Thinking in Blocks: Implications of Using Abstract Syntax Trees as the Underlying Program Model

Daniel Wendel, Paul Medlock-Walton
Scheller Teacher Education Program
Massachusetts Institute of Technology
Cambridge, MA, USA
{djwendel, paulmw}@mit.edu

Abstract—**This paper examines the implications of using Abstract Syntax Trees (ASTs) as the underlying model for program editors and source control. For editors, working at the level of the AST enables error prevention, efficient auto-completion, and seamless use of multiple representations (e.g. blocks-to-text-to-blocks). An AST-based system also lends itself to both real-time and asynchronous collaborative editing, through intention-preserving algorithms much simpler than Operational Transformations. AST-based asynchronous collaborative editing makes several improvements to source control compared to Git, notably: reducing conflicts even in same-position edits, and eliminating diffs (and therefore conflicts) due to changes in formatting, spacing, or method ordering. Even text-based languages can reap these benefits, simply by changing the underlying program representation from text to AST.**

Keywords—AST; blocks-based programming; multiple representations; real-time collaboration; source control

I. Introduction

In this paper we examine the implications of using Abstract Syntax Trees (ASTs) as the underlying model for program editors and source control. We first describe ASTs and their similarity to blocks. We then examine the implications of AST-based thinking on editor design. Finally, we examine the possibilities presented by ASTs in terms of real-time and asynchronous collaboration.

In an Abstract Syntax Tree [1], elements of the program are represented as nodes in a tree, with arguments and nested scopes being children of enclosing commands or structures. These children can either be named/ordered, as in the case of arguments/operands, or unordered, as in the case of methods in a class. In this form, whitespace, braces, tabs, semicolons, and even the particular names of commands/keywords are unnecessary, as the logical program structure is contained within the tree itself.

Interestingly, blocks-based programming languages already use a representation that is quite close to an AST. Consider an "if" block, for example. The block either exists, or does not; there is no "i" block created as an intermediate step, and when the block is created, it brings with it the knowledge that it requires one "test" expression and one "body" statement (or list of statements). If a "test" expression is provided, it links to the "test" section of the block; boolean expressions do not fit into the "command"-type socket where the "body" statement connects. If a user moves the "if" block from one part of the program to another, its "test" and "body" arguments move with it. Indeed, in the blocks-based languages we are familiar with, each block represents one node of an AST, and the arguments (sometimes called sockets, slots, or parameters) of each block represent the named, type-specific branches of the node.

The mapping from text-based languages to ASTs is less trivial, but obviously fully solved, as the concept of ASTs comes directly from text-based languages [1]. Compilation/interpretation in common languages proceeds not directly from a flat array of characters, but rather from an AST generated by a prior stage in the pipeline. In the other direction, modern debuggers are proof that mapping from compiled code (often with embedded metadata) back to text is also a fully-solved problem.

In fact, as discussed in Section 2, ASTs serve as an ideal intermediate format for switching between blocks and text representation. App Inventor's [2] use of S-expressions as an intermediate language when compiling apps is an example of a similar idea, at least in the blocks-to-text direction.

In the following sections we examine several benefits to using ASTs as the internal representation of programs, and of using AST nodes as the "thought unit" when thinking about editing and collaboration.

II. Editing

Creating and editing code using AST nodes prevents users from creating syntax errors, allows atomic reordering of commands in a program, can enable efficient auto-completion, and enables users to view code with multiple representations. Modern IDEs such as Eclipse and IntelliJ already use ASTs to provide a variety of error-prevention and efficiency improvement features, but they do not operate on AST nodes as a "thought unit" as blocks-based languages do.

We use the phrase "thought unit" to mean that users are able add, remove, or modify nodes of the AST, but are not able to create partial versions of these nodes. Traditional text based programming environments do allow users to create partial AST nodes by typing incomplete expressions, which causes common syntax errors such as mismatched curly braces in nested "if" statements. Atomic AST node creation using blocks-based programming, typeblocking [3], or context-sensitive command selection prevents syntax errors by only allowing users to add and remove complete commands and expressions.

AST node editing also enables users to reorder nodes while maintaining a syntactically correct program. Blocks-based programming environments like Scratch [4], Blockly [5], and StarLogo [6] accomplish this by allowing users to place blocks in "stacks" on a workspace. Stacks then remain "connected" if dragged from the top block in the stack, allowing whole portions of code, AST sub-trees, to be moved at once. Reordering commands in text-based languages, on the other hand, requires bulk copy/paste or "move line" operations, which often result in incomplete—or worse, complete but logically incorrect—syntax, as braces or parentheses are accidentally left behind.

Context-sensitive command selection, implemented in both Greenfoot 3 [7] and Microsoft's TouchDevelop [8], provides the user with possible commands that can be connected at the location of the cursor. The cursor is located where AST nodes can be inserted, and users select from a list of possible commands that are available. This allows for efficient, auto-complete-based editing, as often only a few keystrokes are required to disambiguate between the available options.

Editing code using AST nodes also enables users to view multiple representations of the code. Greenfoot's editor can switch to a Java text based representation, while TouchDevelop has three different representation modes, one using blocks, another as easy to read text, and a third as a JavaScript-like language. Pencil Code's Droplet [9] editor, which uses an annotated parse tree not unlike an AST, uses an animated transition between blocks and text, making the integration between the representations truly seamless. And GP [10], a new language being demonstrated at this workshop, uses an AST model and parameterized representations to allow users to actually control via a slider the degree to which the program appears as "blocks" or "text".

III. Collaboration

Direct collaboration—working on the same copy of code—is a complex topic that most of our tools so far have only barely touched upon. None of the code.org tools support direct collaboration. Scratch [4], the most popular blocks-based platform, has a very effective community ecosystem designed around "remixing" and the sharing of ideas, but does not support co-ownership of projects. And StarLogo Nova [6], which does support co-ownership of projects, uses a locking mechanism to prevent more than one person from editing the project at a time. To our knowledge, only Zero Robotics [11] uses true real-time collaboration. In this section, we examine the ways in which an AST-backed model can enable collaboration both in real-time and asynchronously, more easily and more effectively than current systems for text-based collaboration.

A. Real-Time Collaboration

Real-time collaborative text editing has been a topic of research for many years, with Operational Transformations (OT) [12] taking over as the primary algorithm for maintaining consistency and causality in the 2000's. Notable editors using some form of OT include SubEthaEdit [13], Google Docs [14], Word Online [15], and Cloud9 [16]. While OT is considered to have solved the real-time collaboration problem, it is also considered to be complicated and difficult to implement correctly [17]. This barrier has prevented many tools from incorporating real-time collaboration, and even tools like Microsoft Office are only beginning to introduce these features now, some 25 years after OT's invention.

Additionally, real-time direct collaboration has made few inroads in programming, perhaps due to low perceived value. While Pair Programming has been widely studied and adopted, it uses a "driver/navigator" approach of trading (real or virtual) keyboard access, rather than giving both collaborators simultaneous access [18]. Tools like Cloud9 allow multiple simultaneous users to edit a program file, but, tellingly, the video on the marketing page shows one user typing comments while another types code. Indeed, it is hard to imagine two people being able to co-edit a text file in any way except turn-taking, due to the fact that ideas and even syntax are incomplete for a majority of the time spent editing, meaning that no incremental testing is possible until all authors complete their changes.

AST-based editing, especially in the context of a blocks-based editor, has the potential to overcome both the issue of difficulty and the issue of value.

1) Simpler Alternatives to OT Made Possible through ASTs

Since the original OT paper in 1989, the algorithm has been modified in many ways to patch flaws and to add stronger guarantees about system behavior with regards to user expectations [19]. Meanwhile, though, alternate approaches have also been proposed, promising similar consistency and intention-preservation, but with much simpler models and implementations [17, 20, 21, 22].

Three approaches in particular are particularly relevant to AST-based systems: Wantaim [20], WOOT (WithOut OT) [21], and Commutative Replicated Data Types [22]. Each of these approaches is based on the insight that the primary source of complication with OT is that edit operations identify their locations based on an index into the total array of characters, which can change depending on edits by other users. By treating data as a linked structure of uniquely-named nodes, the new approaches eliminate most of the complication involved in OT. Conveniently, ASTs can be implemented exactly that way - as a linked structure of uniquely-named nodes.

To further simplify things, platforms like Meteor [23] and Google Drive Realtime API [24] already exist, which are designed to abstract away the process of model synchronization. In order to use such a system, all that is required of the editor is that it must update the UI in response to changes to the model. While such a solution might lack some of the intention-preserving features of an application-native, OT-like system for co-located edits, it would nevertheless enable useable collaboration with very little work on the part of the blocks editor creator.

2) Value of Real-time Collaboration in Blocks-based Environments

While real-time collaboration is clearly simpler to implement in an AST-backed editor than in plain text, what is less obvious is that the value of the collaboration itself may be higher in a block-based environment than in a text one. This stems from two important features of blocks-based editors: syntax atomicity and built-in scratch space.

Syntax atomicity means that a blocks-based program is not syntactically "broken" in intermediate states. That is, even as a user is editing, the compiler always knows the meaning of each syntactic node in existence. Some languages, such as Scratch [4], go one step further and even fill in default values for empty arguments, meaning that the program remains in a valid, executable state at all times. This is clearly not a feature of traditional text environments, and is unique to blocks-based or AST-based systems.

Built-in scratch space is also a uniquely blocks-based idea. In many blocks-based editors, the exact location of the blocks is insignificant, and spare or stray blocks or stacks of blocks can be scattered around the page without affecting the program execution. Only when a block is attached to a "root" block (such as an event block) does it become a part of the program and begin to execute. For this reason, changes or additions to a particular section of code can easily happen "all at once," as new, complete code is moved from scratch space into the main stack of blocks. Indeed, in studying the programming habits of Scratch users, [25] found that most tended to build their programs in this (uniquely blocks-based) "bottom-up" fashion.

Taken together, these features mean that blocks-based code remains in an executable state during most edits. While changes to system-wide designs will of course prevent system-wide execution while they are being completed, small portions of code (for example, individual procedures) can still be run and tested independently, even with multiple authors editing multiple locations in the program. This makes the case for real-time collaboration more compelling in a blocks-based environment than in a traditional text-based one.

B. *Asynchronous Collaboration*

Asynchronous collaboration (e.g. source control) also stands to gain substantially from moving to an AST-based model. While Git [26] and similar systems are adept at enabling large groups of people to contribute to the same codebase, merge conflicts still exist and require structures outside the system (for example, social norms or additional check-in procedures) to ameliorate them. While multiple authors changing the same code in incompatible ways must still require intervention and resolution, two common sources of merge conflicts, from changes that are not actually conflicting, can be avoided by using ASTs: formatting conflicts and cut/paste conflicts.

Formatting conflicts occur for many reasons, but one of the most common in StarLogo Nova's underlying codebase (with which the authors are intimately familiar) is due to new developers making a change and then undoing the change (via CTRL+Z) in different editors. When the line is edited, some editors swap tabs for spaces (or vice-versa), creating a change that is invisible to the developer but causes conflicts with other, real changes to the same line. While any large team will use code conventions and software checks to enforce formatting compliance, such tools solve a problem that need not exist at all; formatting conflicts like these are by definition eliminated in an AST-based system, since spacing (and all other textual representations of syntax structure) is not even stored in the system. Indeed, code conventions could be rendered obsolete immediately, as individual preferences for spacing, line wraps, and even method ordering could be stored as preferences in a developer's profile, rather than in the code repository.

Cut/paste conflicts, or positional change conflicts, are another common source of frustration that could be automatically resolved in an AST-based system. For example, imagine an "if" block with a condition that checks a property of an object. One author may realize that another property also needs to be checked due to a corner case, and make that change. Another author may realize that the entire block needs to be moved one spot down, to ensure that a pre-condition is met, and so cuts and pastes the block. In text-based editors, these operations would result in conflicts at the line of the condition. But in an AST-based system, the edits are performed on different nodes (the "if" node and its condition node), and are not in conflict at all.

Clearly, an AST-based system can reduce reported conflicts from some of the common cases where no actual conflict exists. But what about the case where a true conflict arises due to logically incompatible edits? While ASTs alone do not solve this problem, combining them with an intention-preserving, real-time system would provide significant benefits through its operation log. Whereas diff-based systems merely report the end result of a change, operation-log-based systems allow rewind and playback of a particular author's changes, and some [27] even allow selective undo of past actions. This provides much greater insight and conflict resolution power than current text-diff tools allow.

IV. Conclusions

In this paper we have explored a few of the benefits of using ASTs as the underlying model for programs. These include error-reduced editing, real-time collaboration, and conflict-reduced asynchronous collaboration. Additionally, we have observed that blocks-based editors already use a model that is quite close to ASTs. This means that powerful new features should be able to be added to our blocks-based systems with perhaps less effort than we may have anticipated.

Additionally, blocks-based systems are not the only ones that can benefit from an AST model. Text-based languages too can reap the benefits of improved editing and source control management; in particular, the elimination of whitespace and formatting from source control seems to be a strict improvement.

Using ASTs as the underlying model for programming, regardless of whether the language is natively blocks- or text-based, will enable improvements to our tools and to our reasoning about our tools. While this paper is merely exploratory, highlighting potential areas of impact, our hope is that our community will further investigate and develop these ideas, and ultimately bring about a better experience for our users that follows in the blocks-based philosophy of "getting to the hard fun part" of programming more quickly.

ACKNOWLEDGMENTS

The idea of representing programs as nodes in an AST rather than as blocks or as a string of characters, and the implications of this subtle difference in thinking, developed over several months through conversations with many people. In particular, we thank Michael Kölling of Greenfoot for the inspiration of frame-based editing, Evelyn Eastmond of Scratch for having a wedding attended by John Maloney of Scratch/GP and Matt Green, who first called the AST an "AST" rather than a "block tree", and many colleagues at MIT STEP (Eric Klopfer, Joshua Sheldon, Wendy Huang, Eli Kosminsky, and others) who helped to develop the ideas over many lunch breaks.

REFERENCES

[1] Joel Jones (2003). "Abstract Syntax Tree Implementation Idioms" http://www.hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf

[2] MIT App Inventor. http://appinventor.mit.edu/explore/ retrieved Sept. 2015

[3] McCaffrey, Corey (2006). "StarLogo TNG: The Convergence of Graphical Programming and Text". MIT Master's Thesis, 2006. SSRN:http://ssrn.com/abstract=1639257

[4] Scratch. https://scratch.mit.edu/ retrieved Sept. 2015

[5] Blockly. https://blockly-games.appspot.com/about?lang=en retrieved Sept. 2015

[6] StarLogo Nova. http://www.slnova.org/ retrieved Sept. 2015

[7] Greenfoot. http://www.greenfoot.org/ retrieved Sept. 2015

[8] TouchDevelop. https://www.touchdevelop.com/ retrieved Sept. 2015

[9] Bau, D. A (2014). "Droplet, a Blocks-based Editor for Text Code". Whitepaper. http://ideas.pencilcode.net/home/htmlcss/droplet-paper.pdf retrieved Sept. 2015

[10] John Maloney. "GP". Personal demonstration at MIT STEP lab, August 28, 2015. Unpublished.

[11] Zero Robotics. http://zerorobotics.mit.edu/ retrieved Sept. 2015

[12] Ellis, C.A.; Gibbs, S.J. (1989). "Concurrency control in groupware systems". ACM SIGMOD Record (2): 399–407.doi:10.1145/66926

[13] SubEthaEdit. http://www.codingmonkeys.de/subethaedit/ retrieved Sept. 2015

[14] Google Docs. https://www.google.com/intl/en/docs/about/ retrieved Sept. 2015

[15] Microsoft Office Online. https://office.live.com/start/default.aspx retrieved Sept. 2015

[16] Cloud9. https://c9.io/ retrieved Sept. 2015

[17] Neil Fraser (2009). "Differential Synchronization". Whitepaper. http://neil.fraser.name/writing/sync/ retrieved Sept. 2015

[18] Williams, L. and Kessler, R. (2003). Pair Programming Illuminated. Boston: Addison-Wesley Professional.

[19] Chengzheng Sun; Xiaohua Jia; Yanchun Zhang; Yun Yang; David Chen (1998). "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems". ACM Trans. Comput.-Hum. Interact. (1): 63–108.doi:10.1145/274444.274447

[20] Edya Ladan-Mozes, Qian Liu, Jonathan Rhodes, Daniel Wendel (2006). "Wantaim - A Distributed Real-time Collaborative Text Editor". MIT 6.824. Robert Morris, professor. https://www.academia.edu/14441025/ retrieved Sept. 2015

[21] Gérald Oster, Pascal Urso, Pascal Molli, Abdessamad Imine. Real time group editors without Operational transformation. [Research Report] RR-5580, 2005, pp.24.

[22] Marc Shapiro, Nuno Preguiça. Designing a commutative replicated data type. [Research Report] RR-6320, 2007.

[23] Meteor. https://www.meteor.com/features retrieved Sept. 2015

[24] Google Realtime API: https://developers.google.com/google-apps/realtime/overview retrieved Sept. 2015

[25] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2011. Habits of programming in scratch. In Proceedings of ITiCSE '11. ACM, New York, NY, USA, 168-172. DOI=10.1145/1999747.1999796 http://doi.acm.org/10.1145/1999747.1999796

[26] Git. https://git-scm.com/ retrieved Sept. 2015

[27] Chengzheng Sun (2002). "Undo as concurrent inverse in group editors". ACM Trans. Comput.-Hum. Interact. (4): 309–361.doi:10.1145/586081.586085