to those that realistically would take multiple utterances to dictate. The split point ranges from immediately after the first token to immediately before the last token.[5]

The scenario is very simplistic and does not model a real-life situation (which would involve more utterances for longer expressions); its intention is to compare run times for concatenation.

It corresponds to performing the first tear operation in the above example for *lastIsolatedToken* ranging from "alpha" to "Yankee".

5.3.2 Scenario 2 – Insertion of Missing Part of an Expression

In the second scenario, our starting point is an expression assumed to have already been dictated by the user, but with an omission. An insertion point, and the missing part to be inserted are provided. For simplicity, the latter consists of a single token[6]. This will not affect the number of top-level operations that the incremental parser has to perform, nor will it change the length of token string to be processed by the batch parser. This new expression fragment is inserted into the existing expression, as follows:

Pure batch approach The entire token string (including the inserted part) is reparsed. (Note, the trivial operation of string concatenation is not timed.)

Incremental approach The new utterance is parsed, after which *split* and *merge* operations are used to combine this with the existing tree to produce the required result.

To illustrate, it is the equivalent of running the following code fragment, but for the missing part of "alpha equals sine of capital bravo Yankee end sine" ranging from

[5]These split points are on tokens and not the individual words that make up a lexeme. Splits on individual words would raise the issue of error handling, which is not evaluated.

[6]It is impractical to simulate every possible split of the sample expressions into three, or even produce a large enough sample for random insertions into strings from the data set, so the simple scenario described above is followed.

"equals" to "Yankee" (and including the two invisible multiplication operators).

```
startText = "alpha sine of capital bravo yankee end sine"
missingText = "equals"
print startText
aTree = batchParser.parse(startText)
aTree.printTree()
missingNode = batchParser.parse(missingText)
aYield = TreeYield(aTree)
splitPoint = aYield.getTokenAtIndex(0)
[leftTree, rightTree] = theIncrementalParser.tear(aTree, splitPoint)
newLeftTree = theIncrementalParser.merge(leftTree, missingNode)
finalTree = theIncrementalParser.merge(newLeftTree, rightTree)
print TreeYield(finalTree)
finalTree.printTree()
```

The output of the code fragment is reproduced below, to illustrate the use of invisible multiplication.

```
alpha sine of capital bravo yankee end sine
OP[INVISIBLE_TIMES]()
    ID[SPOKEN_ALPHA](alpha)
    MIXFIX_OPERATOR[SINE](sine of | end sine)
        OP[INVISIBLE_TIMES]()
            ID[CAPITAL_SPOKEN_BRAVO](capital bravo)
            ID[SPOKEN_YANKEE](yankee)
alpha equals sine of capital bravo yankee end sine
OP[EQUAL](equals)
    ID[SPOKEN_ALPHA](alpha)
    MIXFIX_OPERATOR[SINE](sine of | end sine)
        OP[INVISIBLE_TIMES]()
            ID[CAPITAL_SPOKEN_BRAVO](capital bravo)
            ID[SPOKEN_YANKEE](yankee)
```

5.3.3 Scenario 3 – Deletion of Part of an Expression

This exercises the *split* operation on its own by comparing the time taken to delete all of an expression to the left of a split point (using the same parameters as the concatenation scenario), or all of an expression to its right. In the case of the batch approach, this would consist simply of reparsing the part of the expression not to be deleted. The incremental approach will split the AST for the expression into two ASTs, and so the operation is the same, whichever part is to be retained.

This scenario corresponds to the last code fragment executing just the tear for split point ranging from 0 to the index of "Yankee".

5.3.4 Method

To evaluate performance, the elapsed time[7] taken to perform an operation using the pure batch approach is compared with elapsed time taken using the incremental approach. If one approach takes significantly less time than the other then one can conclude the approach that takes less time performs better than the other.

The sample data used are listed in Appendix C and include, but are not limited to, some common equations along with expressions used in an assessment scenario (University of Oxford Mathematical Institute, 2016). Some of the expressions are fragments, while others, though they represent complete expressions, miss closing delimiters at the end of the expression in its spoken form.[8] The timings are performed by processing the entire list of expressions in the same sequence 27 times[9]. The platform is a PC with Intel Core i7-4790 processor and 16GB RAM running Windows 10.[10]

Scenario 1 – Simple Concatenation

Each sample expression is split at every gap between tokens. For the pure batch approach, the time taken to parse the entire expression is recorded. For the incremental approach, the starting point is a parse tree for the first part of the expression, and the total time taken is calculated as the sum of the time taken to batch parse the second part of the expression, and the time taken to merge the resulting tree with the original one.

[7]For evaluation on a Windows platform, Python's time.clock() is used as it provides the required precision. The equivalent on a Mac is time.time().

[8]This is a nod towards realism: the batch parser's error handling will cope with such omissions, and the author took advantage of this fact when entering the data.

[9]Sample sizes were chosen to be as large as possible while allowing the process to finish overnight.

[10]Timings on a PC gave more consistent results from batch to batch than the same procedure when run on a Mac.

Scenario 2 – Insertions

As before, the batch parsing simulation consists merely of parsing the entire expression (as all concatenations will have been made at the string representation level).

For the incremental parse, the starting point for an expression whose entirety would be $s_1s_2s_3$ would be the tree for s_1s_3. Inserting the missing part consists of splitting s_1s_3 at the boundary between s_1 and s_3 (this is the *tear* operation), followed by the parsing of s_2, and subsequent merging of s_1 with s_2 and s_2 with s_3.

This scenario is far from ideal because the timings of the incremental approach include the batch parsing of only a single token, resulting in timings that are overly favourable towards the incremental approach when compared with the timings for simple concatenation. It is included only for completeness.

Scenario 3 – Simple Deletion

As in scenario 1, this process loops through all possible split points. For the pure batch approach, deletion of the part of the expression will consist of extracting the substring to be retained (not timed), and then parsing it.

For the incremental approach, the deletion consists of splitting the tree at the appropriate split point.

5.3.5 Results

For all three scenarios, the first sample of the 27 repetitions was discarded, as the mean time taken for this round was noticeably shorter than for the others, using both approaches[11]. (See table in Appendix D.) It was not possible to draw any conclusions on the distributions of the timings, so they were compared using non-parametric tests.

Scenario 1 – Simple Concatenation

A Mann Whitney U test on sums of total times taken to perform the concatenation, aggregated by sample, gave a test statistic representing total ranks for the batch

[11]The exception was the incremental parsing run time for scenario 3, that involved no step involving batch parsing.

approach minus total ranks for the incremental approach of 676 (with $N = 26$). (In fact, the incremental approach was faster than the batch approach in all cases.) This means that the incremental approach is faster, with a ****p-value of 4.0329×10^{-15}. Based on the sample of 26 values, the incremental approach is 1.984272 (95% CI [1.984080, 1.984464]) times faster than the batch approach.

Taking mean times for tuples of expression and split position for the merge as individual data points, a Wilcoxon signed rank paired difference test (with sample size $n = 802$ and test statistic 321967) leads one to conclude that the incremental approach is faster than the pure batch approach, with ****p-value 2.22×10^{-16}. (The test statistic is the sum of ranks of the batch approach processing time minus sum of ranks of the incremental approach processing time.)

Scenario 2 – Insertions

A Mann Whitney U test was carried out on sums of total times taken to perform the concatenation, aggregated by sample. The test yielded exactly the same results as that for concatenation, as the incremental approach took less time than the batch approach in all cases. Based on the sample of 26 values, the incremental approach is 9.034268 (95% CI [9.031699, 9.036836]) faster than the batch approach. (Recall, in this scenario, the incremental approach is given an unfair advantage.)

Performing a Wilcoxon signed rank paired difference test with sample size 802 on the batch approach versus the incremental approach, gives test statistic 216153, leading one to conclude that the incremental approach is faster, with ****p-value 2.22×10^{-16}.

Scenario 3 – Deletions

The distributions of time for deletion before and after the split point for the pure batch approach are different, so the two sets of timings were each compared separately to the timings for the *split* operation (which can be used for both deletion before and after the split point).

A Mann Whitney U test (as before) on sums by batch of total times taken to delete the expression left of the split point using the different approaches yielded exactly the same results as before. This was also the case for deletion right of the split point, for the same reason. Based on the sample of 26 values, the incremental approach is 3762.528 (95% CI [3728.921, 3796.135]) faster than the batch approach for deletion of the part of the expression left of the split point, and 3751.262 (95% CI [3717.684, 3784.839]) faster than the batch approach for deletion of the part of the expression right of the split point.

Performing a Wilcoxon signed rank paired difference test with sample size 802 on the batch approach for deletion left of the split point, versus the incremental approach, gives test statistic 322003, leading one to conclude that the incremental approach is faster, with ****p-value 2.22×10^{-16}. The same test performed for deletion to the right of the split point yields exactly the same result, given that the incremental approach was faster in all cases.

5.3.6 Discussion

The results suggest that the incremental approach is significantly faster than the batch approach for simple editing operations.

For concatenation, which exercised the *merge* operation, the incremental approach took, on average, just over half the time required for the batch approach. This low ratio (in comparison with the other results) could be explained by the fact that the total time taken to perform the concatenation using the incremental approach includes the (batch) parsing of a token string that is on average half the length of the total expression length (in tokens). For simple deletion, the high ratios (over 3700:1) can be explained by the fact that for the batch approach, deletion will involve reparsing on average half of the expression, while the incremental approach requires just a single *split* operation.

The question arises of how much of the extra time taken up by the pure batch parsing approach is due to poor performance of the scanner. A fairer comparison would either take token strings rather than text strings as starting points for the time measurement, or involve a timing process that could determine how much of the batch parsing time was taken up by scanning. Either of these options would require changes to the batch parser: a task that was outside the scope of this project.

5.4 Conclusion

The algorithms for incremental parsing have been implemented in the context of an operator precedence parser for spoken mathematics, and evaluated for processing time for some simple editing operations in comparison to the batch parsing approach only being used, with results suggesting that incremental parsing is significantly faster.

The object of the programme to which the *TalkMaths* project belongs is to provide similar authoring and editing facilities for any structured document including, in particular, computer program code. While response times of the batch parser for dictation of relatively short mathematical expressions are mostly adequate, they do suggest that the pure batch parsing approach will be unsuitable for these larger structured documents. If the OP parser is to be used in this wider context, the incremental parsing algorithms will need to be applied to this expanded domain. The next section explores how that may be achieved.

Chapter 6

Incremental Operator Precedence Parsing for Spoken Programming Languages

6.1 Introduction

The previous two chapters were concerned with the incremental parsing algorithm in the context of spoken mathematics, which is the traditional domain of OP parsing. This chapter presents and discusses the requirements and limitations of applying OP parsing to other structured languages, in particular, programming languages, but does not explore the questions of display or alternative editing modes. (See for example Begel (2004) and Diekmann and Tratt (2013) for relevant discussion.)

The aim here is to represent any structured document using an OP grammar, so it may initially be parsed using a batch OP parser and, when edited, updated using an incremental OP parser. The tree structure should reflect the logical structure of the document, to facilitate display to the user in a natural way (and perhaps, in the future, form a source for translation). This requirement is not trivial: although it is possible to model a language using an OP grammar, as shown by Barenghi et al. (2015), the tree produced from parsing a program fragment is not amenable to being translated into display format with minimal processing.

The following sections describe the issues that need to be addressed to make this implementation of OP parsing (at both the batch and incremental stages) ready to handle programming languages.

6.2 Spoken Language for Programming

An obvious required extension is the enhancement of the spoken vocabulary. As well as handling typed lexemes, the parser needs to process spoken versions of programming constructs. The issues associated with spoken program code are well known and they are not considered here – see for example Begel (2004; 2005) – instead the question is raised of how lexemes from a spoken version of program code should map onto the internal representation and the extent to which they should be retained in the data structures.

Example

Take, for example, a scenario in which a user has set up a typical "while" loop using template lexemes "while", "do" and "end while". *TalkMaths* would create a composite node with appropriate delimiter tokens, plus the original lexemes. Suppose some time later the user wants to change the loop type to a "for" loop. Using a keyboard editor for a language that uses **{** and **}** delimiters for blocks, only the loop control part at the beginning would need to be changed – the closing delimiter could be left untouched – but in this scenario "end while" would have been recorded against the closing delimiter. A decision will need to be made on whether to update the delimiter to reflect the fact that the original utterance of "end while" is no longer appropriate, or if the spoken lexeme should be discarded at the time the loop was first dictated, to be replaced by a generic "end of block" delimiter. The second alternative may well be preferable; if some equivalent of the SPEech EDitor developed by Begel (2005) were required (providing an editing pane on the spoken version of the program code), the rendering of the code would employ some standard output version of the delimiters rather than what the user originally dictated.

The above paragraph covers just one example that applies to program code. One could argue that such problems occur even more frequently in the domain of spoken mathematics.

6.3 Extending Identifier Length in the Language

Currently, identifiers (names of variables) in *TalkMaths* are limited to single characters. While extending the length of identifier names in the language definition is no great challenge, complications arise for the speech recognition phase. In order to pass valid lexemes to the parser, a limited vocabulary for the ASR needs to be set up to filter out unwanted homophones or cases of poor speech recognition. This may be implemented using a utility such as Vocola[1] (that has been successfully used in the past to specify the restricted vocabulary used by *TalkMaths* (Attanayake, 2014)). For identifiers other than single characters to be used, not only does the dictation style of these need to be considered (see for example Begel (2005)) but also an appropriate mechanism needs to be developed to add the relevant words to the ASR vocabulary, along with the form in which they are to be passed to the parser. (Symbol or translation tables are not considered here – if the ASR interface can be relied upon to pass on strings that will be recognised as identifiers then no further work is necessary.)

6.4 Operator Precedence of Programming Constructs

An obvious question that arises here is what precedence should be given to constructs with which operator precedence is not normally associated, for example loops. Because the templates described by Attanayake (2014) are being used to represent distfix operators, constructs in the language may be divided into those used in expressions (as in our mathematical language) and those associated with control flow, such as statement lists, loops or conditionals. As templates (which in the current parser implementation may only represent closed distfix operators) are in effect bracketting constructs, they do not feature in the operator precedence table, so do not need to be considered here. Any control flow language construct modelled by a "true" operator will require an entry in the OP table, giving rise to the question of what precedence and associativity they should have.[2] These constructs are to be treated by the parser in the same way as templates, so given that their precedence is to impose a bracketting structure on the content, it must be lower than "normal" operators. In the simplest case, if all such operators can be treated as having left associativity, program control flow constructs

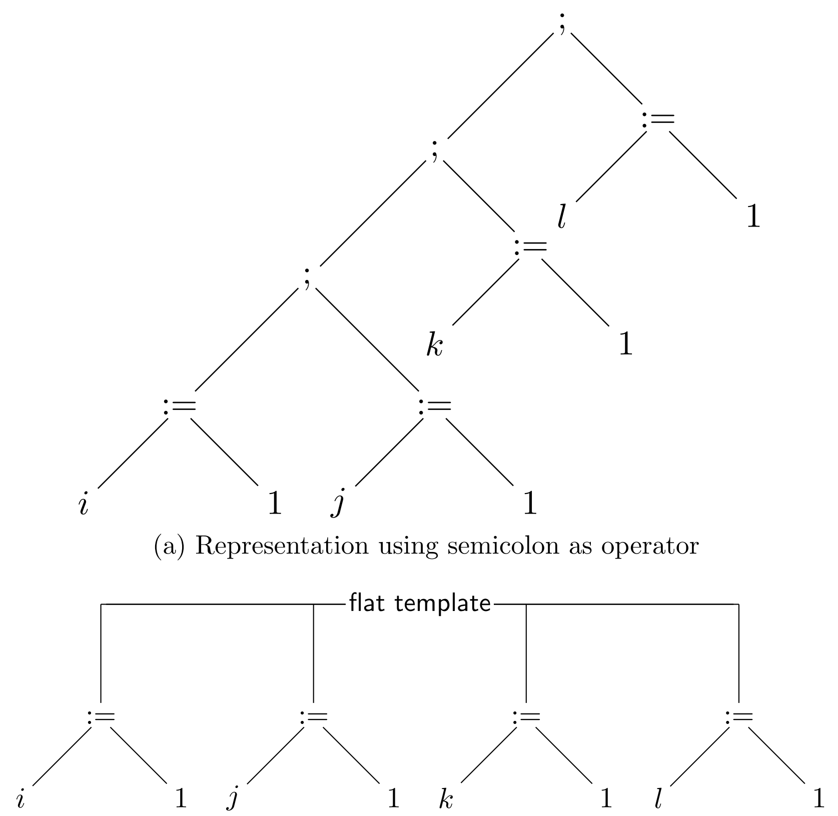
[1]See `vocola.net` web site for more information.

[2]Constructs for expressions will be treated in the usual way, having an entry in the OP table.

could be modelled as belonging to a single precedence class m, with $m \lessdot n$ for any other precedence class n, and $m \gtrdot m$.

6.5 Flat Representation of Lists

As mentioned by Wagner and Graham (1998), a drawback of using OP grammar to model languages that contain frequent sequences (specifically statements in the context of a programming language) is that such sequences are represented by deep one-sided subtrees rather than flat structures. For example, as illustrated in Figure 6.1, the sequence of statements `i := 1; j:= 1; k := 1; l := 1` would be represented by a tree of height 4, while its equivalent flat representation would require a height of only 2.



(a) Representation using semicolon as operator

(b) Equivalent flat representation

Figure 6.1: Alternative representations of a statement list

A possible optimisation may be to set up an additional type of template in the language definition, which specified at least one operand, with two or more operands being separated by instances of a specific delimiter. The language description would have to provide start and end delimiters, along with a separator delimiter, as a replacement for the two productions currently in the underlying grammar (one bracketing distfix operator to represent the main structure, plus one binary operator to separate the elements within the structure). Although that would work for many cases, a more elegant solution could be provided if the parsers permitted open mixfix operators to be used, where the optimisation would be provided by a multi-ary (taking a variable number of operands) operator to separate the items in the list. It is not clear at this stage, whether either of these optimisations would improve performance to the extent that the optimisation should be implemented in the batch and incremental parsers.

6.6 Removing the Unique Lexeme Restriction

By far the most challenging issue is the fact that the operator precedence parsing algorithm designed by Attanayake (2014) requires all lexemes in templates to be unique. For example, even if the two forms of integral were defined with different closing delimiters "end definite integral" and "end indefinite integral", they would share internal delimiter "with respect to". The definite integral would also share an internal delimiter "to" with a summation template. One could argue that the language should be designed to avoid such "clashes" by replacing phrase "from a to b" with some limit definition template, and phrase "*expression* with respect to *variable*" with an integral body template, but this would result in the definite integral template being of the form "**integral from** $\langle limit - definition\rangle\langle integral - body\rangle$ **end definite integral**", thus violating the requirements of an operator grammar.[3] Expecting the user to insert extra words into their dictation in order to make every lexeme unique would also not be acceptable, because of both the extra cognitive load and greatly increased "wordiness" of the dictation.

One solution would be to follow the example of Aasa (1995) by requiring the lexer to differentiate between such lexemes (for example, determine whether a "to" belongs to a sum, an integral, or some other construct), generating tokens that are unique to each template or operator. This would work perfectly for a batch parser, but complicate matters in the incremental parsing stage, as is illustrated by the following example.

[3]The original *TalkMaths* (Attanayake, 2014) language definition did not include such problematic templates.

6.6.1 Example: Changing the Nature of a For Loop

Suppose templates have been set up for two types of *for* loop[4]. The first one could correspond to a loop with header of form "**for** *item* **in** *list* **do** ... **end for**" and have token sequence (ITERATE_OVER, ITERATE_IN, ITERATE_DO, ITERATE_END). The second could represent a counted loop with header of form "**loop for** *counter* **from** *initial-value* **to** *end-value* ... **end for**" and have token sequence (FOR_LOOP, FOR_-FROM, FOR_TO, FOR_DO, FOR_END).

Suppose, then, that the user has dictated some code along the lines of,

"`for aWord in myList do` ... `end for`"

and wants to change this to

"`for i from 1 to length(myList) do` ... `end for`".

The sequence of editing actions would consist of splitting the corresponding tree somewhere around the "do" currently represented by token ITERATE_DO (let us, for argument's sake, say it was split after the "do"), replacing the tree fragment for

"`for aWord in myList do`"

to

"`for i from 1 to length(myList) do`"

and then merging the trees back together. The result would effectively be a bad match, with the tree for the loop body being headed by a node containing just ITERATE_END, and appearing subordinate to the node now identified as a FOR_LOOP composite node.

The incremental parser needs to recognise the `end for` working as an end delimiter for both loop varieties.

One could argue that if the split were made before the "do", the problem would be averted by the loop body being enclosed by lexemes `do` and `end for`, which would not change, but although the lexemes have remained the same, their tokens stem from different templates. The bad match would remain, because the original ITERATE_DO and ITERATE_END would not match the new FOR_LOOP, FOR_FROM and FOR_TO at the beginning of the loop header.

[4]This example does not represent a specific language – just one that that has two varieties of this kind of loop.

6.6.2 Avoiding Unnecessary Bad Matches

Unless a structured editor is to be used to change the nature of a composite node, an alternative way needs to be found to resolve this issue.

Assuming the token stream has been preserved or recreated, one approach would involve using an incremental lexer (as suggested by Aasa (1995)), but considering the potential distance (in the token stream) between the part of the composite node being changed and the rest of the lexemes representing its delimiters, this would be costly in terms of performance.

The preference here would be to permit a given token to be used by multiple mixfix operators, and for the parser to resolve the intrinsic ambiguity. Modifications to the batch OP parser are beyond the scope of this project; here the focus is on how the incremental parser could cope with the change.

6.6.3 Combining Composite Nodes where Unique Lexemes are not Required

This section gives an outline of modifications required to the incremental parsing algorithm in order to be able to drop the unique lexeme restriction.

The scenarios under which composite nodes may be combined (or not) are as follows.

No fit: Left and right nodes are from different templates and no template exists containing the concatenation of their delimiters in the same order. One node is placed subordinate to the other, as is currently done.

Perfect fit: Left node has delimiters $l_1, l_2, \ldots, l_m$, right node has delimiters $r_1, r_2, \ldots, r_n$ and there exists a template with complete delimiter list $l_1, l_2, \ldots, l_m, r_1, r_2, \ldots, r_n$. This includes the familiar case of both nodes belonging to the same template.

Fit with missing delimiters: As for perfect fit, but not all delimiters are present, either in the left node or right node or both. It is not possible to know how large any gaps are, but some template exists containing the concatenation of their extant delimiters, in the same order (though not necessarily consecutive). A new composite node may be formed, using the position of the delimiters in this template as reference. In the case of the node sequence matching more than one template, some decision will have to be made on which template to choose

– if the nodes do not both belong to the same template, then perhaps choose one of the template identifiers from the nodes being joined, or alternatively, the template that most closely fits the extant nodes, based on some edit distance measure for the joined node delimiters and the candidate templates.

Same templates with overlap: Both nodes currently belong to the same template, but left and right nodes have delimiters $d_1, d_2, \dots, d_k$ and $d_j, d_{j+1}, \dots, d_n$ respectively for some $j \leq k$.

Different originating templates that fit: This is the scenario that is needed to permit the user to change, for example, using our mathematical definition language example, a definite integral to an indefinite integral when these different structures are permitted to be described having at the end just "end integral".

Note, if both composite nodes have the same template currently associated with them, one could argue that an implementation could optimise the above categorisation by assuming a fit, behaving in the same way as the current algorithm.

As far as implementation is concerned, identification of subsequences of templates is non-trivial, because each delimiter may be represented by a number of different lexemes. (This would constitute another reason to follow the example of Begel (2005) in using a single "official" form for each token.) Assuming a mapping from multiple lexemes to a single token, an algorithm to identify the template for a combined node will work as follows.

- The language definition can be used to identify a list of templates associated with each token. This list would best be generated during batch parser initialisation (so it can be used by the batch parser to maximise efficiency), and remain available to the incremental parser.
- Identify first-cut possible templates from a union of sets of possible templates (using the list created at initialisation) for all tokens in a candidate combined node. For example, if the tokens in the candidate combined mode were FOR, IN and OF, the set of possible templates would consist of all templates that contain token FOR, token IN or token OF.
- Discard those templates for which the token sequence is not a subsequence of the template's delimiters.

- If more than one candidate is left, choose a template from one of the original nodes (if either is in the resulting set), or choose the template that minimises missing lexemes. If still more than one choice remains, then some other way of choosing a template would have to be employed.

6.6.4 Avoiding Overlaps

Another fact to be noted is that delimiters' lexemes do not necessarily consist of a single word. This raises the possibility that the concatenation of the words in two different delimiters might form a third delimiter. While it can be expected that if the word sequence is contained in an utterance it would probably be interpreted as the third delimiter (and so no conflict would arise), the question arises of whether a user would expect to be able to fit two fragments together with, for example, single word delimiters at the boundaries, and expect a new delimiter to be formed. A (somewhat artificial) illustration could be fragments "r **all** s" and "x **over** y" having the "s" and "x" removed and then joined together to make "r **all over** y", where "**all over**" appears as a single delimiter in some other template. Such an issue would be more likely to happen if using a spoken language editor pane; while it may be worth adopting an approach such as that described by Barenghi et al. (2015) (perhaps trying alternate versions of the incremental parse and choosing the one that produces the best composite node fit), the low frequency of edits like these may not warrant the effort.

6.7 Conclusion

This chapter develops extensions that can be applied to both the batch parser developed by Attanayake (2014) and the incremental parser described in Chapter 4, in order to handle a programming language or other structured content. This can provide an environment in which meaningfully formatted display of such content can be updated rapidly in response to spoken editing commands. The main idea is to describe how the algorithm needs to be changed to handle programming languages. These modifications could also improve the usability of the spoken mathematics language when applied to *TalkMaths*.

While not implemented in the scope of this thesis, they would not present any major problems apart form the challenge of integrating them with state-of-the-art proprietary ASR software.

Chapter 7

Conclusions and Further Work

7.1 Thesis Summary

In summary, this thesis has contributed to the field of HCI and incremental parsing algorithms as follows. It has identified principles to follow when designing speech controlled editors for structured content, extended an incremental parsing algorithm to facilitate one of these principles, and described modifications required to that algorithm for it to be applied to typical programming languages.

Having compiled a list of INUI design principles in Chapter 2, speed of update was identified as an important factor in providing a more natural style of interaction for speech controlled editing of structured content. To facilitate this, in Chapter 4 the incremental OP parsing algorithm of Heeman (1990) was improved in terms of efficiency and correctness, and extended to handle the distfix operators developed by Attanayake (2014) for modelling mathematical expressions using an OP grammar. The chapter finished by finding that the time complexities of splitting and concatenating expressions using the incremental parsing algorithms are better than using the batch parser equivalents. Chapter 5 described the implementation of this algorithm as part of the *TalkMaths* parser (that as part of this project was improved to give faster response times), and in experimental tests found the incremental parser to perform significantly better in terms of execution times than reparsing the entire content in response to editing actions.

The application of both the batch (Attanayake, 2014) and incremental parsers to program code was investigated, with the conclusion that use of these is feasible with the following modifications described in Chapter 6.

- The restriction that no tokens should appear in more than one distfix operator should be lifted.
- The identification of composite nodes that may be combined should no longer be restricted to those that represented the same distfix operator at creation time; instead, their individual token sequences should be assessed after which they may be combined or staggered as appropriate.
- A mechanism needs to be set up that permits identifiers longer than a single character to be passed to the batch parser. This would be the responsibility of the interface between the ASR software and *TalkMaths*.

7.2 Review of Aim and Objectives

The aim of the project was to contribute to research on facilitating speech control of editors of structured content such as mathematics or computer program code. This has been achieved through the following.

- Based on a literature review of natural user interfaces and intuitive interfaces, a list of general INUI design principles was compiled, which along with issues specific to spoken mathematics, informed a set of guidelines to follow when designing a front end for a system such as *TalkMaths*. This paves the way for usability research to test the effectiveness of such a user interface.
- An incremental parsing algorithm was designed that acts on abstract syntax trees representing expressions built using an OP parser and modelling operators with more than one operand using the constructs designed by Attanayake (2014). This was implemented to work with Attanayake's batch parser, tested for correctness, and compared with this batch parser in terms of response time for handling changes to mathematical expressions.
- The modifications necessary for the parsers to be used for computer program code as well as mathematical expressions were identified, and a high-level design for the required modifications to the incremental parsing algorithm were provided.

7.3 Limitations and Reflections on Thesis

The HCI aspect of this thesis focussed on speech for input and visual output. It would be of interest to explore how INUI principles could be applied to speech output, and how this could be combined with speech input in the speech-controlled editor context.

The question of whether it is appropriate to continue modelling lists using binary separator operators was left open, as it has not been determined whether implementation of operators that take a variable number of operands would in fact improve performance of such a system.

Although the incremental OP parser developed in this thesis was compared with its batch equivalent in the context of mathematical expressions, a type of material that is readily represented using an OP language, it is not known how well it would perform in comparison with a more traditional incremental parser when applied to an environment in which speech control is used to edit computer program code, such as that developed by Begel (2005).

7.4 Suggestions for Further Work

The following are envisaged as next steps for the *TalkMaths* research programme.

- Design, implement and evaluate a front-end for *TalkMaths* that adheres to the INUI principles for speech editors, and which makes use of both the batch and incremental parsing facilities. This may then be tested for usability.
- Modify the batch and incremental parsers as described in this thesis to enable them to be applied to programming languages, along with a simple speech-controlled interface that will permit multiple character variables.
- Describe a spoken version of an existing programming language in the language definition used by the parsers and investigate the issues involved in using this architecture to create and edit computer program code.

The results of this study may then be used to guide production of a version of *TalkMaths* that may be used as a practical program code editor, to the benefit of users whose circumstances dictate that they may not be able to use keyboard and mouse to write computer program code.

Bibliography

Aasa, A. (1995), ‘Precedences in specifications and implementations of programming languages’, *Theoretical Computer Science* **142**(1), 3–26.

Aho, V. A., Sethi, R. and Ullman, J. D. (2003), *Compilers : Principlies, Techniques, and Rools*, international edn, Prentice-Hall Inc, Upper Saddle River, N.J.

Aho, V A; Ullman, J. D. (1972), *The Theory of Parsing, Translation and Compiling*, Vol. 1: Parsing, Prentice-Hall Inc, Upper Saddle River, N.J.

Asikhia, O. K., Hicks, Y., Setchi, R. and Walters, A. (2015), ‘Conceptual Framework for Evaluating Intuitive Interaction Based on Image Schemas’, *Interacting with Computers* **27**(3), 287–310.

Attanayake, D. R. (2014), Statistical Language Modelling and Novel Parsing Techniques for Enhanced Creation and Editing of Mathematical E-Content Using Spoken Input, PhD thesis, Kingston University, Kingston-upon-Thames, UK. Accessed on 30/05/2015.

Attanayake, D. R., Denholm-Price, J., Hunter, G., Pfluegel, E. and Wigmore, A. (2012), Intelligent assistive interfaces for editing mathematics, *in* ‘Workshop Proceedings of the 8th International Conference on Intelligent Environments’, Vol. 13, IOS Press, pp. 286–297.

Bahlke, R. and Snelting, G. (1986), ‘The psg system: from formal language definitions to interactive programming environments’, *ACM Trans. Program. Lang. Syst.* **8**(4), 547–576.

Ballance, R. A., Butcher, J. and Graham, S. L. (1988), Grammatical abstraction and incremental syntax analysis in a language-based editor, *in* ‘Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation’, PLDI ’88, ACM, New York, NY, USA, pp. 185–198.

Ballance, R. A., Graham, S. L. and Van De Vanter, M. L. (1992), 'The Pan language-based editing system', *ACM Transactions on Software Engineering and Methodology (TOSEM)* **1**(1), 95–127.

Barenghi, A., Reghizzi, S. C., Mandrioli, D., Panella, F. and Pradella, M. (2015), 'Parallel parsing made practical', *Science of Computer Programming* **112**(3), 195 – 226.

Barenghi, A., Reghizzi, S. C., Mandrioli, D. and Pradella, M. (2013), 'Parallel parsing of operator precedence grammars', *Information Processing Letters* **113**(7), 245–249.

Begel, A. B. (2004), Spoken language support for software development, *in* 'IEEE Symposium on Visual Languages and Human Centric Computing', pp. 271 – 272.

Begel, A. B. (2005), Spoken Language Support for Software Development, PhD thesis, University of California, Berkeley. Accessed 23/04/2019.
URL: *http://digitalassets.lib.berkeley.edu/techreports/ucb/text/EECS-2006-8.pdf*

Begel, A; Graham, S. L. (2006), 'XGLR – an algorithm for ambiguity in programming languages', *Science of Computer Programming - The Fourth Workshop on Language Descriptions, Tools, and Applications* **61**(3), 211–227.

Bernardy, J.-P. (2009), Lazy functional incremental parsing, *in* 'Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell', pp. 49–60.

Blackler, A. L. and Hurtienne, J. (2007), 'Towards a unified view of intuitive interaction: definitions, models and tools across the world', *MMI-interaktiv* **13**(2007), 36–54.

Cook, P. and Welsh, J. (2001), 'Incremental parsing in language-based editors: user needs and how to meet them', *Software: Practice and Experience* **31**(15), 1461–1486.

Cross, J. and Huang, L. (2016), 'Incremental Parsing with Minimal Features Using Bi-Directional LSTM', *arXiv e-prints* p. arXiv:1606.06406.

Danielsson, N. and Norell, U. (2011), Parsing mixfix operators, *in* S.-B. Scholz and O. Chitil, eds, 'Implementation and Application of Functional Languages', Vol. 5836 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 80–99.

Degano, P., Mannucci, S. and Mojana, B. (1988), 'Efficient incremental LR parsing for syntax-directed editors', *ACM Trans. Program. Lang. Syst.* **10**(3), 345–373.

Desilets, A. (2001), 'Voicegrip: A tool for programming-by-voice', *International Journal of Speech Technology* **4**(2), 103–116.

Desilets, A., Fox, D. C. and Norton, S. (2006), Voicecode: an innovative speech interface for programming-by-voice, *in* 'CHI '06 Extended Abstracts on Human Factors in Computing Systems', ACM, New York, NY, USA, pp. 239 – 242.

Diekmann, L. and Tratt, L. (2013), Parsing composed grammars with language boxes, *in* 'Workshop on Scalable Language Specifications'.

Diekmann, L. and Tratt, L. (2019), Default disambiguation for online parsers, *in* 'Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering', Association for Computing Machinery, pp. 88–99.

Dubroy, P. and Warth, A. (2017), Incremental packrat parsing, *in* 'Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering', SLE 2017, ACM, New York, NY, USA, pp. 14–25.

Earley, J. and Caizergues, P. (1972), 'A method for incrementally compiling languages with nested statement structure', *Commun. ACM* **15**(12), 1040–1044.

Elepfandt, M. and Grund, M. (2012), Move it there, or not?: The design of voice commands for gaze with speech, *in* 'Proceedings of the 4th Workshop on Eye Gaze in Intelligent Human Machine Interaction', Gaze-In '12, ACM, New York, NY, USA, pp. 12:1–12:3.

Fateman, R. (2013), How can we speak math? Accessed on 21/05/2013.
URL: *http://http.cs.berkeley.edu/ fateman/papers/speakmath.pdf*

Fauconnier, G. and Turner, M. (2008), *The way we think: Conceptual blending and the mind's hidden complexities*, Basic Books, New York.

Ferro, M. V. and Dion, B. A. (1994), Efficient incremental parsing for context-free languages, *in* 'Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)', IEEE, pp. 241–252.

Fjeld, M., Bichsel, M. and Rauterberg, M. (1998), Build-it: an intuitive design tool based on direct object manipulation, *in* 'Gesture and Sign Language in Human-Computer Interaction', Springer, Berlin & Heidelberg, pp. 297–308.

Fjeld, M., Bichsel, M. and Rauterberg, M. (1999), 'Build-it: a brick-based tool for direct interaction', *Engineering Psychology and Cognitive Ergonomics (EPCE)* **4**, 205–212.

Ford, B. (2002), Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl, *in* 'Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming', ICFP '02, ACM, New York, NY, USA, pp. 36–47.

Gafter, N. M. (1990), Parallel incremental compilation, Technical report, DTIC Document.

Ghezzi, C. and Mandrioli, D. (1977), A note on deterministic parsing from left to right and from right to left, Technical Report Int. Rep. IEEEPM 77-11, Istituto di Elettrotecnica ed Elettronica, Politeenico di Milano, Italy.

Ghezzi, C. and Mandrioli, D. (1979), 'Incremental parsing', *ACM Transactions on Programming Languages and Systems* **1**(1), 58–70.

Ghezzi, C. and Mandrioli, D. (1980), 'Augmenting parsers to support incrementality', *Journal of the ACM* **27**(3), 564–579.

Ghosh, S., Shruthi, C. S., Bansal, H. and Sethia, A. (2017), What is user's perception of naturalness? An exploration of natural user experience, *in* R. Bernhaupt, G. Dalvi, A. Joshi, D. K. Balkrishan, J. O'Neill and M. Winckler, eds, 'Human-Computer Interaction - INTERACT 2017', Springer International Publishing, Cham, pp. 224–242.

Gordon, B. M. and Luger, G. F. (2012), English for spoken programming, *in* 'The 6th International Conference on Soft Computing and Intelligent Systems, and The 13th International Symposium on Advanced Intelligence Systems', IEEE, pp. 16–20.

Hacker, W. (1994), 'Action regulation theory and occupational psychology: Review of German empirical research since 1987.', *German Journal of Psychology* **18**(2), 91–120.

Heeman, F. C. (1990), 'Incremental parsing of expressions', *Journal of Systems and Software* **13**, 55–69.

Heering, J., Klint, P. and Rekers, J. (1994), 'Lazy and incremental program generation', *ACM Transactions on Programming Languages and Systems (TOPLAS)* **16**(3), 1010–1023.

Hefny, A., Hassan, H. and Bahgat, M. (2011), Incremental combinatory categorial grammar and its derivations, *in* A. F. Gelbukh, ed., 'Computational Linguistics and Intelligent Text Processing', Springer Berlin Heidelberg, pp. 96–108.

Howell, R. R. (2008), On asymptotic notation with multiple variables, Technical report, Department of Computing and Information Sciences, Kansas State University, Manhattan, KS.

Huang, Liang; Sagae, K. (2010), Dynamic programming for linear-time incremental parsing, *in* 'Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics', pp. 1077–1086.

Hurtienne, J. and Israel, J. H. (2007), Image schemas and their metaphorical extensions: intuitive patterns for tangible interaction, *in* 'Proceedings of the 1st International Conference on Tangible and Embedded Interaction', ACM, ACM Press, pp. 127–134.

Isaac, M. J., Pfluegel, E., Hunter, G. and Denholm-Price, J. (2015), Intuitive NUIs for speech editing of structured content (work in progress), *in* '26th Annual Workshop of Psychology of Programming Interest Group, PPIG 2015', pp. 1–5.

Jalili, F. and Gallier, J. H. (1982), Building friendly parsers, *in* 'Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', POPL '82, ACM, New York, NY, USA, pp. 196–206.

Jetter, H.-C., Reiterer, H. and Geyer, F. (2014), 'Blended interaction: understanding natural human-computer interaction in post-WIMP interactive spaces', *Personal and Ubiquitous Computing* **18**(5), 1139–1158.

Kahrs, M. (1979), Implementation of an interactive programming system, *in* 'ACM SIGPLAN Notices', Vol. 14, ACM, pp. 76–82.

Kaiser, G. E. and Kant, E. (1985), 'Incremental parsing without a parser', *Journal of Systems and Software* **5**(2), 121–144.

Kato, Y. and Matsubara, S. (2015), 'Identifying nonlocal dependencies in incremental parsing', *IEICE Transactions on Information and Systems* **E98D**(4), 994–998.

Kaur, M., Tremaine, M., Huang, N., Wilder, J., Gacovski, Z., Flippo, F. and Mantravadi, C. S. (2003), Where is it? Event synchronization in gaze-speech input systems, *in* 'Proceedings of the 5th International Conference on Multimodal Interfaces', ACM, pp. 151–158.

Kirsh, D. and Maglio, P. (1994), 'On distinguishing epistemic from pragmatic action', *Cognitive Science* **18**(4), 513–549.

Koehn, A. and Menzel, W. (2014), Incremental predictive parsing with turboparser, *in* 'Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics', Vol. 2, Assoc Computat Linguist; Baidu; Bloomberg; Google; Microsoft; Nuance; Yahoo Labs; Informat Sci Inst, Xerox Res Ctr Europe; Brandeis Univ; Facebook; Yandex; Amazon Com; IBM Watson; Johns Hopkins Univ; A9; AI@ISI; Xerox, Assoc Computational Linguistics – ACL, pp. 803–808.

Konstas, J. and Keller, F. (2015), Semantic role labeling improves incremental parsing, *in* 'Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing', Vol. 1, Assoc Computat Linguist; Asian Federat Nat Language Proc; CreditEase; Baidu; Tencent; Alibaba Grp; Samsung; Microsoft; Google; Facebook; SinoVoice; Huawei; Nuance; Amazon; Voicebox Technologies; Baobab; Sogou, Assoc Computational Linguistics – ACL, pp. 1191–1201.

LaLonde, W. R. and des Rivieres, J. (1981), 'Handling operator precedence in arithmetic expressions with tree transformations', *ACM Trans. Program. Lang. Syst.* **3**, 83–103.

Larchêveque, J.-M. (1995), 'Optimal incremental parsing', *ACM Transactions on Programming Languages and Systems* **17**, 1–15.

Li, W. X. (1996), 'Building efficient incremental LL parsers by augmenting LL tables and threading parse trees', *Computer Languages* **22**(4), 225–35.

Lindén, G. (1994), Incremental updates in structured documents, Licentiate thesis, University of Helsinki.

MacLean, S. and Labahn, G. (2013), 'A new approach for recognizing handwritten mathematics using relational grammars and fuzzy sets', *International Journal on Document Analysis and Recognition* **16**(2), 139–163.

Maglio, P. P., Matlock, T., Campbell, C. S., Zhai, S. and Smith, B. A. (2000), Gaze and speech in attentive user interfaces, *in* 'Advances in Multimodal Interfaces – ICMI 2000', Springer, pp. 1–7.

Medina-Mora, R. and Feiler, P. (1981), 'An incremental programming environment', *Software Engineering, IEEE Transactions on* **SE-7**(5), 472–482.

Morris, J. M. and Schwartz, M. D. (1981), 'The design of a language-directed editor for block-structured languages', *ACM SIGOA Newsletter* **2**(1-2), 28–33.

Murching, A. M., Prasad, Y. and Srikant, Y. (1990), 'Incremental recursive descent parsing', *Computer Languages* **15**(4), 193 – 204.

Naumann, A., Hurtienne, J., Israel, J. H., Mohs, C., Kindsmüller, M. C., Meyer, H. A. and Hußlein, S. (2007), Intuitive use of user interfaces: defining a vague concept, *in* 'Engineering Psychology and Cognitive Ergonomics', Springer, pp. 128–136.

Petrone, L. (1995), Reusing batch parsers as incremental parsers, *in* P. Thiagarajan, ed., 'Foundations of Software Technology and Theoretical Computer Science', Vol. 1026 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 111–123.

Pfluegel, E., Wigmore, A. and Attanayake, D. (2011), Speech-based editing paradigms for mathematical content, Technical Report KU-CISM-2011-1, Kingston University.

Rauterberg, M. (1999), 'From gesture to action: Natural user interfaces', *Mens-Machine Interactive: Diesrede 1999* pp. 15–25.

Reghizzi, S. C. and Mandrioli, D. (2012), 'Operator precedence and the visibly push-down property', *Journal of Computer and System Sciences* **78**(6), 1837–1867.

Rekers, J. G. (1992), Parser generation for interactive environments, PhD thesis, University of Amsterdam.

Rekers, J. and Koorn, W. (1991), 'Substring parsing for arbitrary context-free grammars', *ACM SIGPLAN Notices* **26**(5), 59–66.

Schwartz, M. D., Delisle, N. M. and Begwani, V. S. (1984), 'Incremental compilation in Magpie', *SIGPLAN Not.* **19**(6), 122–131.

Shilling, J. (1993), 'Incremental LL(1) parsing in language-based editors', *Software Engineering, IEEE Transactions on* **19**(9), 935–940.

Sibert, L. E. and Jacob, R. J. (2000), Evaluation of eye gaze interaction, *in* 'Proceedings of the SIGCHI Conference on Human Factors in Computing Systems', ACM, pp. 281–288.

Sijm, M. P. (2019), Incremental scannerless generalized LR parsing, *in* 'Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity', Association for Computing Machinery, pp. 54–56.

Soiffer, N. (1991), The design of a user interface for computer algebra systems, Technical report, University of California, Berkeley.

Stedmon, A. W., Patel, H., Sharples, S. C. and Wilson, J. R. (2011), 'Developing speech input for virtual reality applications: A reality based interaction approach', *International Journal of Human-Computer Studies* **69**(1), 3–8.

Teitelbaum, T. and Reps, T. (1981), 'The cornell program synthesizer: a syntax-directed programming environment', *Commun. ACM* **24**(9), 563–573.

University of Oxford Mathematical Institute (2016), 'Solutions for admissions test in Mathematics, Computer Science and Joint Schools'. Accessed 23/04/2019.
URL: *https://www.maths.ox.ac.uk/system/files/attachments/websolutions15_1.pdf*

Vieira, D., Freitas, J. a. D., Acartürk, C., Teixeira, A., Sousa, L., Silva, S., Candeias, S. and Dias, M. S. (2015), "Read that article": Exploring synergies between gaze and speech interaction, *in* 'Proceedings of the 17th International ACM SIGACCESS Conference on Computers & Accessibility', ASSETS '15, ACM, New York, NY, USA, pp. 341–342.

Wagner, T. A. and Graham, S. L. (1998), 'Efficient and flexible incremental parsing', *ACM Transactions on Programming Languages and Systems* **20**, 980 – 1013.

Wagner, Tim A; Graham, S. L. (1997), Incremental analysis of real programming languages, *in* 'Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation', ACM, pp. 31–43.

Wegman, M. (1980), Parsing for structural editors, *in* '21st Annual Symposium on Foundations of Computer Science, 1980', pp. 320 –327.

Wigdor, D., Fletcher, J. and Morrison, G. (2009), Designing user interfaces for multi-touch and gesture devices, *in* 'CHI '09 Extended Abstracts on Human Factors in Computing Systems', CHI EA '09, ACM, New York, NY, USA, pp. 2755–2758.

Wigdor, D. and Wixon, D. (2011), *Brave NUI world: designing natural user interfaces for touch and gesture*, Elsevier Science Inc., London.

Wigmore, A. M. (2011), Speech-Based Creation and Editing of Mathematical Content, PhD thesis, Kingston University, Kingston-upon-Thames, UK.

Wigmore, A. M., Hunter, G. J., Pfluegel, E. and Denholm-Price, J. (2009), TalkMaths: A speech user interface for dictating mathematical expressions into electronic documents, *in* '2nd ISCA Workshop of Speech and Language Technology in Education (SLaTE 2009)', International Speech Communication Association (ISCA), pp. 3–4.

Wigmore, A. M., Hunter, G., Pfluegel, E., Denholm-Price, J. and Binelli, V. (2009), ‘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* **28**(2), 177–189.

Yang, W. (1993), ‘An incremental LL(1) parsing algorithm’, *Information Processing Letters* **48**(2), 67 – 72.

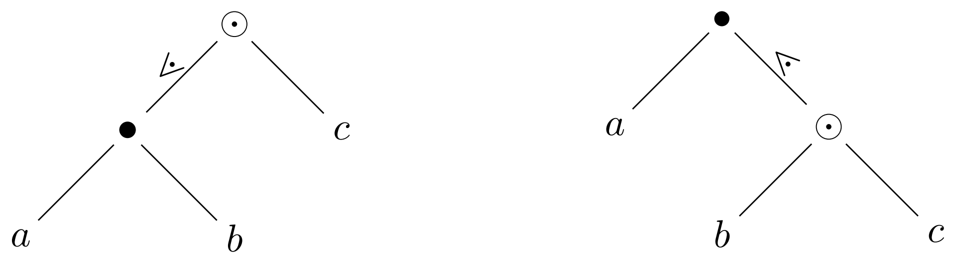
Yang, W. (1994), Incremental LR parsing, *in* ‘1994 International Computer Symposium Conference Proceedings’, pp. 577–83.

Yeh, D. and Kastens, U. (1988), ‘Automatic construction of incremental LR(1) parsers’, *SIGPLAN Notices* **23**(3), 33–42.

Appendix A

Twiddle Operations

All operations, including the original LEFT-SUBORDINATE and RIGHT-SUBORDINATE developed by LaLonde and des Rivieres (1981)[1], are shown in Figures A.1 to A.7.

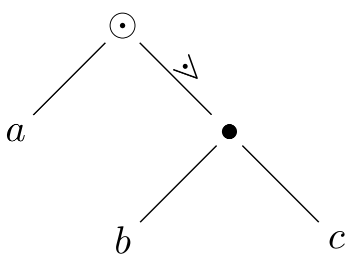


(a) Before RIGHT-SUBORDINATE (b) After RIGHT-SUBORDINATE

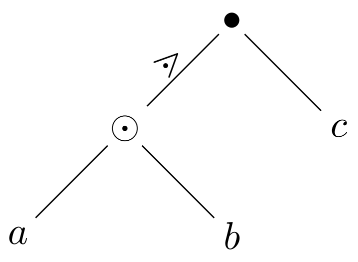
Figure A.1: RIGHT-SUBORDINATE tree transformation (LaLonde and des Rivieres, 1981) $a \bullet b \odot c$, where $\odot \gtrdot \bullet$, also known as *twiddleleft* (Kaiser and Kant, 1985)

There are two cases where a twiddle can not be used to correct violation of operator precedence. The only way of rearranging the tree in Figure A.8a so that operator precedence is preserved would change the expression to $a \odot b \bullet$, with $\bullet$ applying to the entire expression rather than just a. In Figure A.8b, rearrangement would have a similar effect, with the expression being changed to $\bullet\ a \odot b$.

[1]This is to save the reader the trouble of having to obtain a copy of the LaLonde and des Rivieres (1981) paper, which is not available electronically.

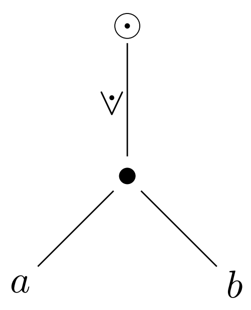


(a) Before LEFT-SUBORDINATE

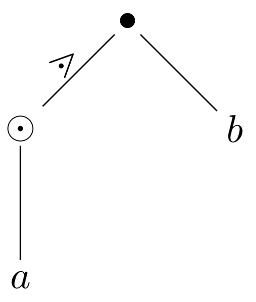


(b) After LEFT-SUBORDINATE

Figure A.2: LEFT-SUBORDINATE tree transformation (LaLonde and des Rivieres, 1981) $a \odot b \bullet c$, where $\odot \gtrdot \bullet$, also known as *twiddleright* (Kaiser and Kant, 1985)

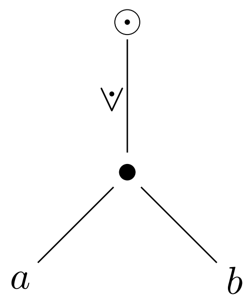


(a) Before *twiddlePrefixLeft*

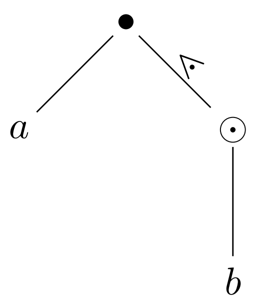


(b) After *twiddlePrefixLeft*

Figure A.3: *twiddlePrefixLeft* tree transformation for representation of $\odot a \bullet b$ where $\odot$ is a prefix operator, $\bullet$ is binary, and $\odot \gtrdot \bullet$

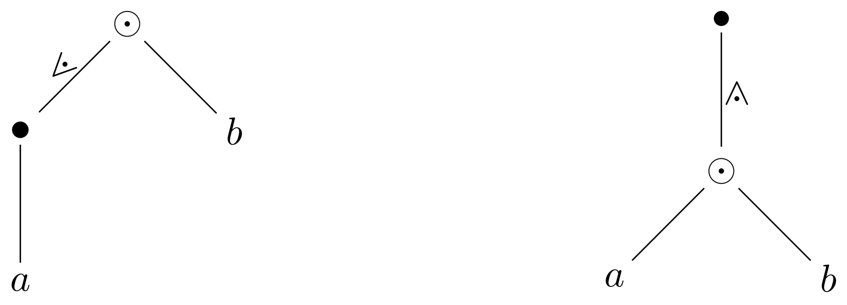


(a) Before *twiddlePostfixRight*



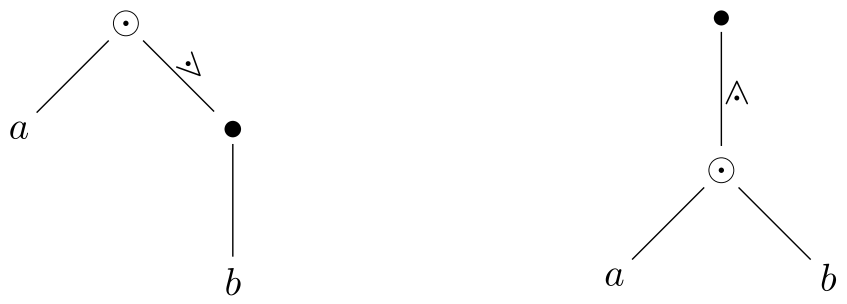
(b) After *twiddlePostfixRight*

Figure A.4: *twiddlePostfixRight* tree transformation for representation of $a \bullet b \odot$ where $\odot$ is a postfix operator, $\bullet$ is binary, and $\odot \gtrdot \bullet$



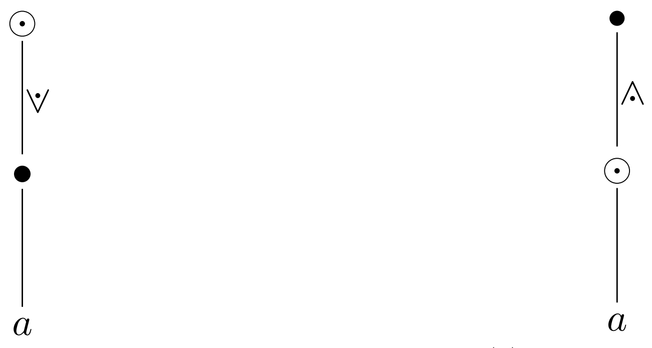
(a) Before *twiddleLeftPrefix* (b) After *twiddleLeftPrefix*

Figure A.5: *twiddleLeftPrefix* tree transformation for representation of $\bullet a \odot b$ where $\bullet$ is a prefix operator, $\odot$ is binary, and $\odot \gtrdot \bullet$



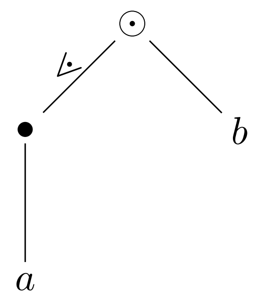
(a) Before *twiddleRightPostfix* (b) After *twiddleRightPostfix*

Figure A.6: *twiddleRightPostfix* tree transformation for representation of $a \odot b\bullet$ where $\bullet$ is a postfix operator, $\odot$ is binary, and $\odot \gtrdot \bullet$

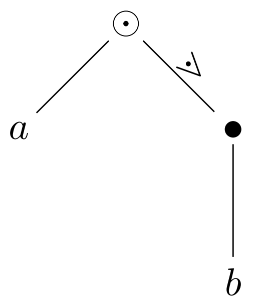


(a) Before *twiddleUnaries* (b) After *twiddleUnaries*

Figure A.7: *twiddleUnaries* tree transformation for representation of $\odot a\bullet$ where $\bullet$ is a postfix operator, $\odot$ is prefix (or $\bullet$ is prefix and $\odot$ postfix), and $\odot \gtrdot \bullet$



(a) $a \bullet \odot b$, $\bullet$ postfix



(b) $a \odot \bullet b$, $\bullet$ prefix

Figure A.8: Situations where a twiddle cannot be made without changing the intended meaning, where $\odot \gtrdot \bullet$

Appendix B

Fragments of *opmerge* Algorithm

Note, where reference is made to whether or not a node has a child, this refers to a non-empty child.

Algorithm 6 Fragment of *opmerge*: $c_L Prefix\_c_R Prefix$

```
1: if $c_L$ has a child then
2:     push $(c_L, 1)$ onto $P$
3:     set $c_L$ to decoupled child of $c_L$
4: else
5:     if $c_R \lessdot c_L$ then
6:         $c_R$ should go under $c_L$ but deal with potential OP violation
7:     else
8:         place $c_R$ under $c_L$
9:     end if
10:    set $workingDown$ to false, $s$ to $c_L$
11: end if
```

Algorithm 7 Fragment of *opmerge*: $c_L Prefix\_c_R Postfix$

```
if $c_R \gtrdot c_L$ then
    if $c_L$ has a child then
        push $(c_L, 1)$ onto $P$
        set $c_L$ to decoupled child of $c_L$
    else
        place $c_R$ under $c_L$
        set $workingDown$ to false, $s$ to $c_L$
    end if
else
    if $c_R$ has a child then
        push $(c_R, 1)$ onto $P$
        set $c_R$ to decoupled child of $c_R$
    else
        place $c_L$ under $c_R$
        set $workingDown$ to false, $s$ to $c_R$
    end if
end if
```

Algorithm 8 Fragment of *opmerge*: $c_L Prefix\_c_R Binary$

```
if $c_R \gtrdot c_L$ then
    if $c_L$ has a child then
        push $(c_L, 1)$ onto $P$
        set $c_L$ to decoupled child of $c_L$
    else
        place $c_R$ under $c_L$
        set $workingDown$ to false, $s$ to $c_L$
    end if
else
    if $c_R$ has a left-hand child then
        push $(c_R, 1)$ onto $P$
        set $c_R$ to decoupled left-hand child of $c_R$
    else
        make $c_L$ the left-hand child of $c_R$
        set $workingDown$ to false, $s$ to $c_R$
    end if
end if
```

Algorithm 9 Fragment of *opmerge*: $c_L Prefix\_c_R Other$

```
1: if $c_L$ has a child then
2:    if $c_R$ has a child then
3:       push $(c_L, 1)$ onto $P$
4:       set $c_L$ to decoupled child of $c_L$
5:    else
6:       set $workingDown$ to false, $s$ to $c_L$
7:    end if
8: else
9:    place $c_R$ under $c_L$
10:   set $workingDown$ to false, $s$ to $c_L$
11: end if
```

Algorithm 10 Fragment of *opmerge*: $c_L Postfix\_c_R Other$

```
1: set $workingDown$ to false
2: if $c_R$ is empty then
3:    set $s$ to $c_L$
4: else
5:    set $s$ to a tree with default operator as root and children $c_L$ and $c_R$
6: end if
```

Algorithm 11 Fragment of *opmerge*: $c_L Postfix\_c_R Postfix$

```
1: set $workingDown$ to false
2: if $c_R$ has a child then
3:    create a tree $j$ with default operator as root and $c_L$ as its left-hand child
4:    if $c_R \lessdot$ default operator then
5:       remove the child of $c_R$ and make it the right-hand child of $j$
6:       make $j$ the child of $c_R$
7:       set $s$ to $c_R$
8:    else
9:       place $c_R$ under $j$ as right-hand child
10:      set $s$ to $j$
11:   end if
12: else
13:   if $c_L \lessdot c_R$ then
14:      $c_L$ should go under $c_R$ but deal with potential OP violation
15:   else
16:      place $c_L$ under $c_R$
17:   end if
18:   set $s$ to $c_R$
19: end if
```

Algorithm 12 Fragment of *opmerge*: $c_LPostfix\_c_RBinary$

```
 1: if $c_R$ has a left-hand child then
 2:    push $(c_R, 1)$ onto $P$
 3:    set $c_R$ to decoupled left-hand child of $c_R$
 4: else
 5:    place $c_L$ under $c_R$ as left-hand child
 6:    set $workingDown$ to false, $s$ to $c_R$
 7: end if
```

Algorithm 13 Fragment of *opmerge*: $c_LBinary\_c_RPostfix$

```
 1: if $c_L \lessdot c_R$ then
 2:    if $c_L$ has a right-hand child then
 3:       push $(c_L, 2)$ onto $P$
 4:       set $c_L$ to decoupled right-hand child of $c_L$
 5:    else
 6:       place $c_R$ under $c_L$ as right-hand child
 7:       set $workingDown$ to false, $s$ to $c_L$
 8:    end if
 9: else
10:    if $c_R$ has a child then
11:       push $(c_R, 1)$ onto $P$
12:       set $c_R$ to the decoupled child of $c_R$
13:    else
14:       place $c_L$ under $c_R$
15:       set $workingDown$ to false, $s$ to $c_R$
16:    end if
17: end if
```

Algorithm 14 Fragment of *opmerge*: $c_LBinary\_c_RBinary$ – the case dealt with by Heeman (1990)

```
 1: if $c_L \lessdot c_R$ then
 2:    if $c_L$ has a right-hand child then
 3:       push $(c_L, 2)$ onto $P$
 4:       st $c_L$ to decoupled right-hand child of $c_L$
 5:    else
 6:       place $c_R$ under $c_L$ as right-hand child
 7:       set $workingDown$ to false, $s$ to $c_L$
 8:    end if
 9: else
10:    if $c_R$ has a left-hand child then
11:       push $(c_R, 1)$ onto $P$
12:       set $c_R$ to the decoupled left-hand child of $c_R$
13:    else
14:       place $c_L$ under $c_R$ as left-hand child
15:       set $workingDown$ to false, $s$ to $c_R$
16:    end if
17: end if
```

Algorithm 15 Fragment of *opmerge*: $c_L Other\_c_R Postfix$

```
1: if $c_R$ has a child then
2:     push $(c_R, 1)$ onto $P$
3:     set $c_R$ to the decoupled child of $c_R$
4: else
5:     place $c_L$ under $c_R$
6:     set $workingDown$ to false, $s$ to $c_R$
7: end if
```

Algorithm 16 Fragment of *opmerge*: $c_L Other\_c_R Binary$

```
1: if $c_R$ has a left-hand child then
2:     push $(c_R, 1)$ onto $P$
3:     set $c_R$ to the decoupled left-hand child of $c_R$
4: else
5:     place $c_L$ under $c_R$ as left-hand child
6:     set $workingDown$ to false, $s$ to $c_R$
7: end if
```

Appendix C

Sample Data Used for Evaluation

Expression as string	**Represents**
x = 4(n+1) squared - 3	$x = 4(n+1)^2 - 3$
(sine (x) end sine + 1) squared greater or equal 0	$(\sin(x) + 1)^2 \geq 0$
cosine x end cosine squared + sine x end sine squared equals 1	$\cos x^2 + \sin x^2 = 1$
g(f(A)) = fraction A to the power of 6 over 3ˆ4	$g(f(A)) = \frac{A^6}{3^4}$
2 cosine (2x) end cosine + 2 =	$2\cos(2x) + 2 =$
4 - 5 x squared - 6 x cubed = (x squared + 2) squared	$4 - 5x^2 - 6x^3 = (x^2+2)^2$
fraction 10 times 9 times 8 times 7 over 9 times 6 factorial	$\frac{10\times9\times8\times7}{9\times6!}$
k cubed less than k cubed + 2 k squared + 2k + 1 is less than (k+1) cubed	$k^3 < k^3 + 2k^2 + 2k + 1 < (k+1)^3$

Expression as string	**Represents**
2k squared + 2k + 1 = 3k squared + 3k + 1	$2k^2 + 2k + 1 = 3k^2 + 3k + 1$
1 / 320 is less than absolute value f(x) - g(x) end absolute value = 1 / 100 less than or equal to 1/100	$1/320 < \|f(x) - g(x)\| = 1/100 \leq 1/100$
absolute value f(x) - g(x) end absolute value = absolute value x - (x + fraction sine (f xˆ2) end sine over 400 end fraction) end absolute value = absolute value fraction sine (f xˆ2) end sine over 400 end fraction end absolute value less than or equal to 1/400 less than or equal to 1/320	$\|f(x) - g(x)\| = \|x - (x + \frac{\sin(fx^2)}{400})\| = \|\frac{\sin(fx^2)}{400}\| \leq 1/400 \leq 1/320$
g(x) = 1 + integral from 0 to x of f(t) dt end integral = 1 + integral from 0 to x of (1 + t + t squared / 3 + t cubed / 6) dt	$g(x) = 1 + \int\limits_0^x f(t)dt = 1 + \int\limits_0^x (1 + t + t^2/3 + t^3/6)dt$
= 1 + [t + t squared / 2 + t cubed / 6 + tˆ4 / 24] superscript x subscript 0	$= 1 + [t + t^2/2 + t^3/6 + t^4/24]^{x_0}$
h(x) - f(x) = g(x) - f(x) + h(x) - g(x)	$h(x) - f(x) = g(x) - f(x) + h(x) - g(x)$
= g(x) - f(x) + (1 + integral from 0 to x of h(t) dt end integral) - (1 + integral from 0 to x of f(t) dt end integral)	$= g(x) - f(x) + (1 + \int\limits_0^x h(t)dt) - (1 + \int\limits_0^x f(t)dt)$
= g(x) - f(x) + integral from 0 to x of (h(t) - f(t)) dt	$= g(x) - f(x) + \int\limits_0^x (h(t) - f(t))dt$

Expression as string	**Represents**
integral from 0 to x (h(t) - f(t))dt end integral less than or equal x(h(x sub 0) - f(x sub 0) less than or equal to 1/2 (h(x sub 0) - f(x sub 0))	$\int_{0}^{x(h(t)-f(t))dt} \leq x(h(x_0) - f(x_0) \leq 1/2(h(x_0) - f(x_0))$
L(A) = fraction 2 pi - (pi / 2 - 1) - pi / 2 over 2 pi end fraction	$L(A) = \frac{2pi-(pi/2-1)-pi/2}{2pi}$
= fraction greek pi + 1 over 2 greek pi	$= \frac{\pi+1}{2\pi}$
alpha plus bravo	$a + b$
delta multiplied by 2	$d \times 2$
{e times (c + (a , + b) ˆ d] + f	$\{e \times (c + (a, +b)^d] + f$
a + b + c times d times e	$a + b + c \times d \times e$
not fraction a + b over c times d end fraction	$\neg \frac{a+b}{c \times d}$
sum of cosine of greek theta end cosine end sum factorial	$\sum \cos \theta!$
{e times (c + (a + b) ˆ d] + f	$\{e \times (c + (a + b)^d] + f$
hˆg times v) end fraction	$\frac{}{h^g \times v)}$
hˆg v end fraction	$\frac{}{h^g v}$
alpha squared plus bravo squared equals charlie squared	$a^2 + b^2 = c^2$
log xy = log x + log y	$\ln xy = \ln x + \ln y$

Expression as string	**Represents**
fraction df over dt end fraction = limit as h tends to 0 end limit fraction f(t+h)-f(t) over h	$\frac{df}{dt} = \lim_{h\to 0} \frac{f(t+h)-f(t)}{h}$
f = G times fraction m subscript 1 m subscript 2 over r squared end fraction	$f = G \times \frac{m_1 m_2}{r^2}$
india squared = -1	$i^2 = -1$
V - E + F = 2	$V - E + F = 2$
capital greek phi open brackets x close brackets = fraction 1 over square root of 2 greek pi greek rho end square root end fraction e to the power of fraction (e - greek mu) squared over 2 greek rho squared end fraction	$\Phi(x) = \frac{1}{\sqrt{2\pi\rho}} e^{\frac{(e-\mu)^2}{2\rho^2}}$
fraction partial squared u over partial t squared end fraction = charlie squared fraction partial squared u over partial x squared end fraction	$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$
f (greek omega) = integral from minus infinity to infinity of foxtrot open brackets x close brackets echo to the power of begin minus 2 greek pi india x greek omega end dx end integral	$f(\omega) = \int\limits_{-\infty}^{\infty} f(x) e^{-2\pi i x\omega} dx$
greek rho (fraction partial victor over partial t end fraction plus v dot del v) = - nabla p + dell dot capital tango + f	$\rho(\frac{\partial v}{\partial t} + v \cdot \nabla v) = -\nabla p + \nabla \cdot T + f$

Expression as string	**Represents**
nabla dot capital echo equals fraction greek rho over greek epsilon end fraction	$\nabla \cdot E = \frac{\rho}{\epsilon}$
nabla dot H = 0	$\nabla \cdot H = 0$
nabla times E = minus fraction 1 over e end fraction fraction partial H over partial t end fraction	$\nabla \times E = -\frac{1}{e}\frac{\partial H}{\partial t}$
nabla times H = fraction 1 over e end fraction fraction partial E over partial t end fraction	$\nabla \times H = \frac{1}{e}\frac{\partial E}{\partial t}$
dS greater than or equal 0	$dS \geq 0$
india hotel fraction partial over partial t end fraction capital greek psi equals capital hotel capital greek psi	$ih\frac{\partial}{\partial t}\Psi = H\Psi$
H = sum of p(x) log p(x)	$H = \sum p(x) \ln p(x)$
x sub begin t+1 end = kx sub t open brackets 1 - x subscript t)	$x_{t+1} = kx_t(1 - x_t)$
fraction 1 over 2 end fraction greek sigma squared capital sierra squared begin fraction partial squared V over partial S squared end fraction + rS begin fraction partial V over partial S end fraction + begin fraction partial V over partial t end fraction - rV = 0	$\frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} + \frac{\partial V}{\partial t} - rV = 0$

Appendix D

Evaluation Results

The following table shows run times for all three scenarios by repetition (sample) number.

Sample	Mean run time (milliseconds, 2dp)						
	Scenario 1 batch	Scenario 1 incremental	Scenario 2 batch	Scenario 2 incremental	Scenario 3 batch (left)	Scenario 3 batch (right)	Scenario 3 incremental
1	638.97	319.81	412.78	50.38	330.65	334.47	0.12
2	875.30	441.00	577.89	64.11	439.79	438.95	0.12
3	873.46	440.28	576.11	63.87	437.82	436.50	0.12
4	872.75	439.86	573.29	63.50	437.71	436.42	0.12
5	873.55	440.10	573.15	63.43	438.04	436.40	0.12
6	873.07	440.00	573.21	63.44	437.94	436.43	0.12
7	872.77	439.89	573.11	63.45	437.83	436.53	0.11
8	872.85	440.07	573.22	63.47	437.95	436.53	0.12
9	872.74	439.73	573.29	63.44	438.00	436.56	0.11
10	873.37	440.32	573.17	63.47	437.98	436.58	0.11

Sample	Mean run time (milliseconds, 2dp)						
	Scenario 1 batch	Scenario 1 incremental	Scenario 2 batch	Scenario 2 incremental	Scenario 3 batch (left)	Scenario 3 batch (right)	Scenario 3 incremental
11	873.17	439.92	573.27	63.44	438.04	437.05	0.11
12	873.26	440.10	573.30	63.45	437.81	436.53	0.12
13	873.17	440.10	573.03	63.45	437.97	436.67	0.12
14	873.08	439.84	573.40	63.52	437.56	436.32	0.12
15	873.53	440.23	573.12	63.43	437.97	436.63	0.12
16	873.35	440.25	573.75	63.45	438.09	436.97	0.12
17	873.01	440.07	573.17	63.41	437.97	436.64	0.11
18	876.54	441.84	573.19	63.42	437.96	436.54	0.12
19	875.50	441.18	573.23	63.43	437.82	436.49	0.11
20	875.51	441.30	573.23	63.41	438.22	436.76	0.11
21	875.91	441.20	573.32	63.44	437.86	436.53	0.11
22	875.67	441.33	573.28	63.42	437.92	436.90	0.12
23	875.23	441.08	573.19	63.41	437.73	436.38	0.12
24	875.34	441.13	573.40	63.45	437.89	436.61	0.12
25	875.57	441.33	573.23	63.44	438.00	436.58	0.12
26	875.62	441.24	573.57	63.46	438.05	436.57	0.12
27	875.31	440.99	573.38	63.44	437.92	436.66	0.11