VLSS

Journal of Visual Languages and Sentient Systems

Volume 3, July, 2017

Journal of
Visual Languages and Sentient Systems

An Open Access Journal published by
KSI Research Inc.
156 Park Square Lane, Pittsburgh, PA 15238 USA

VLSS Editorial Board

DOI: 10.18293/VLSS2017

Proceedings preparation, editing and printing are sponsored by KSI Research Inc.

Journal of
Visual Languages and Sentient Systems

Volume 3, 2017

Table of Contents

INFORMATION FOR AUTHORS

The Journal of Visual Languages and Sentient Systems (VLSS) is intended to be a forum for researchers, practitioners and developers to exchange ideas and research results, for the advancement of visual languages and sentient multimedia systems. Sentient systems are distributed systems capable of actively interacting with the environment by gathering, processing, interpreting, storing and retrieving multimedia information originated from sensors, robots, actuators, websites and other information sources. In order for sentient systems to function efficiently and effectively, visual languages may play an important role.

VLSS publishes research papers, state-of-the-art surveys, review articles, in the areas of visual languages, sentient multimedia systems, distributed multimedia systems, sensor networks, multimedia interfaces, visual communication, multi-media communications, cognitive aspects of sensor-based systems, and parallel/distributed/neural computing & representations for multimedia information processing. Papers are also welcome to report on actual use, experience, transferred technologies in sentient multimedia systems and applications. Timely research notes, viewpoint articles, book reviews and tool reviews, not to exceed three pages, can also be submitted to VLSS.

Manuscripts shall be submitted electronically to VLSS. Original papers only will be considered. Manuscripts should follow the double-column format and be submitted in the form of a pdf file. Page 1 should contain the article title, author(s), and affiliation(s); the name and complete mailing address of the person to whom correspondence should be sent, and a short abstract (100-150 words). Any footnotes to the title (indicated by *, +, etc.) should be placed at the bottom of page 1.

Manuscripts are accepted for review with the understanding that the same work has not been and will not be nor is presently submitted elsewhere, and that its submission for publication has been approved by all of the authors and by the institution where the work was carried out; further, that any person cited as a course of personal communications has approved such citation. Written authorization may be required at the Editor's discretion. Articles and any other material published in VLSS represent the opinions of the author(s) and should not be construed to reflect the opinions of the Editor(s) and the Publisher.

Paper submission should be through https://www.easychair.org/conferences/?conf=dmsvlss2017

For further information contact: vlss@ksiresearch.org

Guest Editor Introduction to the
VLSS Special Issue on Blocks Programming

Franklyn Turbak
Computer Science Department
Wellesley College, Wellesley MA, USA
fturbak@wellesley.edu

This special issue of the *Journal of Visual Languages and Sentient Systems (VLSS)* focuses on *Blocks Programming*. Over the past twenty years, blocks programming languages have evolved from research lab prototypes to practical tools used by tens of millions of people. The core idea behind these languages is simple: rather than constructing programs out of sequence of characters that are lexed into tokens that are parsed into syntax trees, why not construct syntax trees more directly by composing drag-and-drop visual fragments that represent the nodes of the trees?

There are now a host of blocks languages that can be used for a wide variety of purposes, such as creating games, scripting 3D animations, inventing mobile apps, developing agent-based simulations, controlling robots, generating 3D models. and collecting/analyzing/visualizing scientific data.

Educators have seized upon blocks languages as a way to introduce beginners to computer science concepts. Environments like Scratch, Blockly, Pencil Code, App Inventor, Snap!, AgentSheets/AgentCubes, and Alice/Looking Glass have become popular ways to introduce students (from grade schoolers through graduate students) to computational thinking and programming. This is done not only in traditional classrooms, but also in extracurricular settings via online activities like Code.org's *Hour of Code*.

But blocks environments are also a boon to a broader population whose primary goal is to "make stuff" rather than learn computer science principles. This includes artists, scientists, hobbyists, entrepreneurs, and other end users and so-called *casual programmers* who want to build computational artifacts for themselves, their families, their workplaces, their communities, and the world at large.

There are many questions surrounding blocks languages. Why are they so popular? What are their key advantages and limitations compared to other programming environments? Do they really lower barriers to programming, or is that just hype? In what ways do they help and hinder those who use them as a stepping stone to traditional text-based languages? What are effective pedagogical strategies to use with blocks languages, both in traditional classroom settings and in informal and open-ended learning environments? How does the two-dimensional nature of blocks programming workspaces affect the way people create, modify, navigate, and search through their code? Do blocks environments have any features that are worth incorporating into IDEs for traditional programming environments? What are effective mechanisms for multiple people to collaborate on a single blocks program when they are co-located or are working together remotely? Can blocks-based environments be made accessible to users with visual or motor impairments?

These questions are being investigated by educators, software developers, and researchers from areas that include visual languages, human-computer interaction, programming languages, education, psychology, learning science, and data analytics. This special issue gives a sample of the kinds of research and development that is taking place in the blocks programming community.

Many blocks languages are *domain-specific languages (DSLs)* that are not general programming systems but are instead targeted at tasks in a particular domain. An excellent example of such a DSL is *SparqlBlocks*, a blocks-based environment for making queries to find information encoded in linked data within the semantic web. This system is described by its creators, **Miguel Ceriani and Paolo Bottoni**, in the first article of this issue, *SparqlBlocks: Using Blocks to Design Structured Linked Data Queries*. The *SparqlBlocks* programming environment highlights several key advantages that blocks languages have with respect to traditional textual notations (in this case, the text-based SPARQL query language). First, *SparqlBlocks* language features are represented by drag-and-drop blocks whose shapes both suggest how they can be composed and only allow syntactically valid connections to be made. Second, generating and editing queries involves manipulating high-level chunks of information (blocks) as opposed to low-level character representations, which not only aids the cognitive task of assembling queries but also avoids frustrating errors due to misspelled keywords or incorrect punctuation. Third, the blocks of the language are organized into categories within a toolbox menu, allowing users to learn available functionality by exploring the menu options and use recognition rather than recall to find the constructs that that they need. Fourth, the system has a *liveness* property such that query results are automatically displayed as soon as a valid query is created and updated as soon as the query is modified. (Although liveness has nothing to do with blocks per se, it is a common feature of many blocks environments.)

In addition to these standard benefits, *SparqlBlocks* has an innovative twist: the query result tables themselves are represented as blocks, which encourages the use of result fragments in new queries. Anyone who has programmed using JSON data representations or Lisp-like s-expressions will appreciate the power unleashed by having output data representations that can easily be repurposed as input data within a program.

SparqlBlocks is clearly not a language targeted at children, but at people who understand the Resource Description Framework (RDF) framework of the semantic web. So it serves as an interesting case study for those who are dismissive about blocks languages, and claim they can't be used for "real" coding or are just a stepping stone for programming in traditional text-based languages. In this case, it's easy to imagine a class of users who would be completely satisfied with using *SparqlBlocks* for all their semantic web queries without ever feeling the need to learn the textual SPARQL syntax. Perhaps the naysayers are confused by the brightly colored blocks and the fact that many current blocks languages have limited environments that are targeted at those who have never programmed before, and so they incorrectly conclude that such environments are only good for kids and other newbies and could never be of use to adults or seasoned programmers. But fundamentally, blocks languages are just an alternative way of representing the syntax trees that underly any program, so there's no reason that they can't express everything that a traditional text-based language can express. And it's worth remembering that it was not so long ago that "real programming" was done only in assembly code and higher-level programming languages were such a futuristic idea that their study was considered a branch of artificial intelligence. It will be interesting to see what is considered "real programming" twenty to forty years hence.

This is not to say that blocks languages don't have downsides. Common complaints are that blocks representations are are less dense than text representations (so less information can be conveyed in one screenful) and that they have high *viscosity* — creating, moving, connecting, disconnecting, copying, and deleting blocks when editing a program can take more operations and time than entering and editing text. And syntactically correct programs aren't necessarily semantically meaningful. For example, in *SparqlBlocks*, many users struggle with the correct usage of literals and variables in their queries.

An important line of research in blocks based languages is mitigating these downsides. The next three articles in this special issue address this theme.

One idea for combining the benefits of blocks and text within a single environment is *bidirectional mode switching*, in which there is a way to convert a blocks program workspace into an editor buffer with an isomorphic text program and vice versa. There are now several blocks programming environments that support bidirectional mode switching, but one of the earliest was Tiled Grace, which is described by its developers **Michael Homer and James Noble** in their article *Lessons in Combining Block-based and Textual Programming*. Tiled Grace is a blocks version of the existing Grace text-based language. Users can edit a Tiled Grace program either in blocks mode or text mode and, with the press of a button, see their program transformed by a smooth animation into the other mode. Each mode has its own advantages. Blocks mode allows editing the program syntax tree with meaningful chunks, provides a toolbox that organizes the language constructs into inspectable categories, and provides visual feedback for syntax errors and program dependencies (e.g., associating references of variable and method names with their declarations). Text mode allows low-viscosity editing of a denser representation of the same program.

The article describes an experiment in which early undergraduate Java programmers were asked to write, modify, correct, and describe Tiled Grace programs, and were allowed to switch between blocks mode and text mode as many times as they wanted to in order to complete their tasks. One interesting result is that participants switched modes numerous times in performing the tasks. Many participants (who were already used to text-based programming in Java) found drag-and-drop editing with blocks frustrating, so would switch to text mode to edit the programs. But some participants would switch to blocks mode to perform certain kinds of edits. Intriguingly, over a third of switches to blocks mode involved no edits, implying that participants were using it just to examine the structure of the code.

The authors compare their study to some other studies involving bidirectional mode switching. These other studies involved different environments, different tasks, and participants with different programming backgrounds, so it's not surprising that there is a lot of variability between results. But clearly, bidirectional mode switching is a rich area for future study. and can give insight into the relative advantages of blocks vs. text for creating, modifying, debugging, and understanding programs.

A different approach to mediating between blocks and text is described in the *Frame-Based Editing* article by **Michael Kölling, Neil C. C. Brown, and Amjad Altadmri**. They describe a novel editing paradigm based on *frames* that is the basis for an editor for the Java-like Stride language that has been incorporated into their Greenfoot and BlueJ programming environments. Rather than switching between blocks and text views, the frame-based editor combines key aspects of both blocks and text in a single unified view.

Declarations and statements are represented as frames, which are visual block-like entities that can be created, moved, copied, and deleted as single syntactic units. Frames have slots for nested program structure. There are two types of slots. A frame slot is filled with another frame, while a text slot is filled by typing characters on the keyboard (or copying and pasting text from elsewhere). Text slots are used for entering expressions (e.g., numbers, variables references, arithmetic/relational/logical operations, etc.) and specifying things like parameter names, types, and the assigned variable in an assignment statement. Although frames and frame/slot cursors

can be selected/dragged/positioned with a mouse, there are keystrokes for creating, selecting, copying, pasting, and deleting frames and for navigating to frames and slots. This makes it possible to edit Stride programs using only keystrokes, even though frames are block-like features.

The Stride frame-based editor supports numerous other features that give it a low-viscosity "feel" more like a professional text-based Integrated Development Environment (IDE) than a high-viscosity visual language. The article describes a few usability experiments with the Stride editor, some of which are positive but others of which indicate areas for improvement. More detailed experiments with frame-based editing and comparing it with editing in other blocks based environments are obvious areas for future work, as are incorporating frame-based features into other blocks-based environments and adapting frame-based editors to other text-based languages.

This article also describes how frame-based and blocks-based editing arose out of two research threads: one involving structure editors for text-based programs, and another involving visual programming languages. Members of the blocks community should familiarize themselves with both of these lines of research.

Alex Repenning's article *Moving Beyond Syntax: Lessons from 20 Years of Blocks Programming in AgentSheets* is his personal reflection about the history and future of blocks programming. Repenning's 1995 AgentSheets environment for rule-based programming of agents in a two-dimensional grid included a blocks programming editor for specifying the rules that control the agents. Repenning defines four affordances for blocks programming that he uses as a lens to discuss the history of blocks programming and related work in visual programming. Readers interested in the historical underpinnings of blocks programming should follow up on the work cited by Repenning as well as the structure editor work summarized by **Kölling, Brown, and Altadmri**.

The main thrust of Repenning's article is that although blocks languages have clear benefits for making the syntax of programming less frustrating for beginners, developers of these environments should focus more on improving their semantics and pragmatics (which he defines as *the study of what code means in particular situations*). He gives some compelling examples from AgentSheets and the follow-on AgentCubes languages: automatically generated context-based documentation and error messages, and so-called *conversational programming* and *live palette* techniques that support context-based debugging and prediction of program behavior — e.g., dynamically showing the values of rule conditions and which rules would fire as agents are dragged through different cells in a grid world. Such feedback can give programmers a better understanding of the *behavior* of their programs, not just their *structure*.

Repenning also argues that environments for beginners need to focus less on traditional programming skills and more on computational thinking and problem-solving skills. In this context, there are advantages to having restricted computational thinking environments that are tuned to having relatively simple solutions to certain classes of problems rather than general programming tools that can solve any problem. Repenning notes that solutions with general programming tools are often swamped by *accidental complexity*, whereas in more constrained environments, it is easier to support pragmatics. Many of the semantic, pragmatic, and computational thinking benefits mentioned by Repenning are strongly rooted in the particular grid-with-agents-controlled-by-rules paradigm embodied in his AgentSheets and AgentCubes environments. But how can these sorts of benefits be realized in other paradigms, domains, and modalities? This is a rich area for future research. As a concrete challenge to the reader, I suggest considering how to apply some of Repenning's ideas to **Ceriani and Bottoni's** *SparqlBlocks* system, or some other blocks language of their choice.

Although Repenning defines four key affordances of blocks programming, he doesn't explore in detail *why* those affordances should help blocks programmers. In their article *How Block-based Languages Support Novices*, **David Weintrop and Uri Wilensky** propose a framework for understanding how such affordances help learners construct meaning from the environment in the context of solving a particular problem. They use a notion of *webbing* to describe all resources available to the learner in a situated computational problem solving task. Their framework involves two dimensions: the first dimension distinguishes internal cognitive processes vs. external communication; the second dimension distinguishes generating a program vs. interpreting one. Based on the 2 x 2 matrix involving these dimensions, they describe four roles for the learner. They then give examples from a study involving RoboBuilder (a blocks-based environment for describing the behavior of screen-based robots that battle each other) of how affordances of blocks help learners in each of the four roles. An interesting aspect of their analysis is it suggests that designers should consider all four roles when designing a blocks language. However, there may be trade-offs in design that benefit one role at the expense of another. Elaborating on design decisions and trade-offs implied by this framework, as well as applying it to non-blocks programming languages, are avenues for future research.

Blocks programming environments are used in many situations where novice programmers make heavy use of example programs to learn features of the language, often with little or no help from an instructor or peer. In order to design helpful examples and incorporate them into the learning process, it is essential to better understand how blocks programmers learn via examples. This is the motivation behind the work by **Michelle Ichinco, Kyle Harms, and Caitlin Kelleher** in their article *Towards Understanding Successful Novice Example Use in Blocks-based Programming*. They investigated two features of examples in Looking Glass: (1) different styles of annotations that explain how an example works and (2) surface and structural similarities between and example and the assigned task (which involved modifying a given program).

In a study of 99 children aged 10 to 15, they found that annotations on examples (vs. no annotations) helped participants complete a similar task, but not a dissimilar one. Also, subjects completed more tasks using similar examples than dissimilar ones, and the similarity was correlated with getting to a later stage in a 4-stage model of task completion. However, the tasks were challenging for the subjects, and many subjects did not get past the 1st of the 4 stages for many of the tasks. More work clearly needs to be done to help novices get unwedged from this 1st stage.

To understand the level of understanding that subjects had of the relationship between the examples and tasks, subjects were asked to draw analogical mappings that connected parts of the example program to corresponding parts of the (unmodified) task program. The study found strong correlations between annotations and correct mappings and between example similarity and correct mappings, but only a weak correlation between correct mappings and task success.

Finally the study looked at how programming behavior was correlated with task completion. For example, editing the program in the 1st stage or performing many user interface operations in the 2nd stage were strong indicators that subjects would *not* complete the task.

This study indicates that example-based learning in blocks programming may not be particularly effective. It remains to be seen whether more carefully crafted examples with more explicit analogical mappings, possibly enhanced with other forms of scaffolding, can increase the effectiveness of examples.

The special issue concludes with the topic of accessibility of blocks programming environments. Many people naturally assume that the visual, drag-and-drop nature of blocks languages inherently makes them inaccessible to those with visual or motor impairments. But this is not necessarily the case! The fact that blocks programming reifies syntax tree nodes and focuses on creating, modifying, and navigating syntax trees may be able to support programming for those with disabilities better than character-based languages.

In their research notes paper *Design Considerations to Increase Block-based Language Accessibility For Blind Programmers Via Blockly*, **Stephanie Ludi and Mary Spencer** describe their work-in-progress on extending the Blockly browser-based blocks language framework to support visually impaired users. One aspect of their work is extending Blockly with screen reader support so that it can pronounce block assemblies (including unfilled sockets) in a meaningful way. Another is providing keyboard support for navigating, connecting, disconnecting, etc. blocks and using menus on the workspace. (As discussed in Sec 14.5 of **Kölling, Brown, and Altadmri's** article, the fact that their frame-based Stride editor already has complete keyboard support for all editing operations makes it an interesting candidate for accessibility.) They also reference work on vocal interfaces that can make blocks programming accessible to people with motor impairments.

In the spirit of promoting accessibility, all researcher who develop and study blocks environments are encouraged to become familiar with ways of making these environments more accessible.

Many individuals contributed to this special issue. I am grateful Professor S-K. Chang, the Editor-in-Chief of *VLSS*, for his support and encouragement for this special issue. Many thanks to the authors who responded to the call for papers for this issue. Of the nine papers that were submitted, six were accepted as full papers (subject to revisions) and one was accepted as a research notes paper. I especially appreciate the willingness of the authors of the accepted papers to go through three rounds of revisions on their papers (two based on reviewer feedback and a final one based on my feedback), which led to very high-quality results. Finally, I am indebted to our reviewers, whose constructive comments during two rounds of reviews greatly improved the quality of the papers.

Preparing this special issue has been a rewarding experience. It has underscored how much exciting work is going on in the area of blocks programming and exposed fertile ground for future research, both in terms of studying how people use and learn from blocks environments and improving the environments to address numerous problems that have been identified in existing systems.

There is a vibrant community of researchers conducting work on blocks programming. Join us!

SparqlBlocks: Using Blocks to Design Structured Linked Data Queries

Miguel Ceriani[1,2] and Paolo Bottoni[1]
[1]Sapienza, University of Rome, Italy
[2]Instituto Tecnológico de Buenos Aires, Argentina
{ceriani,bottoni}@di.uniroma1.it

Abstract *While many Linked Data sources are available, the task of building structured queries on them is still a challenging one for users who are not conversant in the specialised query languages required for their effective use. A key hindering factor is the lack of intuitive user interfaces for these languages. The block programming paradigm is becoming popular for the development of visual interfaces that are easy to use and guaranteed to generate syntactically correct programs, promoting a gradual and modular approach to the task of programming. We exploit these features of the block paradigm to develop SparqlBlocks, a visual language and an integrated user interface in which both Linked Data queries and results are represented as blocks, supporting a modular and exploratory approach to query design. By integrating the presentation of queries and results, reuse of results in the refinement of queries is promoted, as well as the exploration of both the data and the structure of Linked Data sources. SparqlBlocks has been evaluated with 11 users literate in computer science but with small to no expertise in querying Linked Data. After a tutorial, all the users were able to build at least a simple query and all but two were able to build nontrivial queries.*

1. Introduction

Linked Data [1] — the structured data available online — are increasing both in quantity and diversity [2]. A key advantage of the Linked Data model is its support for serendipitous exploration and reuse of existing data. In practice, though, exploring and querying Linked Data is not trivial and requires knowledge of RDF [3] (the basic data model), SPARQL [4] (the standard query language), and a number of schemas and ontologies (the domain/dataset specific data models).

Existing experimental tools for non-experts (see for example [5, 6]), while being effective for some cases, do not support the user much in an incremental process of query design, as reusing intermediate queries and results for new queries is not easy.

As block programming languages [7, 8] — in which coding occurs by dragging and connecting fragments shaped like jigsaw puzzle pieces — have been successfully adopted to introduce programming to non-experts, we decided to leverage the block programming paradigm to design Linked Data queries, supporting an exploratory approach to query design in which language affordances are visually exposed and syntax errors are avoided. Hence, users are not required to know in advance the details of RDF and SPARQL, while the flexibility and expressiveness of a complex query language are preserved.

A specific challenge in querying Linked Data is supporting their heterogeneous nature: no expert can achieve perfect knowledge of the structure and semantics of all the data they may need to use from multiple sources; systems should thus help in discovering structure and semantics of data sources. To deal with this, we build on the block programming approach, presenting a novel paradigm for interactive queries in which both the queries and their results are integrated and interoperable in the workspace. Results are available as blocks that can be used as part of existing queries (to refine them) or to build new (follow-up) queries. Queries are dynamically executed as they are built or modified. As a result, the supporting environment fosters an exploratory approach such that users may start querying datasets without knowing their specific organization and gather progressively more detailed information.

The described approach has been implemented in SparqlBlocks[1] [9, 10], a visual language and an associated visual environment for designing and executing queries on Linked Data sources. The target user of the system is anyone interested in building queries beyond simple data browsing, on one or multiple datasets. The tool may also work as an educational aid for learning Linked Data technologies.

In this paper, we describe in detail the language and user interface of SparqlBlocks and report on a controlled user evaluation aimed at proving that the SparqlBlocks environment may be used to build nontrivial queries on Linked Data without prior knowledge of RDF, SPARQL, or even of the data source's content and structure.

Paper organisation. After introducing the technological background in Section 2, we present and discuss related work in Section 3. In Section 4 we describe the SparqlBlocks visual language and environment, reviewing their specific requirements and features, while in Section 5 we describe its implementation. An analysis from the perspectives of cognitive dimensions and query affordances is given in Section 6, followed in Section 7 by the description of preliminary informal

DOI reference number: 10.18293/VLSS2017-011

[1]http://sparqlblocks.org/

feedback from users and how, based on that feedback, the design evolved. The tool has been evaluated in a user study, described in Section 8, that involved 11 users without prior knowledge of either RDF, SPARQL, or the structure and content of the used dataset. The experimental results are shown both from a quantitative and a qualitative point of view in Sections 9 and 10, respectively. As pointed out in the conclusive discussion in Section 11, the results are encouraging, as users were satisfied by the user interface and able to build nontrivial queries, but they also reveal a number of possible improvements for the tool.

2. Background

We introduce the basics of the RDF data model, the SPARQL query language, and DBpedia, a dataset that will be used for examples and evaluation. We then briefly describe the block programming paradigm.

2.1. Data Model: RDF

In the Resource Description Framework (RDF) [3], the data model proposed by the W3C for Linked Data, knowledge is represented via *RDF statements* about *resources*, which are meant to represent anything in the "universe of discourse", e.g. documents, people, physical objects, abstract concepts. An RDF statement is represented by an *RDF triple*, composed of *subject* (a resource), *predicate* (specified by a resource as well, called a *property*), and *object* (a resource or a *literal*, i.e. a value from a basic type). An *RDF graph* is a set of RDF triples.

A literal is a simple value that can be either a language-tagged string — a string associated with a language tag that identifies the (natural) language of the label — or a typed literal — a value expressed with a string and an associated type that may be any IRI, but which is usually one of the basic datatypes defined by W3C for the XML Schema Definition Language (XSD) 1.1 [11].

Resources are identified by an *Internationalized Resource Identifier* (IRI) [12], a generalization of the *Uniform Resource Identifier* (URI) [13] for retrieving content in an HTTP context. A resource may have one or more *types*, which are also resources. If a resource is used as type for other ones, it is called a *class*. A human-readable version of a resource's name is a string literal, called its *label*. A resource may have one or more labels. Thanks to language tags, a resource can have labels in different languages.

An *RDF dataset* is a set of *named RDF graphs*, i.e., RDF graphs associated with an IRI, the *graph name*, along with a single *default graph*, an RDF graph without a name. Named RDF graphs are used to represent data associated with specific contexts. Usually the default graph of an RDF dataset is either the union of all the named graphs or holds meta-data about the named graphs.

RDF graphs and RDF datasets can be serialized through different concrete RDF syntaxes (Turtle, JSON-LD, XML/RDF). A common feature of multiple RDF syntaxes (used also in SPARQL) is that *prefixes* can be used in place of the initial part of an IRI, which represents specific namespaces for vocabularies or sets of resources. For example, the two IRI namespaces for standard RDF concepts[2] are usually referred to via the prefixes `rdf:` and `rdfs:`, as in `rdf:type`, which is the property used to associate a resource with its type(s) and in `rdfs:label`, which is the property used to associate a resource with its label(s).

Figure 1 shows a graphical depiction of an RDF graph where resources are represented as ovals, literals as rectangles, and triples as labelled arrows connecting them (from subject to object, while the label represents the predicate). The labels for resources and predicates (that are resources too) are IRIs in prefix notation, while the labels for literals are the lexical representation of the literals.

In Listing 1, the same RDF graph is represented using Turtle, an RDF syntax offering prefix notation. Turtle allows authors to avoid repeating the subject of a sequence of triples when it is the same. The semicolon (;) separates predicate/object pairs that apply to the same subject. The dot (.) separates blocks of triples having a common subject.

```
@base             <http://www.w3.org/> .
@prefix rdf:      <1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:     <2000/01/rdf-schema#> .
@prefix dbpedia:  <http://dbpedia.org/resource/> .
@prefix dbo:      <http://dbpedia.org/ontology/> .

dbpedia:Mount_Everest
    rdf:type dbo:Mountain ;
    rdfs:label "Mount␣Everest"@en ;
    dbo:elevation 8848 ;
    dbo:locatedInArea dbpedia:Nepal ;
    dbo:locatedInArea dbpedia:China .

dbpedia:K2
    rdf:type dbo:Mountain ;
    rdfs:label "K2"@en ;
    dbo:elevation 8611 ;
    dbo:locatedInArea dbpedia:China .

dbpedia:Nepal
    rdf:type dbo:Country ;
    rdfs:label "Nepal"@en .

dbpedia:China
    rdf:type dbo:Country ;
    rdfs:label "China"@en .
```

Listing 1. Turtle code for the RDF graph in Figure 1.

2.2. Query Language: SPARQL

SPARQL [4] is the standard query language for RDF datasets[3], based on the notion of *triple pattern*, an RDF triple in which each component can be replaced by a variable. A *basic graph pattern* is a set of triple patterns associated with

[2] http://www.w3.org/1999/02/22-rdf-syntax-ns# and http://www.w3.org/2000/01/rdf-schema#

[3] For conciseness, rather than describing SPARQL syntax in detail, we just show the basics of SPARQL semantics and then a few elements of the syntax. This should be sufficient to both (1) have an idea of the language and (2) understand the SPARQL queries that are shown in the paper in comparison with SparqlBlocks syntax.

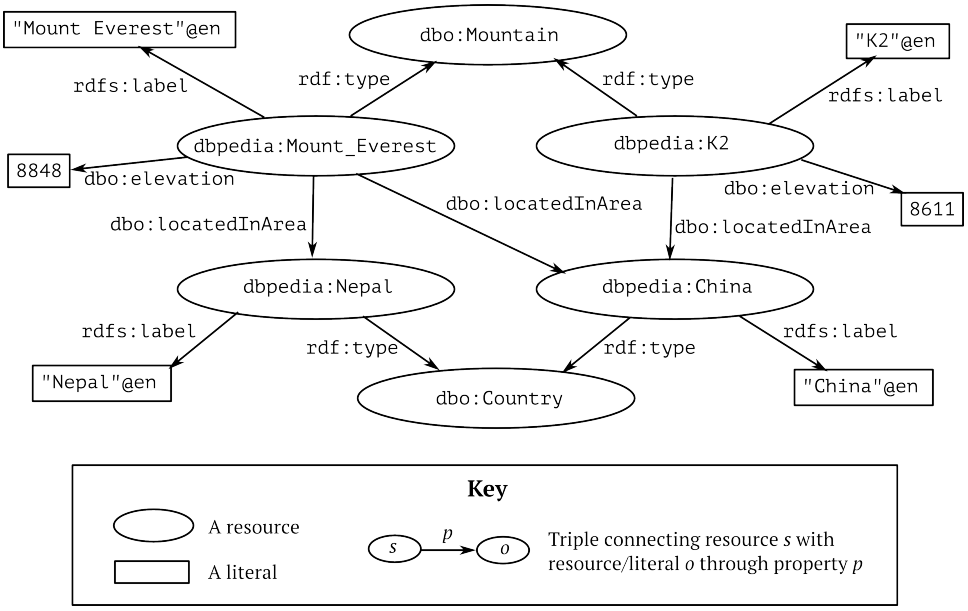


Figure 1. Part of the RDF graph served by the SPARQL endpoint of DBpedia.

a specific input graph (the default graph, a graph named by a IRI, or a generic named graph referred to via a variable). When executing a SPARQL query, the basic graph pattern is matched against the input RDF dataset and the result is a multiset of tuples, each tuple corresponding to a binding for each of the variables. Relations generated through basic graph patterns can be filtered, composed, or grouped using relational operators. The result of a SPARQL SELECT query (one of the available query types and the one that will be considered in this work) is a multiset of tuples that can be optionally ordered, making it a list of tuples.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT DISTINCT * WHERE {
  ?mount
    rdf:type dbo:Mountain ;
    dbo:elevation ?height .
}
ORDER BY DESC(?height)
LIMIT 3
```

Listing 2. SPARQL corresponding to blocks in Figure 11.

Table 1. Results of query in Listing 2.

?mount	?height
<http://dbpedia.org/resource/Mount_Everest>	8848.0
<http://dbpedia.org/resource/K2>	8611.0
<http://dbpedia.org/resource/Kangchenjunga>	8586.0

SPARQL syntax is reminiscent of SQL syntax and clauses like SELECT, ORDER BY, and LIMIT have in SPARQL the same meaning as in SQL. Basic graph patterns are represented reusing Turtle syntax for RDF graphs with the addition of variables, which are represented by labels prefixed by quotation mark and have as scope the whole query[4]. Both basic graph patterns and operations like filter or union are specified in the WHERE clause of the query. Listing 2 shows a query to get the three highest mountains. Table 1 shows the results of the query when executed on the RDF graph in Figure 1.

The *SPARQL Protocol* [14] is a protocol over HTTP used to provide a Web API to query an RDF dataset. The client sends queries to a service offering this API and gets back the results. The URI at which a SPARQL Protocol service listens for requests is called a *SPARQL endpoint*. Several RDF datasets offer publicly accessible SPARQL endpoints.

2.3. A Reference Dataset: DBpedia

Both in the examples and in the user evaluation we used a well known Linked Data source: DBpedia [15], an RDF dataset generated from Wikipedia. It is obtained through a set of scripts that extract structured data from the Wikipedia pages, especially leveraging the content of infoboxes, fixed-format tables that can be found in the right-hand corner of some articles. In the Linked Data community, DBpedia is widely considered a reference dataset because of its wide coverage and the long-term continuity of the project.

2.4. Block Programming

Programming environments for block languages typically offer a visual, drag-and-drop interface. Blocks are graphical elements associated with code snippets, which can be connected to one another like jigsaw puzzle pieces, at the same time composing in a syntactically correct form the associated code. Different kinds of blocks have different shapes, so that

[4]When using subqueries, which are neither described here nor available in SparqlBlocks, the scope of a variable is instead restricted to the current subquery, unless it is projected out in the SELECT clause

the way they can be combined is visually apparent, exposing the affordances of the language and preventing syntax errors. In Section 4.3 we will present in detail the features of block programming environments that we have adopted in SparqlBlocks.

From now on, we adopt for block programming concepts the notation used in the documentation and code of Blockly [16], a widely used block programming library on which our tool is based.

3. Related Work

We report on visual query tools for Linked Data as well as on block programming environments.

3.1. Structured Queries on Linked Data

Several interactive tools have been proposed to support structured querying of RDF data sources, at various levels of abstraction and using different paradigms. A basic distinction can be made between tools: (1) those requiring usage of SPARQL syntax and (2) those based on metaphors aimed at lowering the learning curve and providing more intuitive interaction. The first kind of UIs includes advanced editors, e.g. YASGUI [17], or integrated environments, e.g. Twinkle[5], but users still have to know SPARQL and the vocabularies used in order to design a query.

UIs of the second kind provide interaction with another — textual or visual — representation of the query, then transformed to SPARQL to be executed. Text-based UIs use forms, e.g. SPARQLViz [18], or controlled construction of natural language statements, e.g. SPARKLIS [5]. These systems do not scale well when the query complexity increases and do not easily permit code reuse. As for visual tools, most of them use a graph-based paradigm (e.g. NITELIGHT [6], QueryVOWL [19]), others a dataflow-based paradigm (e.g. SparqlFilterFlow [20]), and at least one, VQS [21], a combination of both. Graph-based interfaces fit the RDF graph pattern matching model well and they can be see as one possible RDF embodiment of the successful *query-by-example* paradigm, in which the user defines queries by perusing the same structure of the data (tables in the original definition for relational databases [22], graphs in the case of RDF). Dataflow-based interfaces, on the other side, are effective in representing SPARQL functional operators (e.g., `UNION`). However, both types of interfaces are inefficient in terms of screen real estate and may present problems with interaction.

There is an existing tool that adopts the block programming paradigm for building SPARQL queries: the SPARQL/CQELS Visual Editor designed for the Super Stream Collider framework [23]. In that tool, blocks strictly follow the language structure and syntax and the UI requires at least basic knowledge of SPARQL. Conversely, the SparqlBlocks UI is intended to to provide blocks that should be mostly self describing and usable also without previous knowledge of the SPARQL syntax.

[5]http://www.ldodds.com/projects/twinkle/

Finally, for most of the existing tools, the visualization of the result set is passive and presented in an independent container (e.g., in many Web-based interfaces, the result page replaces the query page). In our system, a query and its results share the same workspace, supporting interactive database exploration.

3.2. Block Programming Environments

The MIT Media Lab pionereed research in educational tools for teaching the use of technology and especially programming skills. This research provided the foundation for the development of StarLogo [24], then Scratch [7] — the first broadly successful visual programming software for kids — and their successor for mobile apps, MIT App Inventor [8], which was originally developed at Google and is now based at MIT. The block programming paradigm has since been applied to several scenarios.

Apart from the cited SPARQL/CQELS Visual Editor, other block programming environments have been created to design structured queries. Bags [25] and DBSnap [26] are two visual tools to design relational algebra expressions. The former has the typical block programming look and is similar to Scratch, while the latter has a peculiar tree appearance to suite the specific context. Results so far show the effectiveness of these approaches to teach relational algebra concepts [26]. In fact, SparqlBlocks' relational operators are modelled in a similar way. But many of our operators are unique to our case because: (1) we operate on a graph data model, instead of on a relational database; (2) we allow the user to perform queries on any online SPARQL endpoint, not just on a predefined set of relations.

It is also interesting also to see how block programming environments following the imperative paradigm have been extended to allow querying and manipulating data from databases. App Inventor has interfaces to online database services like Google Fusion Tables[6] and Firebase Realtime Database[7]. Punya[8], a fork from App Inventor to build mobile applications for crisis data [27], has blocks to send SPARQL queries and receive the results. In the Learning with Data project, Scratch has been extended with blocks that query a specific dataset [28]. The approach followed in all these cases is to offer a fairly thin layer of indirection among the environment and the service offered by the database. In these environments, blocks are used to execute the queries, but not to build them: either only basic queries can be executed (most of the environments) or complex queries are passed as text (e.g., SPARQL queries in Punya) that must be built by other means. In a way, these environments stop where SparqlBlocks kicks in, at the level of the logic of the query. They are thus not comparable with SparqlBlocks, being instead potentially complementary to it.

[6]https://fusiontables.google.com/

[7]https://firebase.google.com/docs/database/

[8]http://air.csail.mit.edu/punya/

4. Features of the Programming Environment

We describe here the SparqlBlocks UI, along with motivations for the main design choices. We first present the requirements we identified and the main strategies chosen to approach them. We then proceed to describe the solution, from its top-level features to the details of the visual language, describing the different types of blocks provided.

4.1. Requirements

Based on an analysis of the existing tools, and the shortcomings discussed above, we identified the following basic requirements for the design of the SparqlBlocks UI:

1. users need not know SPARQL syntax — hence visual clues and constraints should prevent syntax errors;
2. users' need for inputting text should be minimized;
3. users should have direct access to commonly used structures;
4. users should be able to use SparqlBlocks as a step to learn the SPARQL (textual) syntax;
5. users should be able to work even without prior knowledge of the dataset — hence exploratory queries should be explicitly supported.

4.2. Main Strategies

In order to meet the previous requirements, we opted for the following general strategies.

- **Explicit language affordances.** Users should be able to visually explore the set of available language elements and how they can be combined.
- **Prevention of syntax errors.** All admissible sentences in the visual language should correspond to actual queries.
- **Multiple ways of achieving a solution.** It should be possible to build a query in many different ways, as in the underlying SPARQL language, to cope with different approaches.
- **Block Programming.** The user interface should use the block programming metaphor, with language elements represented by visual, composable blocks.
- **Results view integrated with query view.** User interfaces for building queries — both textual and visual ones — typically consist of two separate areas, one for the query and one for the results. They may be shown simultaneously, one along the other, or at different times, one after the other. To facilitate the exploratory design of queries, we should instead experiment with mixing results and views in the same workspace. It should be possible to design more than one query at once and to show the corresponding results beside each query. It should be possible to use the visual elements corresponding to the results of a query to create new queries or to modify the current one.

4.3. General Block Programming Features

Based on these general strategies, we designed a concrete solution, considering also the constraints imposed by the technologies and the required effort. Figure 2 shows a screenshot of the SparqlBlocks UI. We start our presentation of the SparqlBlocks language and environment by listing the top-level features of the solution.

The following features are the ones that are common to most block programming environments.

Language elements as blocks. Language elements, represented as keywords and grammar structures in textual languages, are represented as *blocks* (see labels *a* and *b* in Figure 2). Different language elements are represented by blocks with different visual properties (shape, color, textual labels, icons, etc.). Language items that require flexibility in their definition (like literal values and variable names) are represented in the blocks through *fields* (*c*) that provide an input of some kind (free text, dropdown).

Program structure through block composition. Blocks connect to each other via jigsaw-like *connectors* (*d*) The syntax of the language is then defined by the available blocks and how they may be connected. Possible connections are hinted to the user by the visual properties of the connector (male/female, shape) and implemented by visually dragging and attaching one block to another.

Workspace as canvas. The programmer's *workspace* is shown as an unlimited canvas in which to organize and connect the blocks (*e*). The connections define the program, thus having a directly functional meaning, while the placement of blocks in the canvas space has no effect on the program: it serves the purpose of code organization and potentially of communication in case of a collaborative setting. The canvas has some of the usual controls of the window metaphor, scrollbars and zoom, along with a control of the desktop metaphor, the trash bin to delete blocks.

Visual toolbox as inventory of components. The workspace usually starts empty and blocks are dragged from the area for the *toolbox* component (*f*), where the available blocks are organized into *categories* (*g*).

Shadow blocks as defaults and examples. This is a new feature introduced in Blockly to represent defaults and usage examples. Shadow blocks (*i*) can be blocks of any of the defined block types, with a specific behaviour: they are shown in a lighter shade of the same colour and they disappear if another block is connected in their place. They are used inside other blocks to offer sensible defaults or example values, while remaining less intrusive than regular blocks used for the same purpose: regular blocks would remain on the workspace after replaced and may hence need to be deleted/moved with a further user action.

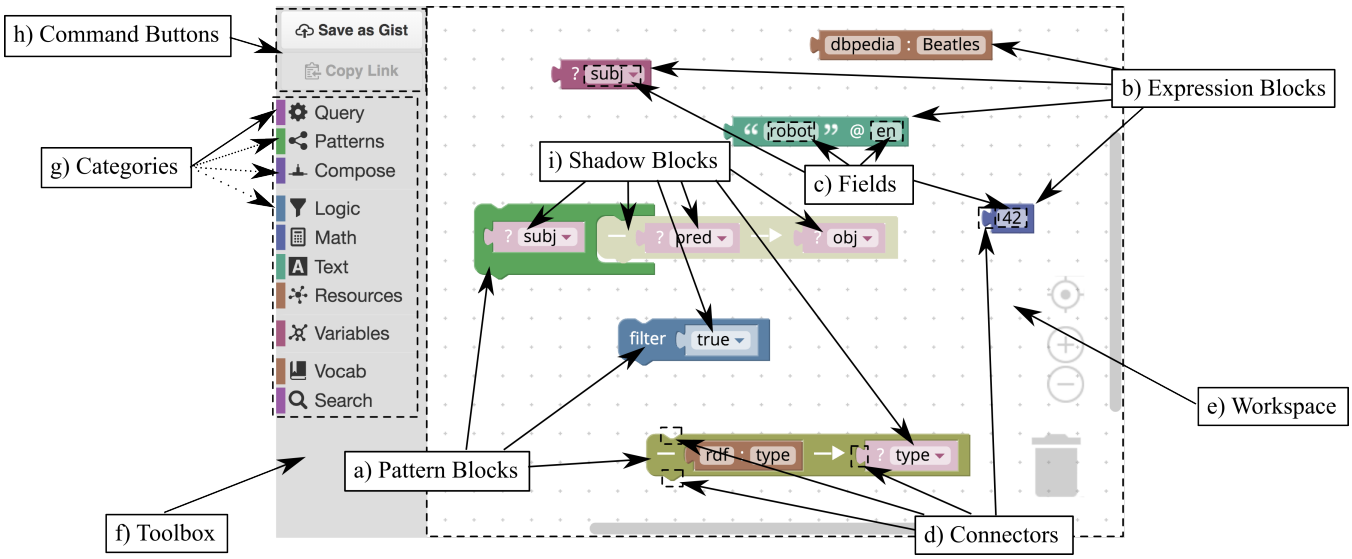


Figure 2. The user interface of SparqlBlocks.

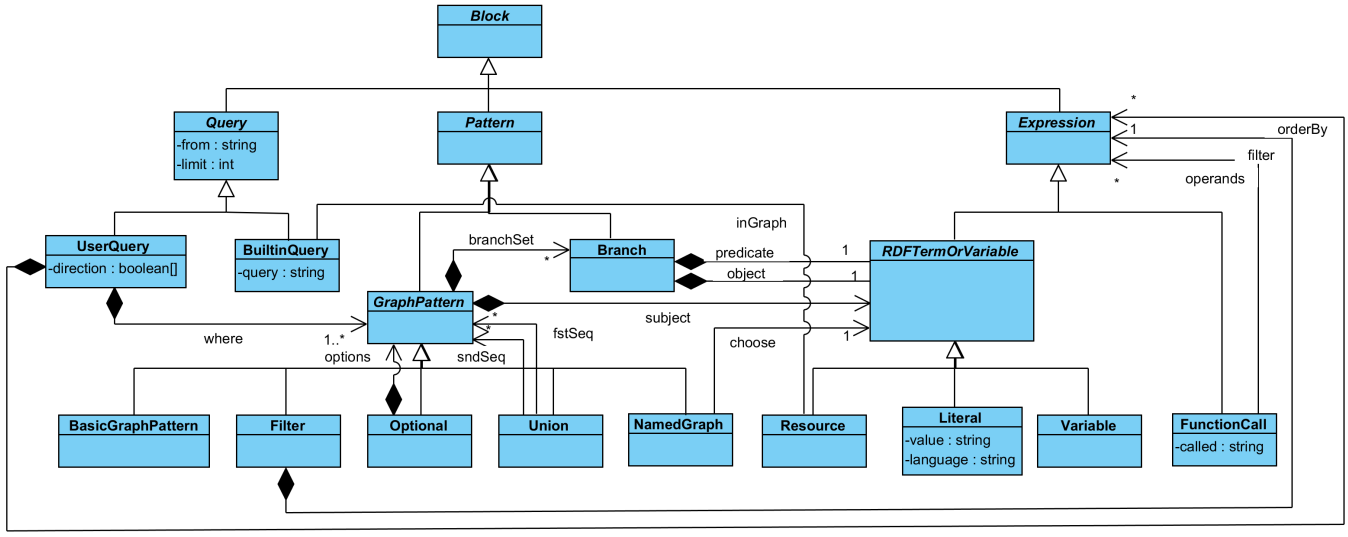


Figure 3. A class diagram synthesising information on the block types in SparqlBlocks.

4.4. Specific Features of SparqBlocks

The following features are the main ones specific to the programming environment of SparqlBlocks.

Query blocks, pattern blocks, and expression blocks. Multiple types of blocks are connected to build queries. Figure 3 shows the taxonomy of block types used in SparqlBlocks, in the form of a class diagram, also presenting information on fields and relations between block types. All the classes that are further specialised are represented as abstracts. Each block of type *Query* represents the execution of a query against a specific SPARQL endpoint (see Section 4.5). It also maintains two pieces of information:

- the *from* text field, in which the URL of the remote SPARQL endpoint has to be specified[9];
- the *limit* numeric text field, which sets the maximum number of result rows that are returned.

A specific query is defined by a structure of *Pattern* blocks that the user composes inside the query block. Each pattern block represents in turn either part of a basic graph pattern on the dataset or some operator to be applied on other patterns (see Section 4.7). Finally, *Expression* blocks are used for language structures representing scalar values, i.e. single RDF terms, variables, and calls to SPARQL functions and operators (see Section 4.6). Each of these three main types of blocks has different external connections, so that users can immediately distinguish the role of these categories: query blocks have

[9] In the examples in this paper the value of the field is always the DBPedia endpoint [15].

no external connections, pattern blocks have top and bottom connections, expression blocks have one left connection. These visual cues are important, but not sufficient, as specific subtypes have different roles in the language. So, when a block of a specific type is expected on a connection end, the systems checks if the block being connected is compatible. If the check fails, the block is "repelled" by the connection, thus signalling to the user that the connection is not permitted.

Results of execution as blocks. The results of a query are shown as blocks that can be used in the workspace, either to modify the query itself, or to build new queries (see Figure 4d). This feature supports an exploratory approach to the design of the queries.

Queries and results integrated in the workspace. To support the reuse of query results as blocks, results are part of the workspace itself, attached to the corresponding query. This enables to have multiple queries and their results directly visible and actionable together in the workspace, avoiding the need to switch between different query outputs.

Live query execution. Queries in the workspace are immediately executed when created and every time they are changed. This means the user has immediate "real time" feedback[10] and may change the query accordingly.

Export of queries and results. Users can export queries to see them represented in the SPARQL syntax. Proficient users may use this feature to sketch a new query visually and then continue working on it in SPARQL, while others may profit from this feature to learn SPARQL syntax. Result sets may also be exported for reuse.

Built-in queries. The blocks defined so far offer a bottom-up approach in which queries are built from components that represent basic functional elements. Along with those basic components, the tool provides a set of higher level blocks to perform specific queries that are often needed. These are the queries used to look for classes and properties in the used vocabulary and for specific resources in the data (see Section 4.8).

4.5. User Query Blocks

The language mimics the structure of SPARQL. As in SPARQL, the simplest query is defined through a basic graph pattern, whose simplest form is in turn a triple pattern. The blocks in Figure 4a form a triple pattern that matches any triple having `dbr:The_Beatles` as subject and `dbo:formerBandMember` as predicate. For each such triple, the variable `member` is bound to the corresponding object (see Section 4.7 for details on pattern blocks).

In order to execute a query defined according to this pattern, the *user query block*[11] (see Figure 4b) is defined. This block, representing a whole query on its own, has no external connections, as it is not meant to be stacked or put inside other blocks. Instead, it has the following fields and connections, besides *limit* and *from*, inherited from the abstract class *Query*:

- the *where* connection, hosting a sequence of graph pattern blocks (see Section 4.7) describing the query;
- a variable number of *orderBy* connections, in each of which an expression block (see Section 4.6), typically a variable, represents an ordering criterion;
- a drop down *direction* field for each *orderBy* connection, to select the direction (ascending/descending) of that specific criterion;

The number of *orderBy* connections is variable because a desired ordering may be obtained by a sequence of criteria (e.g., from a dataset of some people and their telephone numbers, to generate a telephone directory we may order first by surname and then by name when the surname is the same). To avoid to overly complicating the ordering mechanism, this variability is managed in a simple way: even if the ordering is not required, an empty connection is already available when the query is dragged to the workspace; as soon as a block is connected to this connection, a new empty connection is created to the right and so on, so that an empty connection is always available. This is the only case in which we allow an empty connection[12].

As soon as a pattern is attached to a query block, the query is run on the remote dataset. Figure 4c shows the query block connected to the pattern and "waiting for the results". When results are ready (as in Figure 4d), they appear in tabular format attached to the lower connector of the query block. Each row of the results represents a different matching of the pattern, with the corresponding variable bindings. The single data items are represented as expression blocks (specifically, resource and literal blocks) that can be dragged from the result set to create other queries (or even to modify the one that generated them).

If the content of a query block is modified, the new query is executed immediately and results shown as soon as they are available[13]. Query execution is in any case non-blocking, i.e. the UI is reactive and operational even if one or more queries are being executed.

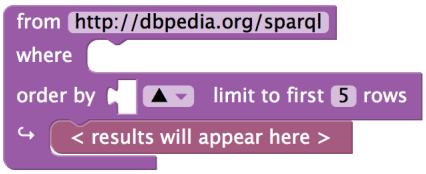
[10]The time to get the results from a remote SPARQL endpoint obviously depends on bandwidth and server response times for the specific query. Nevertheless, as the queries are executed asynchronously, possible delays do not affect the responsiveness of the user interface.

[11]The complete name "use query block" is here used to distinguish this block from the built-in query blocks described in Section 4.8. In the rest of the paper, this block is also called simply "query block" when the meaning is not ambiguous.

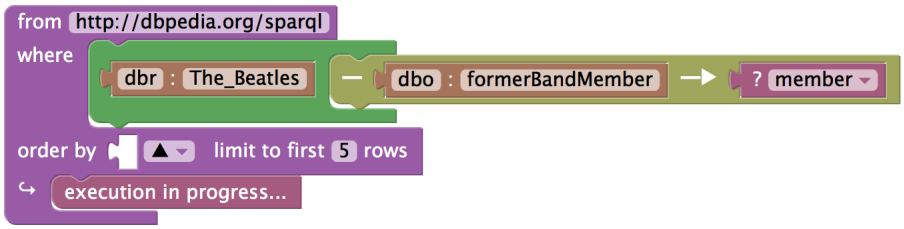
[12]This choice has admittedly the potential to confound the user in believing that a block is strictly required in that empty connection. Nevertheless, the other alternatives that have been evaluated (like using the Blockly mutator mechanism, that requires opening a sort of mini-workspace used to configure a block through blocks representing parts of it) pose other forms of cognitive overhead. In the evaluation described later in this paper, this choice did not cause confusion in the users.

[13]When the query is modified, the table with the previous result set is also immediately wiped (replaced with the "execution in progress..." block in Figure 4c) to avoid any confusion.

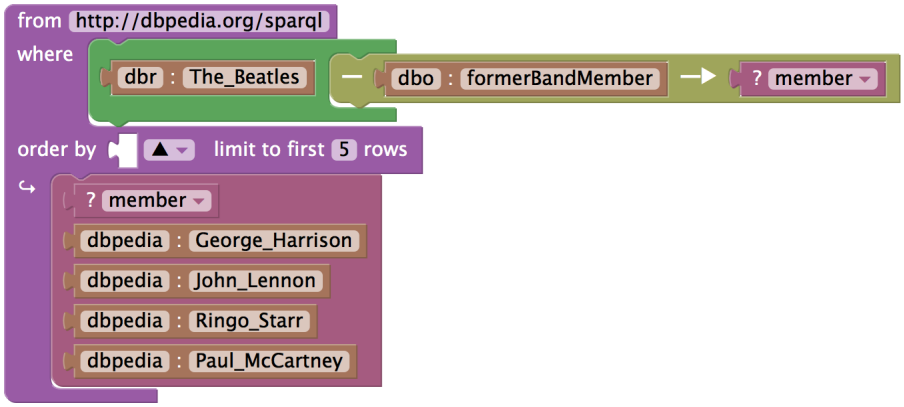
(a) Blocks for pattern



(b) Block for query



(c) Blocks joined and query in execution



(d) Query executed and results shown

Figure 4. Execution of a query in SparqlBlocks.

4.6. Expression Blocks

Expression blocks (see *b* in Figure 2) are visually identified by having horizontal output (male) connectors and represent the elements of the SPARQL language that correspond to scalar values. The expression blocks are further classified in *resource blocks*, *literal blocks*, *variable blocks*, and *function call blocks*.

Resource blocks may be specified either with a full IRI or in prefix notation (e.g. `dbpedia:Beatles` in Figure 2). SparqlBlocks offers ways to look for resources in a dataset and to cast them directly as usable blocks (see Sections 4.5,4.8, and categories *Resources* and *Vocab* in Section 4.9). As a further option, users can look for the IRI through external means (e.g., a Linked Data browser, like the one used as front-end for the resources by DBpedia) and then copy/paste it on a blank resource block; if a known namespace is recognized, the IRI is converted on the fly to the prefixed version.

Literal blocks hold strings with an optional language tag (e.g. `"robot"@en` in Figure 2), booleans, or numbers (e.g. `42` in Figure 2).

Variable blocks are used as wildcards in patterns. They are represented by specific blocks with a drop-down menu, that can be used to create a new variable or use one of the existing ones (`subj` in Figure 2).

Function call blocks represent calls to SPARQL functions and operators (`lengthOf`, `absolute`, `<`, `+`, etc.). Each function call block has a number of internal connections equal to the number of parameters (operands) that the corresponding function (operator) has. Each internal connection accepts an expression block.

Even if the output connectors for all the value blocks have the same appearance, they cannot all be connected in the same ways, because of the different roles in the language. RDF terms and variables can be used in basic graph patterns (with some limitations, see Section 4.7), while functions/operators cannot. The latter are permitted whenever, in SPARQL, expressions can be used. Currently, in SparqlBlocks, this corresponds to the case of filters (see Section 4.7) and ordering.

4.7. Pattern Blocks

Pattern blocks (see *a* in Figure 2) are visually identified by having vertical connectors (female on top and male on bottom)

that allow them to be stacked one under the other and used in specific contexts, like the *where* connector of the query block. As for the value blocks, even if the output connectors of all the pattern blocks have the same appearance, they cannot be connected all in the same way. In the case of pattern blocks there are two strictly separated types: the *pattern blocks*, and the *branch blocks*.

(a) Graph Pattern (b) Branch

Figure 5. Blocks for basic graph patterns.

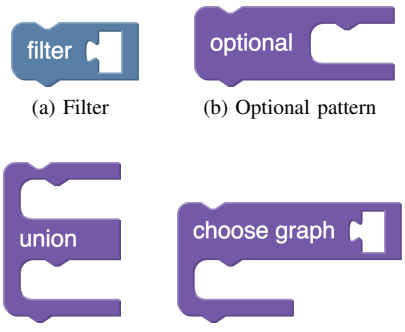
Basic graph patterns are built using the blocks shown in Figure 5, starting from the **graph pattern block** (Figure 5a), used to group triple patterns that have a resource or variable in common. It has two internal connectors:

- a *subject* connector that accepts a resource block or variable block;
- a *branch set* connector for adding triple patterns related to the common resource/variable, accepting a sequence of branch blocks.

A **branch block** adds a triple pattern for which the common resource/variable of the basic graph pattern block is the subject. It has two internal connectors:

- a *predicate* connector that accepts a resource block or a variable block;
- an *object* connector that accepts a resource block, a literal block, or a variable block.

Hence, following SPARQL syntax (see Section 2.2), branch blocks can be stacked into basic graph pattern blocks to join multiple triple patterns that have a resource/variable in common. Basic graph pattern blocks can be in turn stacked one upon another to build more complex basic graph patterns.



(a) Filter (b) Optional pattern

(c) Union of two patterns (d) Named graph pattern

Figure 6. Pattern blocks beyond basic graph patterns.

Apart from the basic graph patterns, other pattern blocks can be used to build more complex queries:

- the **filter block** (Figure 6a), filters the matchings of the graph pattern sequence[14] according to a given condition (which must evaluate to true to pass the filter); the *filter* connector accepts an expression block that will be evaluated as boolean;
- the **optional block** (Figure 6b) adds a sequence of graph pattern blocks as optional in the sense that its matching is not required but if matched the variables will be bound accordingly;
- the **union block** (Figure 6c) adds two sequence of pattern blocks as alternatives in the sense that the union of the matching of the first and the second pattern will be considered;
- the **named graph block** (Figure 6d) selects a specific named graph of the source RDF dataset (via the *choose* connector, accepting a resource block or a variable one), for the sequence of contained graph patterns.

4.8. Built-in Query Blocks

There are some queries that are useful for every RDF dataset, regardless of its specific domain, for example queries used to explore the vocabulary used in a dataset, i.e. the IRIs used for classes, properties, and other important resources. Hence, we present the user with a library of **built-in query blocks** that offer pre-built queries for common tasks. The *BuiltinQuery* type of blocks inherits from *Query* the *from* and *limit* fields, to specify the SPARQL endpoint and the maximum number of result rows, respectively. These blocks also have an *inGraph* connector that optionally accepts a resource block to select a named graph from the dataset (if not used, the default graph is selected). As with user query blocks, the built-in query blocks are executed as soon as the required fields and connections are filled and are executed again each time some of the fields or connections are changed. The results, also in this case, are shown as a table connected to the lower part of the query.

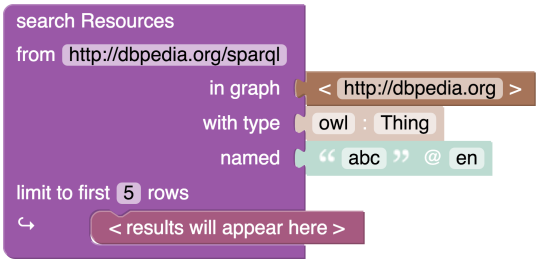
Currently, the library comprises the following blocks[15]:

Search resources block, having a *withType* connector, which accepts a resource block and a *named* connector, accepting a literal block with text (Figure 7a). It looks for resources of the specified type and containing the specified text in their label. The columns of the table of results are *?resource* and *?label*.

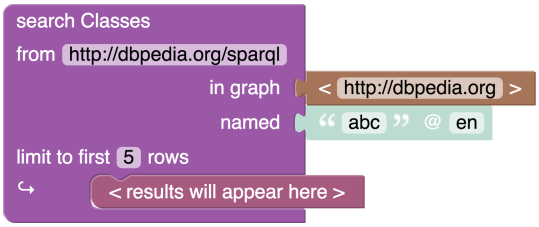
Search classes block, having a *named* connector, which accepts a literal block with text (Figure 7b). It looks for classes whose label contains the specified text. The typical use of a class in a query is to look for instances of that class. So, to give a sensible default and to reduce the potential errors, for every result there is a column with a pre-built pattern looking

[14]The filter block operates on the graph pattern sequence this block is part of, i.e. the sequence comprising blocks stacked both above and below the filter block.

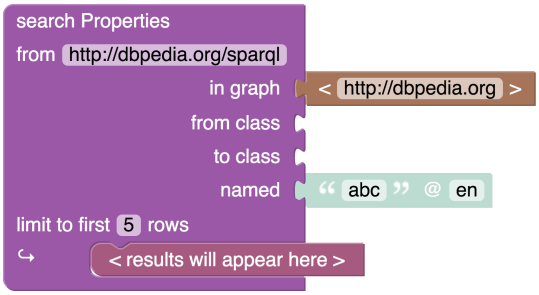
[15]To keep the diagram of Figure 3 simple, we do not detail these specialised classes.



(a) Search Resources



(b) Search Classes



(c) Search Properties

Figure 7. Built-in query blocks.

for instances of that class. The columns of the table of results are *?class*, *?label*, and *?pattern*.

Search properties block, having the optional connectors *fromClass* and *toClass*, both accepting resource blocks, and a *named* connector that accepts a literal block with text (Figure 7c). It looks for properties such that the specified text is contained in their label, optionally going from a given class to another given class. The typical use of a property in a query is to look for pairs of resources connected by that property. So, in analogy with the search classes block, there is a column with a pre-built branch that uses the property. The columns of the table of results are *?property*, *?label*, *?domain*, *?range*, and *?branch*.

4.9. Organization in Categories

Blocks are visualized and organized (in the toolbox) according to their role. The *query* block is provided under the category *Query*. Blocks used to build graph patterns are grouped under the category *Patterns*. The *optional*, *union*, and *named graph* blocks are under the category *Compose*. The categories *Logic* (also containig the *filter* block), *Math*, and *Text* contain *literal*, *operators* and *functions* blocks, organized according to their types. Resource and variable blocks are available in the respective categories *Resources* and *Variables*. The category *Vocab* contains some pattern and branch blocks for standard vocabularies, organized by categories named after each of the vocabularies. Finally, all the built-in query blocks are grouped under the category *Search*, as they all deal with retrieving resources.

5. Implementation

Blockly [16] is a JavaScript library for block programming maintained by Google, on which several tools — including MIT App Inventor 2 — are based. It provides a set of basic blocks covering the structure of typical imperative programs. Most importantly, it is also extensible programmatically to define new blocks. SparqlBlocks is based on an extension of the Blockly JavaScript library, working entirely on the client side. We extended the library to supply the specific blocks needed for SPARQL queries and execution. We also added the necessary code to generate SPARQL fragments from the blocks. The SPARQL execution block listens for changes in its query connection; each time the query changes, the corresponding SPARQL query is generated and sent to a SPARQL endpoint. The SPARQL endpoint to be used is set as a field of the execution block. The results are used to dynamically generate the result block and its sub-blocks. The standard prefix definitions from *prefix.cc*[16] are used to add prefix declarations in the query sent to the endpoint and to convert the IRIs in the result to the prefixed notation.

SparqlBlocks is available online[17] and can be used directly to query any public SPARQL endpoint. Furthermore, the code is free and publicly available[18].

6. Design Evaluation

Before organizing an evaluation with the users, SparqlBlocks was formally evaluated "in house" using the relevant heuristics. We will first present its evaluation in terms of cognitive dimensions and then analyze its affordances in the specific context of query design.

6.1. Cognitive Dimensions

Cognitive dimensions [29] is a framework developed by Green for the analysis of the properties of programming languages. Green and Petre later derived from that general framework a more specific one targeting visual programming languages [30]. An initial evaluation of SparqlBlocks was performed based on the cognitive dimensions deemed as relevant to our context.

Consistency. The block syntax favours internal consistency; external consistency is satisfied with respect to SPARQL textual syntax because the structure is maintained and, partially, with respect to other Blockly-based languages because the appearance and behaviour of basic expressions is preserved.

[16] http://prefix.cc/
[17] http://sparqlblocks.org/
[18] https://github.com/miguel76/SparqlBlocks

Diffuseness. Representation through blocks requires more space than textual representation. However, SparqlBlocks is designed to be efficient in terms of graphic entities, with the visual appearance of each block tuned to minimize the space required for it.

Error-proneness. The chance of making syntax errors is extremely reduced, compared to textual syntax: the affordances are first coarsely constrained by the differences between shapes, then by other visual cues, and finally by the behaviour of visual elements. The use of results for refining existing queries or creating new ones reduces the chance of mismatches between queries and data structure.

Hidden dependencies. Dependency and scope of variables are strictly related to the SPARQL query structure that is closely followed by the block structure.

Premature commitment. Queries can be as easily modified, decomposed, and recomposed at any time, the user is thus not forced in any way to stick to earlier decisions.

Progressive evaluation. A trial and error approach is favoured by the automatic execution of queries in query blocks. While the user builds a query, she can immediately see its results. Furthermore, the user can try, as separate queries, any graph pattern that will eventually become part of the complete query. This method is favoured by the possibility of having multiple queries in the workspace.

6.2. Query Design Affordances

To further argue in favor of the proposed paradigm, we show some of the affordances of SparqlBlocks related to the query construction process. We analyse the basic operations that a developer — whether an expert or a beginner — needs to perform. The underlying idea is that designing a query is an interactive process in which the query emerges from a sequence of basic operational steps, possibly retracting or modifying the effect of previous ones, without having to be completely detailed in advance.

Generalization/Specialization. The design of a query may proceed from a generic version, with a minimal number of constraints, designed to start getting some data. Then, by adding constraints, the query will gradually become more selective. This process, which we call *query specialization*, is supported by reducing the free variables (through replacement with constant values or with already used variables), by adding filter blocks, by adding more complex basic graph patterns to be fulfilled, and by moving patterns out from an *optional* or *union* block. The usage of blocks also for representing query results allows the user to easily identify and manipulate the specific blocks corresponding to resources or literals needed to replace variables or to create filter expressions. An example of *query specialization* is the transformation from the query in Figure 8a (which asks for some mountains and their locations) to the query in Figure 8b (where only Brazilian mountains are selected) by dragging the resource `dbpedia:Brazil` from the results in the place of the variable `area`.

The design of a query may also start from a specific query on known data and then proceed by relaxing some constraints to include a greater set of results. This process of *query generalization* is supported by replacing constant values with variables, by removing filter blocks, by removing graph patterns (or part of), and by moving graph patterns under an *optional* or *union* block. These actions correspond directly and naturally in SparqlBlocks to the removal of blocks or the creation of new ones for the variables. An example of generalization can be the reversal of the specialization example, i.e. from the query in Figure 8b to the one in Figure 8a, by dragging on the workspace a new variable block to replace the resource `dbpedia:Brazil`.

Composition/Decomposition. A complex query may be created by composing different queries together, so that, for example, the output of separate components of the query can be checked before composing them. As the proposed UI permits multiple queries to be built and executed in the same workspace, composition is directly executed by joining blocks from different queries[19]. For example, to build a query for European mountains, the user may first design a query to get mountains (Figure 8a) and another query to get the European countries (Figure 9a). The complete query can be composed by dragging the graph pattern of the latter query and adding it inside the former one and changing the `area` variable to `country` (Figure 9b).

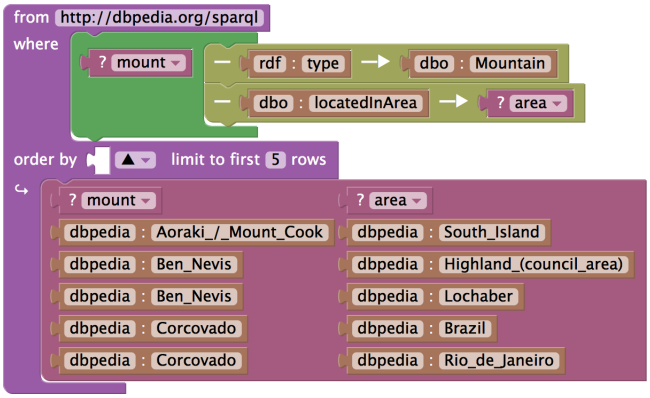
The reverse operation is to decompose a query in smaller ones. From the point of view of the UI the actions are similar to the ones needed for composition: dragging blocks and creating some new ones. Reversing the example of composition gives an example of decomposition.

Stepwise Querying. Sometimes a query design comes after some exploratory steps that identify relevant resources, classes, or properties. This approach is supported by permitting values dragged out of the result set of a query to be used by other queries. The old query may then be removed or simply kept aside for further use. For example, Figure 10 shows the use of a built-in query block to get some candidate classes to represent the set of mountains, by searching among available classes using their *label* attribute. As soon as `dbo:Mountain` is identified as the relevant class, the corresponding resource block may be dragged away and used for follow-up queries, as in Figure 8a (in the object connection of the `rdf:type` branch of the basic graph pattern).

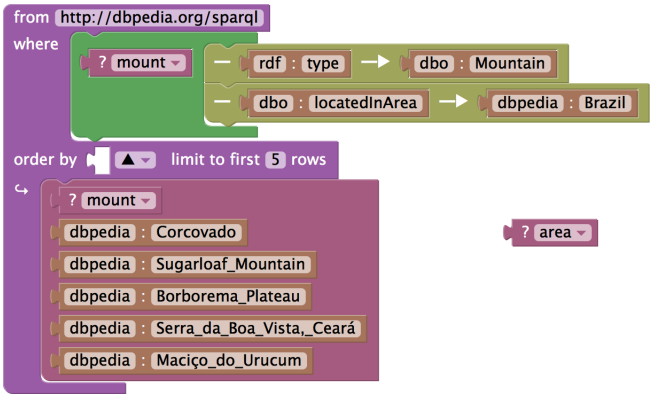
7. Iterative Design

SparqlBlocks has been tested with users to evaluate and identify possible improvements. An informal evaluation has been carried on with expert users, whose observations have led to several improvements in the tool.

[19] Blocks may also be duplicated to preserve the original queries.

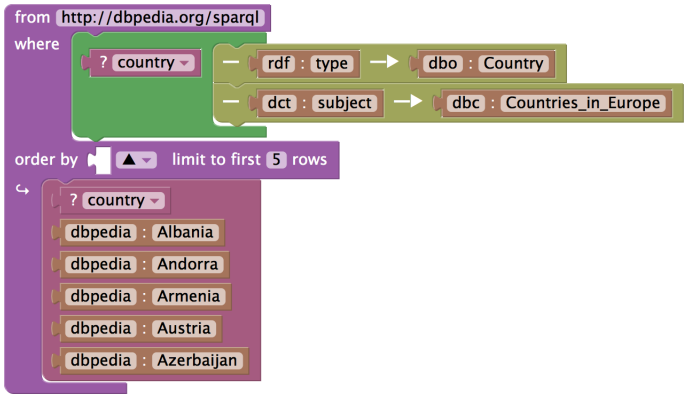


(a) Query to get mountains, generalization of query (b).

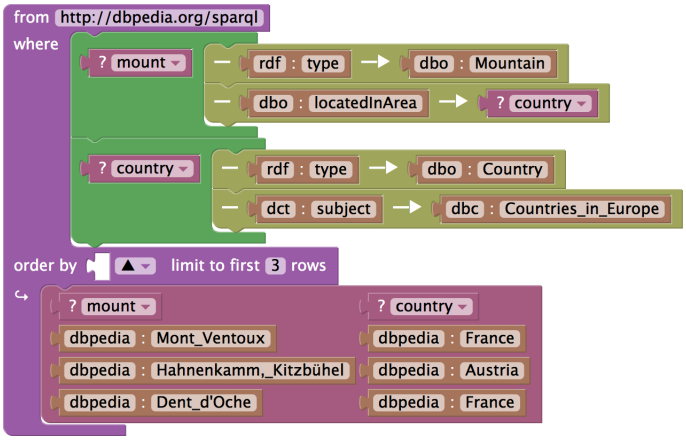


(b) Query to get Brazilian mountains, specialization of query (a).

Figure 8. Example of specialization/generalization.



(a) Query to get European countries.



(b) Query for European mountains, a composition of (a) and (c).

Figure 9. Example of query composition in SparqlBlocks.

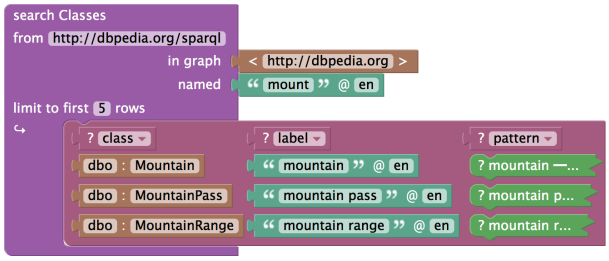


Figure 10. Getting classes for “mount”.

7.1. Feedback from Users

After the first version of the tool was finished, we presented it at the International Semantic Web Conference (ISWC) 2015 and at the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) 2015. These are top conferences for, respectively, the Semantic Web and Visual Languages research communities.

Demonstrating the tool in these two venues allowed us to gather feedback from experts of two complementary aspects of the user interface: its efficacy as a query design tool for the Semantic Web and its expressivity as a visual language (and specifically as a block programming language). The tool was presented to interested users in the informal way typical of software demonstrations in conferences. About a dozen of users were invited to use the application and to share their feelings about it in a colloquial fashion.

The following are the main reactions that we collected about the system as a whole:

- the UI was seen as **appealing** by all the users;
- users had **mixed feelings about ease of use**: although they felt the system was easy to use, they were not immediately confident in using it, due to the perceived complexity of the system in terms of available blocks and possible combinations;
- most users considered the tool as **novel**, especially for the use of the blocks in the results.

Detailed findings on specific UI elements are presented in the next subsection, along with the performed changes.

7.2. Changes to the User Interface

The overall positive reactions concerning novelty and appeal of the SparqlBlocks environment elicited subsequent work to improve it, especially in the areas that were perceived by the users as lacking. The following changes were introduced with the aim to achieve a better user experience.

Join query block with execution block to reduce the number of different available blocks and types of connections. The blocks were distinct because of their different conceptual role, one representing a complete SPARQL query (containing a sequence of patterns and fields for ordering and limiting the results) *vs*. the other representing a SPARQL endpoint (containing a query, the field for the endpoint URL, and a generated output field for the results). In practice, the execution block made sense only connected to the query block (and vice versa) and the chance of moving an entire query from an endpoint to another was not exploited enough to justify the increased complexity. So the two blocks were joined together in a new query block that includes also the text field for the URL of the endpoint and the generated field for the results.

Simplify table of results to avoid confusion, reduce the space occupied, and the cognitive load. The appearance was changed from the previous complex block structure to a single *results* block containing directly the draggable blocks, still arranged in a table. While before there were a large number of blocks used without functional reason, after the change the only non-draggable block is the top one[20].

Replace pre-built queries with built-in query blocks to reduce the cognitive load for both beginners and experienced users. Initially, instead of built-in query blocks, we provided pre-built queries exposing all the details. The idea was to give the chance to not only use them but also to learn from their structure and change them according to specific needs. The pre-built queries occupied a lot of screen space and represented a heavy cognitive load for all the users; beginners were especially not confident of where the parameters for the query had to be placed. After the change, the predefined queries are represented by single blocks (the built-in query blocks) in which the only exposed connections and fields correspond to the parameters that must be set to use the queries.

Reduced use of predefined values to reduce confusion in perceived affordances and avoid the perceived extra work needed to replace values. Blockly allows the toolbox to contain not just basic blocks, but pre-connected groups of blocks. In the first version we used this feature to provide default and example values for basic blocks. That was found to be to confusing, so we reduced the use of these pre-connected sub-blocks, in several cases replacing them with shadow sub-blocks.

[20]Due to the way the layout of blocks is managed in Blockly, it is not currently possible to avoid the use of this last non-functional block without extensive implementation work.

Support for building queries from classes and properties. The typical use of a class in a query is to look for instances of that class. So, to give a sensible default and to reduce the potential errors, for every result of a black-box query searching for classes there is a column with a pre-built pattern looking for instances of that class. Similarly, the typical use of a property in a query is to look for pairs of resources connected by that property. So, for every result of a black-box query searching for properties, there is a column with a pre-built branch that uses the property. In both cases, those added blocks may be reused directly in a query.

Directly offer highly used classes and properties to avoid unnecessary effort to discover or manually write down them. Classes like `foaf:Agent` and properties like `rdfs:seeAlso` are widely used in RDF datasets so it makes sense to have them at hand, rather then discovering them via black-box queries. The new toolbox category *Vocab* contains a set of subcategories corresponding to different vocabularies. For each vocabulary a set of common classes and properties is available, already available in the form of patterns and branches.

Keep in memory resources that have being used during the session, to recover specific resources, classes, properties that may have been found and then lost. The category *Resources* contains now both the blank resource blocks (one in prefix notation and one in full URI notation) and all the previously created resources, the most recent ones on top.

8. User Evaluation

After the updates described in Section 7, the SparqlBlocks UI was evaluated by a group of Master students, PhD students, and young researchers of the Computer Science Department of Sapienza, University of Rome. They all had a solid background in Computer Science but they were not conversant with specific Semantic Web technologies like RDF and SPARQL. This evaluation was designed as a formal assessment involving the solution of three tasks and a questionnaire.

8.1. Setup

Users went through an online questionnaire with questions and activities organized in a sequential manner:

1. rate their own background expertise on Semantic Web concepts (Semantic Web, Linked Data) and technologies (RDF, SPARQL), databases (relational databases, graph databases, SQL), and block programming environments (by indicating known environments);
2. rate the expected complexity of each of the three tasks beforehand, without having seen the tool yet, and figuring out how difficult it would be to look for the reply with access to the Web;
3. follow an interactive tutorial designed to teach basic use of the SparqlBlocks environment;
4. execute three tasks with the tool;

5. rate the actual complexity of executing each task;
6. rate the confidence in the result obtained for each task;
7. rate the tool for perceived easiness of use, appeal, and novelty;
8. write open-ended comments on SparqlBlocks.

All the ratings were on a scale from 0 to 6. For the block programming environments, participants were asked to select the environments that they knew about, among a list including the most common ones (Scratch, MIT App Inventor, etc.) and the option *other* to include other environments.

11 people participated in the evaluation. Even if the whole evaluation could have been executed online, in most of the cases there was a facilitator so that further qualitative data and feedback could be gathered. Only 3 participants executed the tasks on their own. When present, the facilitator assumed also the role of giving real-time help on the elements of the tool, as currently there is not a complete help system embedded within the tool. In any case, no explicit suggestions were given on how to solve the tasks.

There were no time limits, but participants were informed that the estimated duration of the test (including questionnaire, tutorial, and tasks) was 50 minutes. They were told to try to solve as many tasks as they could. They had thus the implicit option to stop before achieving the solution of all the tasks and record what they completed.

8.2. Tasks

The tasks were chosen so as to require the design of structured queries of increasing complexity. Queries were run on the DBpedia SPARQL endpoint. For each task, users were asked to find the resource corresponding to the result by building appropriate queries with the system:

1. third highest mountain in the world;
2. lowest mountain above 8,000 m in the world;
3. third highest mountain between China and Nepal.

These three main tasks were intertwined with some helper tasks that required the user to find the elements that were useful to design the main queries: the class used for mountains and the one for countries, the property used for the elevation of mountains and the one used for their location, and the resources used for China and Nepal. These helper tasks required the appropriate use of black-box query blocks that search for classes, properties, and resources. These classes, properties, and resources could have been given directly to the participants as basic blocks with which to build the queries, but we preferred to test the more realistic situation in which the user has no prior logic of the vocabulary used in the dataset.

Task 1 required building a query with a graph pattern to get all the mountains and their elevations, then to set the ordering to decreasing in respect to the elevation (see Figure 11). Task 2 required adding a filter to that query, so that only mountains with elevation more than 8,000 m. were selected, and to invert the ordering to increasing (see Figure 12). Task 3

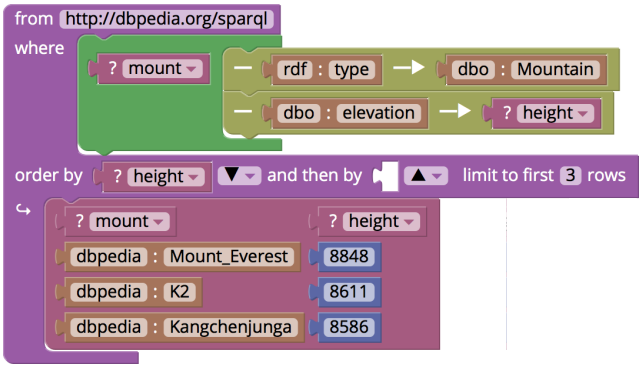


Figure 11. Query to solve Task 1.

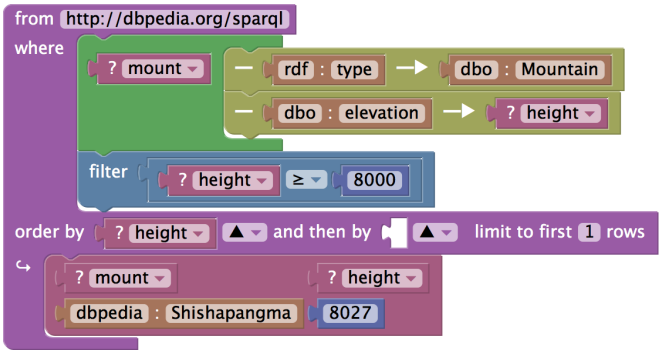


Figure 12. Query to solve Task 2.

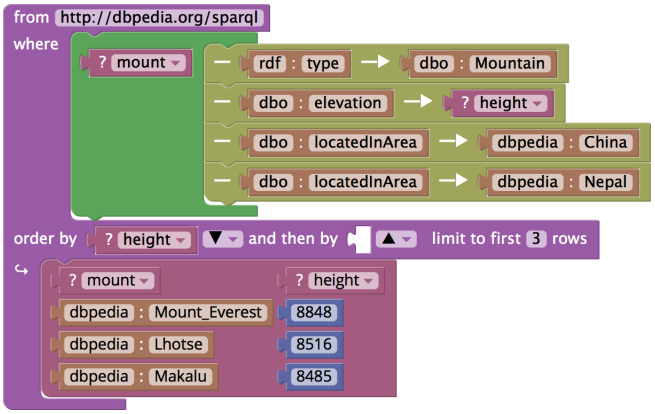


Figure 13. Query to solve Task 3.

required extending the graph pattern such that the mountains selected are required to be located both in China and Nepal (see Figure 13). Queries similar to the ones required in Tasks 1 and 2 were shown in the tutorial (using a text variable instead of a numeric one), while no query in the tutorial had a characteristic required in Task 3: two branches with the same predicate but different objects. Listings 2 (earlier on page 3), 3, and 4 show the SPARQL queries to solve the three tasks. They are the textual counterpart of the block configurations shown in Figures 11, 12, and 13.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT DISTINCT * WHERE {
  ?mount
    rdf:type dbo:Mountain ;
    dbo:elevation ?height .
  FILTER(?height >= 8000) .
}
ORDER BY (?height)
LIMIT 1
```

Listing 3. SPARQL corresponding to blocks in Figure 12.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbpedia: <http://dbpedia.org/resource/>

SELECT DISTINCT * WHERE {
  ?mount
    rdf:type dbo:Mountain ;
    dbo:elevation ?height ;
    dbo:locatedInArea dbpedia:China ;
    dbo:locatedInArea dbpedia:Nepal .
}
ORDER BY DESC(?height)
LIMIT 3
```

Listing 4. SPARQL corresponding to blocks in Figure 13.

9. Quantitative Results

All the participants filled out the questionnaire, followed the tutorial, and were able to solve at least the first task. The average total completion time was slightly higher than the estimated 50 minutes, around 1 hour. 9 participants out of 11 were able to solve all three tasks, while the other two solved only the first task.

We proceed to show in more detail the quantitative data gathered through the questionnaire and measuring completion times for the different parts of the test. The group of participants is small and thus these data do not have statistical significance. Nevertheless, the following quantitative analysis, along with the qualitative analysis presented in Section 10, is useful to discuss the effectiveness and challenges of Sparql-Blocks and to gain insights on aspects to consider for such tools.

9.1. Questionnaire

The plots in figures 14 through 16 are histograms that show the distribution of the ratings given by the participants. The horizontal axis is rating on a scale of 0 to 6, while the vertical axis is the number of participants who gave that rating[21]. Figure 14 shows the distributions for participants' self-assessed relevant background knowledge, while Figure 15 shows, for each task, the distributions for *expected* (assessed before using the tool) and *actual* complexity (using the tool) and confidence of the results obtained (with the tool). Figure 16 shows the distributions for the rating given by the participants to the various aspects of the proposed user interface: ease of use, use

[21]The total height of all the bars in each plot is thus 11 (for the 11 users).

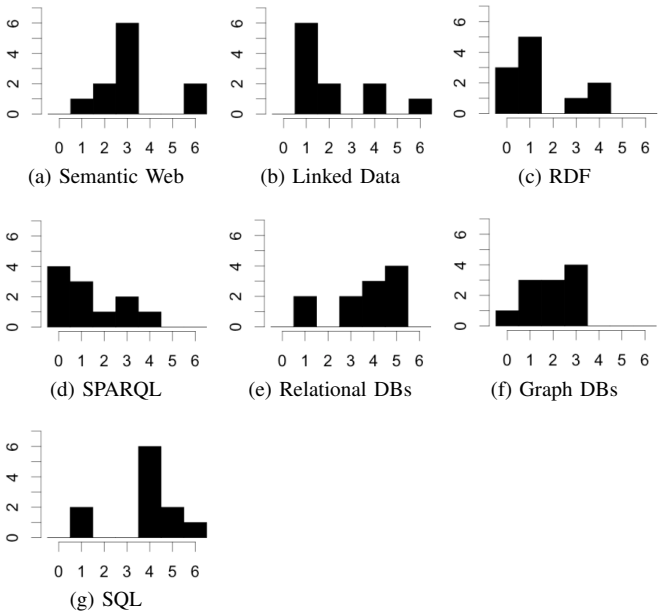


Figure 14. Distributions of knowledge of technologies

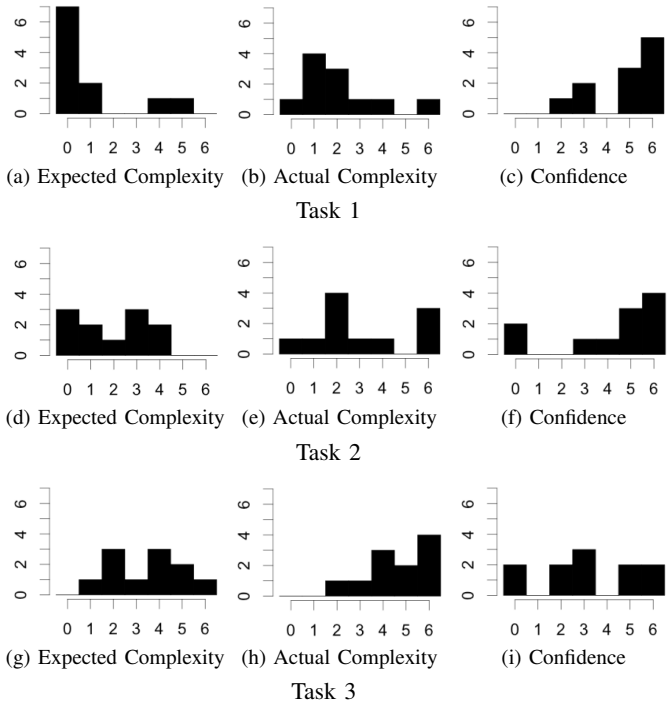


Figure 15. Distributions of Complexity by Task

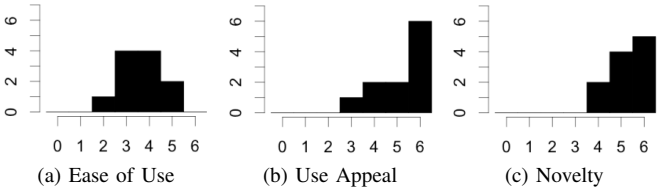


Figure 16. Distributions of Ratings for Tool Aspect

appeal, and perceived novelty. Only two participants reported to have used at least one block programming environment: in both cases they had used Scratch, in one also Blockly Games.

9.2. Completion Times

Due to a bug in the logging of the tool we were not able to record the detailed timing for the test, so for most participants we have just the whole time of completion (including questionnaire and tutorial) that ranged between 46 and 76 minutes, with an average of approximately 62 minutes. For 5 of the participants, anyway, we have the breakdown of times among questionnaire, test, and tutorial, which should give a quite accurate estimate of what happened in the other cases. The time to reply to the questionnaire was consistently around 4-5 minutes; the tutorial took between 19 and 33 minutes to be completed (with an average of approximately 25 minutes); the three tasks were solved between 23 and 48 minutes (with an average of approximately 35 minutes). Among the tasks, the first two required roughly the same amount of time (from 5 to 10 minutes each) while the third required roughly triple effort (from 15 to 30 minutes).

9.3. Discussion

Regarding background knowledge and recalling that all the participants are enrolled in the department of computer science as at least Master students, it is not surprising that relational databases (Figure 14e) and SQL (Figure 14g) are quite well known subjects. Most of the participants declared to have a midway expertise relating the Semantic Web (Figure 14a), but did not know about specific Semantic Web technologies like RDF (Figure 14c) and SPARQL (Figure 14d), with around two thirds of the participants selecting 0 or 1. Also the Linked Data expertise (Figure 14b) is fairly low (again, around two thirds selecting 1), even if it is a term that shares a good part of its meaning and technologies with the Semantic Web concept. Maybe this could be explained by the fact that the term Linked Data is less used than Semantic Web in an academic context. Finally, it is also interesting to see that a technology like graph databases (Figure 14f), which seems to be quite trending among developers, is not well known in an academic group of computer scientists.

We assume that the knowledge of relational databases and SQL has helped participants in understanding the aspects that SPARQL (and our closely related visual language) has in common with relational algebra. At the same time, the basic data model (RDF) is quite different from the relational model and in part closer to the model of graph databases. So the fact that RDF, SPARQL, and graph databases were poorly known implies a potential challenge in understanding the graph data model and the basics of SPARQL, which is built on graph pattern matchings. The qualitative analysis described in Section 10 confirms this issue.

Regarding the predicted complexity of tasks (Figures 15a, 15d, 15g), participants describe a progressively growing complexity from Task 1 to Task 3, in agreement with the increased complexity of the query required to solve tasks. The complexity perceived after solving the tasks (Figures 15b, 15e, 15h) is distributed instead more like the actual solution times: Task 2 is considered only slightly more difficult than Task 1, while Task 3 is perceived as much more complex. The perceived confidence with the found result (Figures 15c, 15f, 15i) follows a similar trend: participants are quite confident in the solutions found for Tasks 1 and 2, while confidence in the solution for Task 3 is mixed. Some hints for the higher complexity found for Task 3 were already given in Section 8.2: the novelty of an aspect of the query with respect to the queries presented in the tutorial. This difference is further discussed in the qualitative analysis described in Section 10.

The comparison between the complexity expected for a task and the actual complexity of solving it with SparqlBlocks leads to a quite unappealing conclusion: the complexity met by participants using the tool is typically higher than the expected one. This is, in afterthought, a reasonable assessment of complexity by the users: even if the tasks cannot be solved by, say, a single Google query (and Tasks 2 and 3 cannot), a user can quite easily find the results in a relatively short time (compared to measured completion times) by exploring the Web. Tasks 2 and 3 may probably be solved quicker by an expert user of SparqlBlocks (Task 1 is such an easily found answer that it would be hard to beat Google time), but for a new user it is not the case. Needing to learn the vocabulary is also a partial hindrance to the quick design of queries.

Perhaps in contrast with the complexity encountered when solving the tasks, the analysis of the global ratings given by the users to the tool is quite encouraging. The ease of use of the tool is considered from average to good, perhaps recognizing that the usage is not trivial, but after a bit of learning SparqlBlocks permits building complex queries without having to learn a specialized language like SPARQL. Better still, the participants rated the tool as being highly appealing and novel. While the assessment of novelty by users not familiar with Semantic Web standards and tools may and should be downplayed, the fact that the tool was appealing to non experts of the field is for us a very promising result. We also recognize that a probable contribution to this result is the appeal given by the block programming paradigm to people that had not experimented with it before (9 participants out of 11).

10. Qualitative Analysis

A facilitator attended most of the user tests (8 out of 11), taking notes on relevant user behaviour, receiving explicit suggestions and feedback, and following the chain of action-related reasoning of the users. Furthermore, the online questionnaire contained an open-ended field for comments that was used to describe the experience by the three participants that worked on their own. The qualitative data gathered by these means is summarized in this section, focusing on perceived issues, strengths, and suggestions.

10.1. Issues Found

The main issues recognized are described in the following paragraphs.

Difficulties in understanding how blocks may be connected. These appear to be due to two main reasons:

- Failure to understand the behaviour of the block-based UI, especially for the cases of stacking pattern/branches (the availability of the top and bottom connections to stack similar blocks was not evident for many participants) and of replacing shadow blocks (the connector was often not recognized as available when a shadow block was in place);
- Failure to understand the basic structure of patterns-/branches, or the role of variables versus resources (many participants were trying to find analogies with relational algebra, which were often misleading with respect to graph pattern matching and the way variables are bound in SPARQL). A recurring error was using properties in expressions or order by clause, where the intended behaviour would have been realized by putting in the same place the object variable connected to the subject by that property (see for example the query in Figure 17, where the intended query would have been the one in Figure 12).

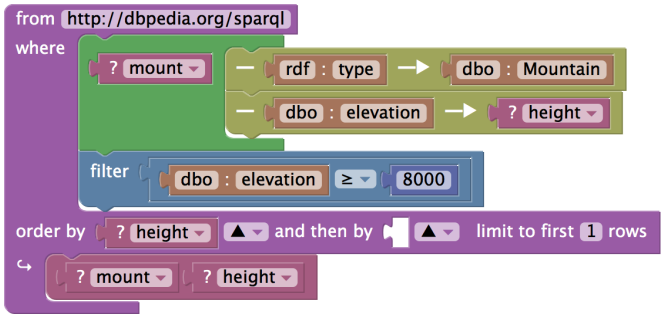


Figure 17. Syntactically correct query that has two constants compared in the filter clause, leading to an empty result set as the filter expression is always false.

Query execution and table results not self-evident. At the beginning of the tutorial, the fact that the query is automatically executed and that the table of results is attached to the same block is not apparent. After some steps of the tutorial, this fact was understood by everyone because it is central to every action that is performed, but missing this fact initially may confound the user and eventually slow down progress in the tutorial.

Operators hidden in drop-down menus were hard to find. Some operator/function blocks contain a drop-down menu for selecting one such item; for example a block is used for all comparison operators (<,≤,=,≠,≥,>) and one for logic operators (*and*,*or*). But in the corresponding category of the toolbox, each block is shown with the default operator/function selected, thus hiding the availability of other operators/functions. That confused many participants, until the logic of the groupings was understood (after which it was not a problem anymore).

Difficulties in using variable blocks. The combination of the nontrivial way in which variables are used in SPARQL (different from both how variables are used in typical programming languages and field names are used in SQL) and some idiosyncrasies in the behaviour of variable blocks (they have a drop-down menu with which the variable used in that place may be changed to be one of the other variables in use or *rename it*, but that changes all the occurrences of the variable) lead to many difficulties in usage of variables. A typical problem was that the participant, trying something or just exploring, randomly changed a default variable to point to another one, leading to the disappearance of the old variable name (because being a variable name generated by default it was not stored), then tried to return to the previous state by using the *rename* command that changed instead the name of that variable in both occurrences. Many times participants expected that dragging a variable from a pattern to a *filter* block or an *order by* field had a copy behaviour rather than a move behaviour.

Complexity of task 3. While the issues in designing queries for Tasks 1 and 2 were mostly related to understanding the user interface and the basic blocks of the language, designing the query for Task 3 proved to be a challenge in terms of actually "thinking about it" for many participants. Several of them initially tried a graph pattern in which the branch with property `dbo:locatedInArea` was used just once and then tried to solve the problem by using a filter that required the corresponding variable to be both `dbpedia:China` and `dbpedia:Nepal`. As the filter is applied to a matching at a time, such query does not give any results (a variable cannot have simultaneously two different values). To solve this task, the query needs to have two branches with property `dbo:locatedInArea` that may connect to two different variables that can then be constrained to be equal respectively to `dbpedia:China` and `dbpedia:Nepal` in a filter (see Figure 18). Even better, these two `dbo:locatedInArea` branches may directly connect to `dbpedia:China` and `dbpedia:Nepal`, respectively (as shown in Figure 13). All the participants who solved Task 3 basically achieved it by using one of these two queries (with some variations), but guessing a working query was challenging for most of them. This not an issue related to environment, but rather a recognition of the added step of reasoning (and comprehension of the system) needed to solve this task.

Unhelpful endpoint errors. The SPARQL protocol, used to communicate with SPARQL endpoints, does not give much support to the sensible communication of server errors, so very different errors (like a syntax error or a time-out) can be distinguished only through a non-standardized textual description returned by the server. As an example, in Table 2 we show the responses of different SPARQL endpoints for the same

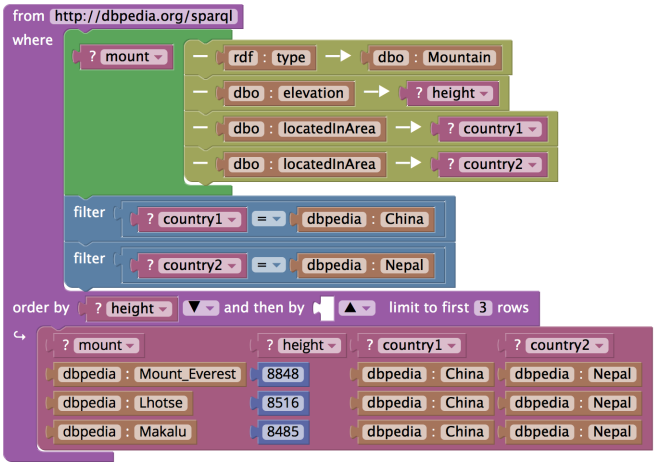


Figure 18. Alternative query to solve task 3.

type of error, a timeout during the execution of the query. SparqlBlocks shows an error in place of the table of results labelled with the truncated (not to break the user interface) error message from the server. This is often not very helpful to the user, like in the example shown in Figure 19 of a query on the WikiData endpoint.

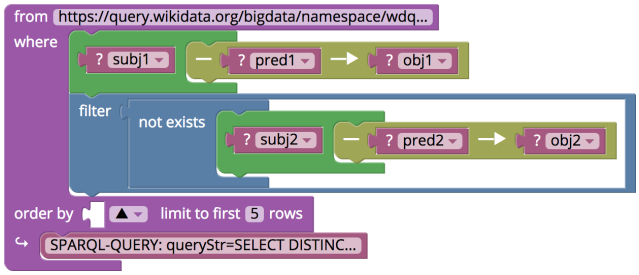


Figure 19. Query block with an error shown.

Acceptance of nonmeaningful queries. One of the main advantages of the block-based interface is that syntax errors are avoided. Nevertheless, the participants sometimes built queries that either contained redundant parts or were bound to return an empty result set, for one of two reasons:

- the syntax was accepted but it actually made no sense; e.g. a variable not used in the query graph patterns was used in a filter or in a block connected through an *order by* connector (see for example the query in Figure 20);
- a query made sense in general terms but it did not respect the dataset semantics; e.g. a property was used as subject or object in a graph pattern[22] (see for example the query in Figure 21).

[22]That could make sense in special cases, like to query ontology meta-data, but, in practice, it is often just an error.

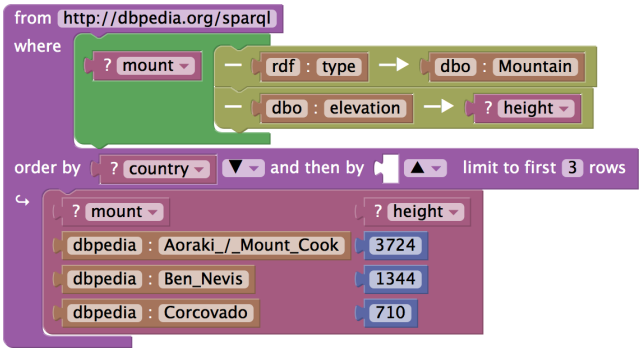


Figure 20. Syntactically correct query having a variable appearing only in the order by clause, which has no effect on the ordering.

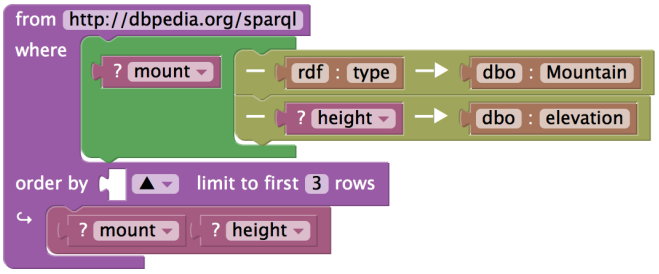


Figure 21. Syntactically correct query in which a property appears in the role of the object, leading to an empty result set.

10.2. Perceived Strengths

Many users saw the following as strengths of the tool. (Phrases shown surrounded by quotation marks are actual quotes from users[23].)

"Once you get used to it, it is very intuitive." After the tutorial and the execution of tasks that were quite intense with a lot of concepts to learn, the participants generally felt quite empowered and they felt that, at that point, they were able to solve similar problems more easily.

"Once you see the connector highlighted, you see where you can put the block." As previously described, participants sometimes felt frustrated for not understanding immediately where a block could be connected. Perhaps for that reason, the visual feedback given when a block is close to a compatible connector (the connector is highlighted) was much appreciated.

"I enjoyed it very much!"/"I loved using this tool!" Many participants expressed enjoyment using the tool, in accordance with the high rate given to Use Appeal in the questionnaire.

"It may be useful for education." Some participants highlighted SparqlBlocks potential as an educational tool.

"Search blocks are really useful!" A participant with previous experience with Semantic Web technologies indicated the black-box query blocks as especially useful, possibly for

[23]Quotes are used to exemplify concepts expressed by multiple users (of whom at least one of them used the given wording).

Table 2. Timeout error response messages from different SPARQL endpoints.

Endpoint	Server	Status Code	Text Message
DBpedia	OL Virtuoso	500 SPARQL Request Failed	Virtuoso S1T00 Error SR171: Transaction timed out SPARQL query: *(SPARQL)*
WikiData	Sesame	500	SPARQL-QUERY: queryStr=*(SPARQL)* *(full Java stack trace of a* `QueryTimeoutException`*)*
Linked Open Aalto Data Service	Apache Fuseki	503 Query timed out	Error 503: Query timed out Fuseki - version 2.5.0 (Build date: ...)

being familiar with the problem of exploring a dataset without prior knowledge of the used vocabularies.

10.3. Suggestions

We received the following suggestions.

To add contextual help and examples associated with the block types and accessible from the workspace.

To add retry button (or auto-retry) for when the query fails for (possibly temporary) connection problems. Now the only (admittedly suboptimal) solution is to disconnect and reconnect the query pattern stack.

Joins among queries. While it is possible to combine any number of patterns in a query to a single endpoint, it is not currently possible to design queries that access multiple endpoints. It would not be difficult to introduce this extension in the user interface, considering that SPARQL supports that feature through an optional extension called SPARQL Federation, but we should connect to a server supporting it. Public SPARQL endpoints like DBpedia do not usually support SPARQL Federation.

11. Conclusions and Future Work

We have proposed SparqlBlocks as a visual language and an interactive environment embodying a new paradigm for querying Linked Data. SparqlBlocks is based on a novel take on block programming: using blocks not only to program but also to show results, which can be incorporated in incremental design of queries. A group of users with strong computer science background but small to no experience in querying Linked Data were able to successfully design non trivial queries with the tool.

At the same time, the evaluation and analysis of the use of SparqlBlocks opens up new questions and stimulates further experimentation in the field of Linked Data access and block programming environments.

11.1. SparqlBlocks

The users had some issues related to some aspects of the user interface, especially concerning the **representation of graph patterns** and the **usage of variables**. While participants managed anyway to effectively use the tool in a relatively short amount of time, tackling these issues is probably critical to lower the bar for expertise and effort required to start using SparqlBlocks. A central point is that maintaining the full expressiveness of SPARQL has a cost in terms of having a complex visual language (the user can easily mix things in legal but meaningless ways). In many cases limiting the expressiveness may help the novice user.

11.2. SPARQL

Some issues or requests push the limits of current SPARQL infrastructure.

Better feedback on server errors would require going beyond the current SPARQL protocol, while in the medium term it could be tackled by designing a layer that may interpret the output of the most used kinds of SPARQL endpoints and give a semantically well-defined answer.

Proposing more complex queries, for example joining patterns from multiple endpoints, requires not just adding the missing pieces of the full SPARQL language, but also having on the server side a system capable of executing those complex distributed queries in an efficient way —which is not a fully solved problem, neither in practice nor in theory.

11.3. Block Programming

Some issues should be analysed in the context of block programming environments.

How to best represent optional component with default values. This concerns the trade-off between (1) offering blocks with defaults (for example through shadow blocks) versus (2) requiring filled connections versus (3) allowing empty connections (with implicit defaults).

Management and visual representation of variables. In several block programming environments, variables are all global, thus sacrificing the principle of information hiding in order to gain the possibility of visually interacting with variable blocks without having to manage scopes. In some cases, like MIT App Inventor 2, the management of variables has been designed to permit lexically scoped variables [31]. Locally scoped variables are introduced as parameters of functions or through a specific block that initializes a set of local variables and encloses the instructions in which they may be used. In SPARQL, variables are locally scoped to queries, but in the language there is no explicit declaration of them. Furthermore, while variables in basic graph patterns may be novel or refer to existing variables, variables in expressions (for *filter* and *order by* fields) should have already been introduced in some basic graph pattern of the same query. We

chose to keep the system simple by managing the variables as if they were all global and leaving it to the user to deal with them in the correct way. There is possibly room for improvements by devising a visual representation of the scope system of SPARQL.

Shapes and allowed connections. The recognition of available connections is paramount to the effective usage of a block programming environment, but a complex language may have many types (hence potentially many connection types) and connectors that accept multiple types. Which is the "right" trade-off in the number of different shapes? How can the UI be augmented to further support fine-grained distinctions among types in a way that the user would still be able to see the potential connections at a glance? Blockly is rather conservative in having just two types of connectors. The OpenBlocks system [32], upon which the original MIT App Inventor and StarLogo TNG were based, had support for more connector shapes and supported polymorphism. Some prototype systems went to the length of supporting arbitrary complex derivative types (like tuples or functions) by graphically composing basic connector shapes through a set of rules [33, 34]. We chose not to follow this line, as we were interested in first analysing the potential of the approach for grasping the essential aspects of SPARQL. Further experimentation will be needed to establish the trade-off between the greater expressivity given by an increase in the available possibilities for manipulation and the consequent increase of cognitive load. It could also be interesting to explore how a hierarchical system of types may be represented through the use of different connection shapes.

Richness and organization of the toolbox. The organization of the toolbox is paramount to user's comprehension of language affordances. It is not trivial to find a compromise on the number of offered blocks. For example, blocks representing multiple operators (e.g., the logic operator block, used for *and* and *or*) are shown just with default one (in this case *and*) to avoid overloading the toolbox. To use another operator, the user must use a drop-down menu and change it. This was found to be non evident to several users. So, at least for some cases, it may be worth to show already all or most of the options as separate blocks in the toolbox, as, for example, MIT App Inventor does.

Extension of the SparqlBlocks' paradigm to similar applications. The results of the evaluation of the tool are so far promising, so it could be interesting to extend the paradigm to similar languages. The proposed incremental approach and UI could potentially be applied to multiple query languages and data models.

12. Acknowledgments

We thank all the reviewers for the valuable contributions they made to the development of this paper. We are specially grateful to Franklyn Turbak for his help and tireless commitment to improve this paper through countless suggestions and amazing editing work. We also thank all the volunteers who participated in the user study for their time.

References

[1] T. Berners-Lee, "Linked data," 2006. [Online]. http://www.w3.org/DesignIssues/LinkedData.html

[2] C. Bizer, T. Heath, and T. Berners-Lee, "Linked Data – The Story So Far," *International Journal on Semantic Web and Information Systems*, vol. 5, no. 3, pp. 1–22, 2009.

[3] R. Cyganiak, D. Wood, and M. Lanthaler, "RDF 1.1 Concepts and Abstract Syntax," W3C REC 25, February 2014. [Online]. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225

[4] S. Harris *et al.*, "SPARQL 1.1 Query Language," W3C REC 21, March 2013. [Online]. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/

[5] S. Ferré, "Sparklis: a SPARQL Endpoint Explorer for Expressive Question Answering," in *Proceedings of the 2014 International Conference on Posters & Demonstrations Track (ISWC-PD'14)*, vol. 1272. CEUR-WS, 2014.

[6] A. Russell, P. R. Smart, D. Braines, and N. R. Shadbolt, "NITELIGHT: A Graphical Tool for Semantic Query Construction," in *Proceedings of the 5th International Workshop on Semantic Web User Interaction (SWUI '08)*, vol. 543. CEUR-WS, 2008.

[7] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: Programming for All," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[8] D. Wolber, H. Abelson, E. Spertus, and L. Looney, *App Inventor 2: Create Your Own Android Apps*. O'Reilly Media, Inc., 2014.

[9] P. Bottoni and M. Ceriani, "Linked data queries as jigsaw puzzles: a visual interface for SPARQL based on Blockly library," in *Proceedings of the 11th Biannual Conference on Italian SIGCHI Chapter (CHItaly 2015)*. ACM, 2015, pp. 86–89.

[10] P. Bottoni and M. Ceriani, "SPARQL Playground: a block programming tool to experiment with SPARQL," in *Proceedings of the ISWC 2015 workshop on Visualizations and User Interfaces for Ontologies and Linked Data (VOILA 2015)*, 2015, p. 103.

[11] D. Peterson, S. S. Gao, A. Malhotra, C. M. Sperberg-McQueen, and H. S. Thompson, "W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes," W3C REC 5, April 2012. [Online]. http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405

[12] M. Duerst and M. Suignard, "Internationalized Resource Identifiers (IRIs)," RFC 3987 (Proposed Standard), Internet Engineering Task Force, Jan. 2005. [Online]. http://www.ietf.org/rfc/rfc3987.txt

[13] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," RFC 3986 (INTERNET STANDARD), Internet Engineering Task Force, Jan. 2005, updated by RFC 6874. [Online]. http://www.ietf.org/rfc/rfc3986.txt

[14] L. Feigenbaum, G. T. Williams, K. G. Clark, and E. Torres, "SPARQL 1.1 Protocol," W3C REC 21, March 2013. [Online]. http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/

[15] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer, "DBpedia – A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia," *Semantic Web Journal*, vol. 5, pp. 1–29, 2014.

[16] N. Fraser, "Blockly — Google Developers," accessed: 2017-05-10. [Online]. https://developers.google.com/blockly/

[17] L. Rietveld and R. Hoekstra, "YASGUI: Not just another SPARQL client," in *Proceedings of the Extended Semantic Web Conference (ESWC 2013) Satellite Events*. Springer, 2013, pp. 78–86.

[18] J. Borsje and H. Embregts, "Graphical query composition and natural language processing in an RDF visualization interface," B.S. Thesis, Erasmus School of Economics and Business Economics, Erasmus University, Rotterdam, 2006.

[19] F. Haag, S. Lohmann, S. Siek, and T. Ertl, "QueryVOWL: Visual composition of SPARQL queries," in *Proceedings of the Extended Semantic Web Conference (ESWC 2015) Satellite Events*. Springer, 2015.

[20] F. Haag, S. Lohmann, S. Bold, and T. Ertl, "Visual SPARQL querying based on extended filter/flow graphs," in *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces (AVI '14)*. ACM, 2014, pp. 305–312.

[21] J. Groppe, S. Groppe, and A. Schleifer, "Visual query system for analyzing social semantic web," in *Proceedings of the 20th International Conference Companion on World Wide Web (WWW '11)*. ACM, 2011, pp. 217–220.

[22] M. M. Zloof, "Query-by-Example: a data base language," *IBM Systems Journal*, vol. 16, no. 4, pp. 324–343, 1977.

[23] H. N. M. Quoc, M. Serrano, D. Le-Phuoc, and M. Hauswirth, "Super stream collider-linked stream mashups for everyone," in *Proceedings of the Semantic Web Challenge co-located with ISWC2012*, 2012.

[24] M. Resnick, "StarLogo: An environment for decentralized modeling and decentralized thinking," in *Conference Companion on Human Factors in Computing Systems (CHI '96)*. ACM, 1996, pp. 11–12.

[25] J. Gorman, S. Gsell, and C. Mayfield, "Learning relational algebra by snapping blocks," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, 2014, pp. 73–78.

[26] Y. N. Silva and J. Chon, "DBSnap: Learning database queries by snapping blocks," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, 2015, pp. 179–184.

[27] A. Jain, J. Adebayo, E. de Leon, W. Li, L. Kagal, P. Meier, and C. Castillo, "Mobile Application Development for Crisis Data," *Procedia Engineering*, vol. 107, pp. 255–262, 2015.

[28] S. Dasgupta and B. M. Hill, "Scratch Community Blocks: Supporting children as data scientists," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, 2017, pp. 3620–3631.

[29] T. R. G. Green, "Cognitive dimensions of notations," *People and Computers V*, pp. 443–460, 1989.

[30] T. R. G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.

[31] F. Turbak, D. Wolber, and P. Medlock-Walton, "The design of naming features in App Inventor 2," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2014, pp. 129–132.

[32] R. V. Roque, "OpenBlocks: an extendable framework for graphical block programming systems," Master's thesis, Massachusetts Institute of Technology, 2007.

[33] M. Vasek, "Representing expressive types in blocks programming languages," Undergraduate thesis, Wellesley College, 2012.

[34] S. Lerner, S. R. Foster, and W. G. Griswold, "Polymorphic blocks: Formalism-inspired UI for structured connectors," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, 2015, pp. 3063–3072.

Lessons in Combining Block-based and Textual Programming

Michael Homer and James Noble
Victoria University of Wellington
New Zealand
{mwh,kjx}@ecs.vuw.ac.nz

Abstract *Tiled Grace is a block-based programming system backed by a conventional textual language that allows switching back and forth between block-based and textual editing of the same code at any time. We discuss the design choices of Tiled Grace in light of existing research and a user experiment conducted with it. We also examine the sorts of task preferred in each mode by users who had the choice of editing either as blocks or as text, and find both positive and cautionary notes for block-based programming in the results.*

1. Introduction

With Tiled Grace, we aimed to produce a block-based programming system that was fully integrated with a conventional textual language aimed at education, Grace [1–3], to allow learners to make the transition from blocks to text gradually over time. At any time, a programmer can with the click of a button switch from editing their code as blocks to editing it as text, and back again, as often as desired and for as long as required. Tiled Grace matches its block syntax exactly with the syntax of the textual language to reinforce knowledge of both, and uses animations to show the correspondence between the two representations while transitioning. Like the textual Grace language, Tiled Grace aims to be a guiding step for novices who will move on to other languages or paradigms.

In this paper we explore our specific design choices made while building Tiled Grace and elaborate on their motivation in relation to existing work in both visual languages and educational psychology. We reflect on those choices in retrospect in light of both subsequent literature and a usability experiment we conducted with the tool. This experiment aimed to determine:

- whether this ability to switch views would actually be used, or if users would merely use one or the other;
- whether the novel animated transition connecting the two views was appreciated, or found confusing or unhelpful;
- whether the error reporting system we had created for the tiled view was helpful, as it was entirely experimental;
- how engaged users were with the tool, as a system that users do not enjoy will not be used;
- and any unanticipated difficulties or usage.

DOI reference number: 10.18293/VLSS2017-007

Our experiment with Tiled Grace also offers a unique opportunity for analysis. For the first time, programmers had the opportunity to edit the same program as both blocks and as text, and particularly to edit *parts* of the program in each mode. We perform a new analysis of the dataset in this paper, examining the choices and revealed preferences of the users in the experiment, with a further goal of finding:

- which tasks users preferred to perform as text and which as tiles;
- and how these patterns vary by the experience level of the user.

From all of this we attempt to draw lessons for the future of block-based programming. While we see a number of positive signs for these languages and editors, we also find some cautionary notes where enthusiasm may not match reality.

In the next section we briefly introduce Tiled Grace at a high level. After that, we outline the more relevant similar systems, and then discuss Tiled Grace's design choices for usability and learnability in relation to others, and where we made trade-offs to support our goal of integrating text and blocks. We then summarise a past usability experiment with the tool, and go on to present a novel analysis of the actual use of each modality, before relating and comparing our results and experience with what has been reported by others. Finally, we attempt to draw some lessons for block languages from both our experience in building a hybrid system and our experimental results.

2. Tiled Grace

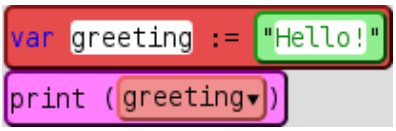
We will briefly introduce the Grace language generally: Grace is a textual, object-oriented, block-structured, curly-brace language aimed at novice programmers, primarily in tertiary study [1]. It aims to support new programmers in the first year or two so that they can develop the understanding required to learn new languages for their later careers, and includes a number of design choices intended to assist that. In this work, we are building on Grace, and on an implementation of Grace, leveraging the existing pedagogical design and implementation work already carried out on the textual language.

Tiled Grace [4, 5] presents an editing environment for Grace programs based on drag-and-drop *tiles*. The basic structure

is common across drag-and-drop block-based programming systems. The tiles for a string and variable, for example, appear as:

Some tiles have *hole*s in them, where another tile may be placed. For example, a variable assignment tile has two holes: one for the variable to be assigned to, and one for the value to be assigned:

The holes are the empty grey rounded-rectangular areas. The user can drag a tile inside a hole to build up their program. Tiles can be connected together in sequence as well. To create a variable and print its value, a var tile and a print tile can be joined together.



A complete program and its output in progress is shown in the Tiled Grace interface in Figure 1. The interface is divided into three main areas: a large workspace area on the left, a toolbox of available tiles, and text and graphical output areas on the right.

Tiles may be dropped anywhere in the workspace pane, and the user can construct different sub-programs in different parts of the area. Different categories of tile can be selected from a pop-up menu that appears when using the toolbox. At the bottom of Figure 1 the dialect selector, run button, and other interface controls are displayed.

The user can switch to a textual editor at any time. The transition from tiled to textual view is shown through a smooth animation where each tile and block of code has a continuous visual identity throughout the transition.

First the tiles fade out to blocks of the corresponding textual code, then the blocks glide into place in a linear textual program, and finally the display switches to editable text. The entire transition takes just under two seconds. When the user chooses to switch back to tiles, the same behaviour occurs in reverse.

Figure 2 shows this transition in progress: while editing the same program as shown in Figure 1, the user has switched to a textual view. First the tiles fade out to blocks of the corresponding syntax-highlighted textual code, while remaining in the same physical location (frame (b)). The code blocks then glide into place (frame (c)), finishing in a linear textual ordering. Finally, the tiles become fully editable ordinary text, as shown in frame (d). In this way, the relationship between tiles and the corresponding part of the textual program is clearly visible.

Each separate group of connected tiles is regarded as an independent part of the program. The ordering between them in the textual display is arbitrary, but consistent across the lifetime of the program. The displayed text is editable if the user wishes: they may change the source code, including adding and removing whole lines or blocks, and then transition back to the tiled view. When exported, a meaningful comment is appended to the end of the program stating the coordinates of each independent "chunk" in pixels, while the chunks are separated from each other by blank lines; within the system the location information is stored in memory during a text-editing session. If a new chunk of code is added, it is assigned a default location upon switching back to tiles.

Tiled Grace innately supports *dialects*, a language variation feature of the textual Grace language [6]. Dialects can provide new features to the language and new tiles in the toolbox, and impose additional restrictions on what can be written. Each module [7] of the program can use a different dialect. Different dialects can provide drastically different sublanguages, with their own tiles and rules, within the same overall syntax and semantics. Because all Grace control structures are defined as methods to begin with, a dialect can introduce its own control structures at will and they will fit in with the rest of the language.

Tiled Grace runs in a commodity web browser, including both the editor and the backing compiler. It can be accessed at http://ecs.vuw.ac.nz/~mwh/minigrace/tiled/, and works at least in recent versions of Firefox, Chrome, and Internet Explorer/Edge. Tiled code is converted behind the scenes to textual Grace code, which is then compiled into JavaScript for execution. The compiler provides a syntax-tree export that is used to transform textual code back into tiles, or to import new textual code.

We will discuss other features of Tiled Grace in relation to the motivations that inspired them in the next section.

3. Existing Block Programming Systems

Several block-based programming languages and systems are in current use, the most well-known of which is probably Scratch [8]. We will briefly introduce the systems most relevant to Tiled Grace, focusing on the aspects that relate to this work. The design dimensions we consider interesting are:

- the role of a **textual modality**: is there none available, export to another language, export to and import from another language, switchable views, or simultaneous display?
- the sort of **block positioning** that it allows: freeform layout or a fixed structure.
- **when errors** are reported: is it when they are introduced, when starting to run the program, at runtime, or never?
- **how errors** are reported: in-place, in a list, one at a time in a fixed place, or with a marker. (In both of these dimensions we are interested in syntax, structure, and type errors, rather than logical errors — the kinds of error that a reasonably conventional textual language might report statically).

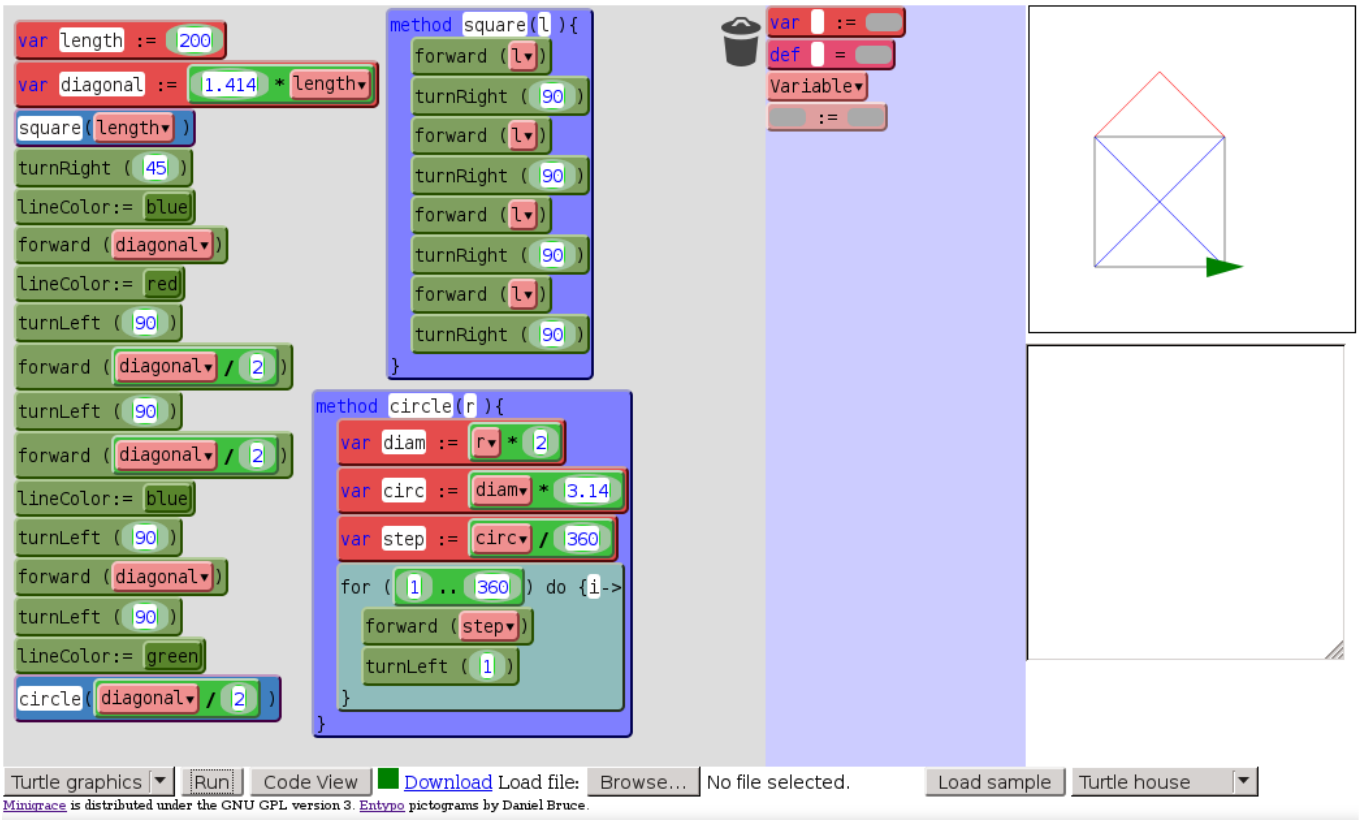


Figure 1. Tiled Grace editing a small program in the "turtle graphics" dialect, currently executing

- are different **types distinguished** by shape, colour, repulsion from incompatible locations, not at all, or some other means?
- how are **dependencies** between different parts of the code (for example, variables and their references) maintained or indicated: not at all, with an overlay, with automated scoped renaming, or something else?

Table 1 summarises each system we consider against each of these dimensions. Tiled Grace has, respectively, a switchable text view, freeform layout, errors shown both when introduced and when run, errors displayed in place, repulsion of incompatible types, and an overlay showing uses and definitions of variables and methods along with automated renaming.

3.1. Scratch

The visual side of Tiled Grace is most similar to Scratch [8], a wholly visual drag-and-drop programming environment with jigsaw puzzle–style pieces, aimed at novices and children. Scratch is purely visual: there is no textual representation of Scratch code at all (although its blocks are all dependent on textual labels). A key boon of Scratch is its immediate graphical microworld. A student can instantly see the effects of running a piece of code, live in front of them. Code can be modified during execution with instant feedback.

Scratch has been very successful in driving engagement, particularly with children. New users who might never have considered programming a computer take to it quickly and begin exploratory programming with little prompting. We observed this engagement ourselves while working with Scratch in an outreach programme to a local school, which led to the conception of Tiled Grace as a way to gain this engagement within a more complete language. Scratch has also proven useful for a variety of other purposes, including driving social interaction between children, promoting storytelling, and teaching music; we did not focus on these areas for Tiled Grace and will not address them further in this paper.

Scratch has freeform positioning of blocks anywhere in the workspace, and uses different block shapes to distinguish types of value. Scratch does not regard any program as erroneous, and will attempt to execute any program, skipping over missing or invalid parts. There is no innate textual form of Scratch, though a debugging export of an entire program is available by a hidden option.

3.2. Squeak Etoys

Etoys is a tile-based programming system built on the Squeak Smalltalk system [9]. Etoys focuses on exploratory learning in general. Textual code equivalent, but not identical, to the tiled code can be exported and executed. Tiled code is always valid, with type errors prevented from being constructed, while erroneous textual code cannot be activated. Tiles corresponding to abilities of each item in the world are

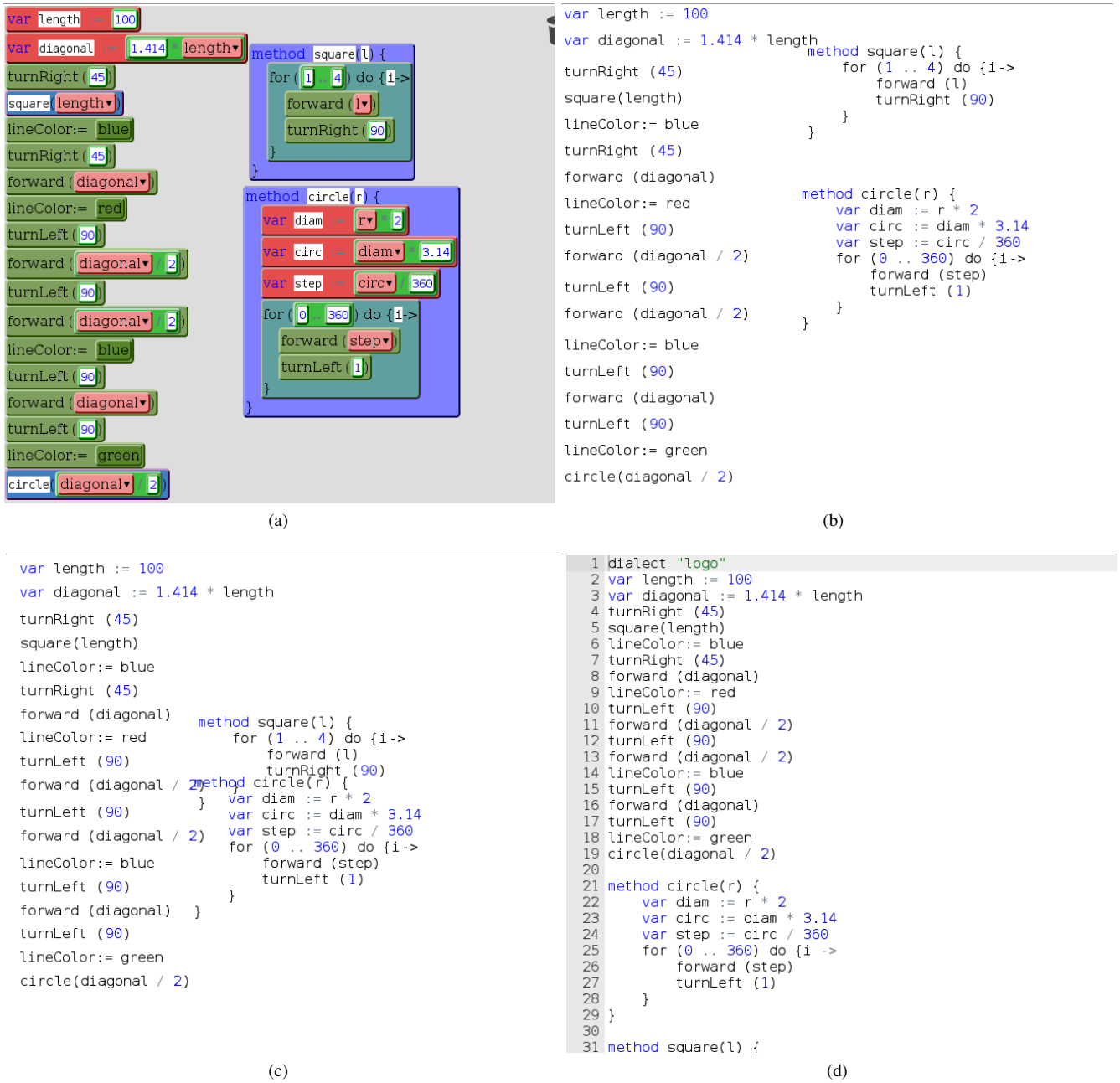


Figure 2. Frames of the animated transition from tiled to textual view. (a) Tiled. (b) Fade backgrounds and highlight syntax. (c) Glide tiles towards their positions in the text. (d) Switch to an actual text editor in-place. Transitioning from textual to tiled view shows the same intermediate states in reverse. The transition from tiles to code, and the movement of code, is smoothly animated.

accessible through a menu on that item, rather than a general broad menu. An interesting aspect of Etoys is that code tiles exist within the same world as user objects, and tiles can be created representing the physical display of blocks of code. These tiles can be used like any other to manipulate the display of the code.

3.3. Alice

Alice [10] is similarly microworld-driven, but aimed at a slightly older audience. Alice has objects in the three-dimensional microworld that are also objects in the object-orientation sense, with a primarily event-driven programming model with common structured-programming features as well. Alice makes heavier use of menus than Scratch, but also has a higher degree of enforced structure. These menus provide on-demand exposure of relevant possibilities and encourage a different style of experimental programming than Scratch's toolbox.

Alice has structured positioning of blocks, with (subtle) shape indicators for some types. Alice supports exporting to

Table 1. Many existing block programming systems laid out according to their design elements relating to each design dimension.

Language	Text	Positioning	When errors	Where errors	Types	Dependencies
Scratch	None	Freeform	Never	n/a	Shape	Renaming (some)
Etoys	Export	Freeform	Never	n/a	Repulsion	Renaming
Alice	Export	Structured	Intro., start	One, list	Shape	Renaming
Blockly	Export	Freeform	Start	Varies	Repulsion	None
App Inventor	None	Freeform	Intro., start	Marker	Repulsion	Renaming
Pencil Code	Switchable	Structured	Start	One	No	None
BlockEditor	Export/Import	Structured	Start	One, list	Shape (some)	None
GP	Export/Import	Freeform	Never	n/a	Shape	Renaming
Calico Jigsaw	Export	Freeform	Runtime	One	No	None
TouchDevelop	None	Structured	Intro.	One	Repulsion	Renaming
Greenfoot	Read-only	Structured	Intro.	Marker	n/a	None
Tiled Grace	Switchable	Freeform	Intro., start	In-place	Repulsion	Overlay, renaming

Java and reports many errors upon introduction, and some when starting the program.

3.4. Blockly

Blockly [11] is very similar in ethos to Scratch, with freeform positioning. Blockly runs in a web browser and incorporates language variants (what we call *dialects*), but in mimicking Scratch also has no editable textual format. Blockly's goal is to support developers embedding a visual language into other systems, both educational and otherwise.

Blockly supports exporting code to a number of languages, but these exports are not bijective. There is no explicit indication of which parts of the visual representation correspond to which parts of the textual representation. Because Blockly is designed for embedding, a range of different behaviours can be provided by the embedder, but repulsion from invalid locations is built in.

3.5. App Inventor

App Inventor [12, 13] is one of a number of Blockly clients, aimed at teaching. App Inventor was extended with a textual language, TAIL, semantically equivalent to its block language [14, 15]. An aspect of the TAIL integration that most systems do not match is the ability to embed a portion of textual code within the block view, as a block containing the text. This feature of TAIL is one that would be particularly useful in systems when there are parts of the textual language that the blocks cannot express, but came after our work so we did not include it.

As well as TAIL, a Python export for App Inventor has been proposed [16]. Neither is currently in the production release. App Inventor marks erroneous blocks with a persistent indicator when the error is detected, and the user can inspect the block individually for an error report. It has no tracking of interdependencies beyond renaming variables.

3.6. Droplet and Pencil Code

Droplet [17, 18] and the closely-related Pencil Code [19] slightly postdate the genesis of Tiled Grace, but also attempt to bridge blocks and text. Pencil Code concentrates on straight-line programs in a Logo-like turtle graphics system and simple audio/drawing programs, and supports editing large subsets of both CoffeeScript and JavaScript as blocks and text, with both block and text editing for each language.

Droplet introduced an animated transition that parallels Tiled Grace's and is now part of mainline Pencil Code with many users. Droplet is a general library supporting any language with an appropriate adapter, but the main use is in Pencil Code with CoffeeScript. Text is treated as the primary representation in Droplet, and it retains complete or nearly-complete information from the source, including comments and layout (which Tiled Grace does not only due to technical limitations of the underlying compiler). Were we creating Tiled Grace now, we would likely build on the Droplet library instead of a bespoke system.

Pencil Code reports some errors upon trying to run the program by way of a popup. Types are not indicated in any way. Blocks are positioned in a structured fashion, and no dependencies are tracked. Switching between block and textual view is possible at any time, except when textual syntax errors exist.

3.7. BlockEditor

Matsuzawa *et al* [20] built BlockEditor, an editor for a visual language called Block that can save the program to textual Java. The Java code can be edited and automatically reimported into BlockEditor. In this way a learner can move between the two languages at will, continuing with the same code. The Block visual language is not exactly the same as the Java textual language, but parallels the structure closely enough for a bijection to exist for the programs in an introductory course. An experiment over a first programming course for non-majors found that users did use both modes, with the rate of Block use trending downwards and Java upwards as a course progressed, and that higher usage of the visual mode corresponded with lower self-efficacy.

BlockEditor allows exporting to Java, editing, and re-importing the code. The block and textual code looks dis-

similar, but is semantically equivalent. Shape is used to mark some types of sockets.

3.8. GP

GP is a general-purpose blocks programming language intending to be useful for casual programmers other than children [21]. GP is principally block-based (building on Snap! [22]), but experimentally has had an editable text mode (exposing the underlying LISP-like language), highly condensed blocks appearing as text, and text-based insertion of blocks. The text-like blocks were found to be less jarring (though less powerful) than exposing export and import of the underlying data structure had been. Permitting not only animation but the ability to select intermediate points exposes the text-block transition to a further degree than other systems, including Tiled Grace.

3.9. Calico Jigsaw

Calico [23] is a multi-language IDE for introductory programming, which includes a visual language called Jigsaw. Jigsaw uses puzzle pieces and drag-and-drop, and the Calico system enables exporting the program to other textual languages, primarily Python. The Jigsaw syntax is distinct from any textual language and export is to a text file. Jigsaw allows a degree of freeform positioning of blocks, and reports errors primarily at run time, with the triggering block marked.

3.10. TouchDevelop

TouchDevelop [24] integrates an essentially textual language with an IDE aimed at touch-screen usage, rendering the program as large blocks. The IDE avoids most use of textual input by having the user manipulate the syntax tree itself: the user touches where they want to change and the IDE presents them with a list of options they can put there. When the programmer adds a new element the system will prompt them to fill in any required arguments, like the condition of a loop. While the syntax is reasonably conventional, there is no direct textual form of TouchDevelop code, and some aspects, such as comments, are shown only by typographical features. Editing always corresponds essentially to textual insertion or deletion. TouchDevelop has fully structured positioning and enforces that the program is well-formed whenever possible.

3.11. Greenfoot and Stride

Greenfoot is an introductory programming system with a two-dimensional microworld, which has recently been extended with a "frame-based" editor [25, 26]. Like a block system, Greenfoot's frame-based editor presents hierarchically-related elements of the code as nested indivisible oblongs with slots for subsidiary elements, but like a textual language interaction and input is principally with the keyboard. Its Stride language reuses the concepts of Java and uses textual labels and structure closely matching Java syntax.

The programmer manipulates the syntax tree at the level of individual nodes using single-letter keyboard shortcuts, but some slots use free text entry even for structured elements (such as method references or loop conditions), so Stride is a hybrid structured and unstructured editor. These unstructured fields are the only place that syntactic errors can be introduced, other than empty mandatory fields. An individual module is always a well-formed class down to some subsidiary point.

A Java view of the module can be shown at any time, including while the program is in an erroneous state. This view is not editable, but inserts the necessary braces, comment markers, and other syntactic structure, while removing additional labels present in the frame view, with an animated transition preserving the identity of the elements common to both views.

3.12. Other systems

A number of other systems, or experimental systems, have incorporated some level of textual code alongside blocks. These include a simultaneous-display version of Snap! [27] showing JavaScript code (not part of mainline Snap! development), and systems for defining extension blocks using some host textual language [22, 28, 29]. We discuss the Snap! extension briefly below, but consider merely-extensible systems out of scope for this paper.

4. Designing Tiled Grace

Why build Tiled Grace when Scratch, Alice, and similar systems already exist? Our design goal for Tiled Grace was to provide the engagement and lessened syntactic burden of these existing systems, while introducing the concepts of a textual language at the same time so that the user could transition into it at their own pace, in accordance with educational psychology principles. We built on top of an existing conventional textual language aimed at education in order to leverage existing education design work, rather than reinventing it, building an interface for editing that textual code rather than a new language. In this section we break down some motivating aspects of our design, particularly in relation to other languages and approaches we built on or steered away from.

4.1. Migration

A key goal in Tiled Grace was to enable its own obsolescence for each user in their own time, where they could move on to the textual paradigm when ready and with the support to do so successfully. Building on a general-purpose language ensured that there was no functional limitation in the tiled view (as contrasted with microworld-focused languages), but did not alone ensure that it would be feasible to move on. Ultimately, Tiled Grace aimed to ease beginning with Grace, to match the Grace language's goal of easing beginning with programming, in both cases as an initial step only.

A well-reported problem with moving on from visual to textual languages [30], and moving between languages early in learning in general, is that learners find it difficult to connect analogous concepts in one language to the other. In particular,

it is known from both educational psychology in general, and computer science education specifically, that transitioning between languages early in learning is unhelpful [31], or indeed any attempted *transfer of learning* at an early stage without very careful structuring [32, 33].

For learners to achieve transfer they must be taught the concepts in a fashion that facilitates transfer [32]. Without such teaching the knowledge tends to be *inert*: it can be applied within its original context, but learners will not generalise from that context to apply their knowledge elsewhere. Perkins and Martin found that students learning to program would learn language constructs inertly, and so had difficulty applying their knowledge to the act of programming, notwithstanding that the distance of transfer is minimal in this case [34], while Dyck and Mayer found that without transfer-focused teaching learners of BASIC would master the syntax of the language, but struggle more with semantics than those taught with transfer [35]. An assumption in much teaching is that transfer to similar domains will occur automatically, but research has not borne this assumption out in practice [32], instead finding that instruction must be tailored to assist transfer; in the literature, this tailoring to target explicit transfer is called *bridging* [35–37]

These ideas and experiences were a strong influence on our design of Tiled Grace. In particular, the animated transition between visual and textual representation aims to assist bridging by demonstrating the exact parallel between the two sides. Similarly, we made the block structure match the syntax of the textual language, and even display the relevant syntax in-place on each tile. In this way the user was always seeing the textual syntax, even while editing blocks, and would gain some familiarity with what to expect in text. Using text as the primary representation also ensured a convenient interchange format for both whole and partial programs.

Restricting ourselves to an exact match with the textual syntax limited what we could do in the blocks. Unlike other visual block-based languages we could not use additional layout or components within a single tile to make the block language simpler (for example, using multiple successive holes without intervening syntax, adding extra labels, or physically offsetting or aligning some fields to distinguish them with no other syntax), because that would break the direct correspondence with the textual syntax we did not control. This is one area where our goal of integrating both worlds has made the system weaker in respect of one approach or the other than a "pure" block or text language.

Given all of the above, there is a fair question in the air — why switch to textual languages at all? Aside from the dearth of professional block languages for those students who would like a job in future, a key issue in existing block languages is that using them is exhausting, having high "viscosity" [13, 38] — the difficulty of making a local change. Even reading and understanding a complex program with many nested blocks can be difficult [14]. While designing Tiled Grace we knew from our own experience that as we had come to know the system better we had found the drag-and-drop interface of Scratch increasingly tiresome to use. Moving to the toolbox, dragging a tile out, switching to a different pane, finding the next tile, and so on, becomes repetitive and frustrating over time. Novice users, however, do not find this: when everything is new, the impact of retrieving each tile is unnoticed next to the difficulty of the concepts being dealt with. The toolbox is an excellent discovery mechanism for novices, the ease of getting something going is a significant driver of engagement, and the lack of syntax errors removes a major confound faced by novices. Novices are eventually no longer quite so novice, and so may want to move on.

A commonly-repeated maxim is that a good programmer can easily learn a new language. Novices are not good programmers, however, and a course structure predicated on making a language transition will likely run into trouble. Nonetheless, introductory tertiary courses in Scratch and Alice move on to other languages early, often within the first course, as programs become too complex for such languages. This has implications for the design of educational languages more generally as well: because an educational language explicitly expects learners to move on to other languages afterwards, the language must support the learner for long enough to allow them to build sufficient competence that they can successfully transfer their skills to another language.

4.2. Event versus Process

One issue with language transitions is that they are essentially "one-way" *events*: the learner must apply what they know about the earlier language to the later, but movement in the other direction is restricted. This is a problem not only for transferring concepts, but because this "event" model makes the two sides seem qualitatively different and opposed.

Powers, Ecott, and Hirshfield found that students learning Alice and a textual language in the same course frequently felt that Alice was not a "real" language [30]. Students who struggled with the textual-language part of the course felt that what they had been doing in Alice "didn't count" or was "too easy", that textual code was "real programming" and were inclined towards believing that they were not actually capable of programming; this inclination is harmful in itself. Lewis *et al* found that more students rated a picture of random green-on-black symbols from the film *The Matrix* as "definitely" or "somewhat like" programming than an image of the Lego Mindstorms programming environment (a colourful drag-and-drop system), despite the fact that those students had been learning Scratch [39, 40]. These examples are just some of the motivating concerns we had about visual-textual transitions when setting out to design Tiled Grace.

By contrast with approaches moving between multiple languages supporting a single paradigm each, Tiled Grace has a deliberately permeable barrier: a user can use the visual language, the textual language, and the visual language again, even within the same program if desired. As in BlockEditor [20], permitting both views avoids the transition event altogether, so that moving from blocks to text becomes a *process* rather than an event. A programmer can start to move

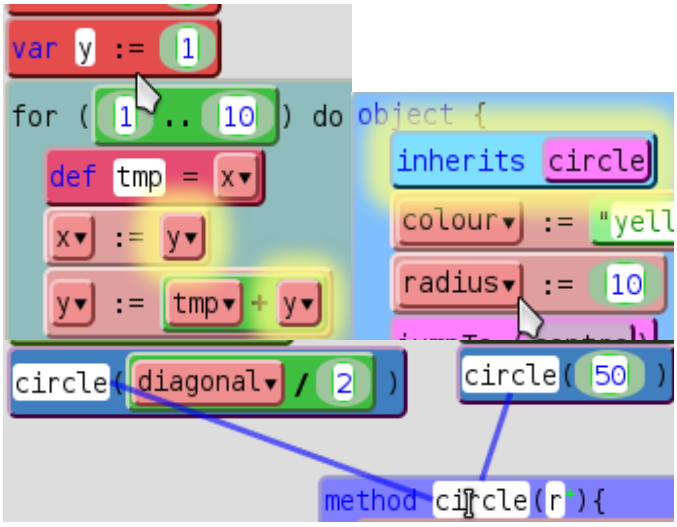


Figure 3. Overlaid dependency indicators for variables, methods, and inherited fields.

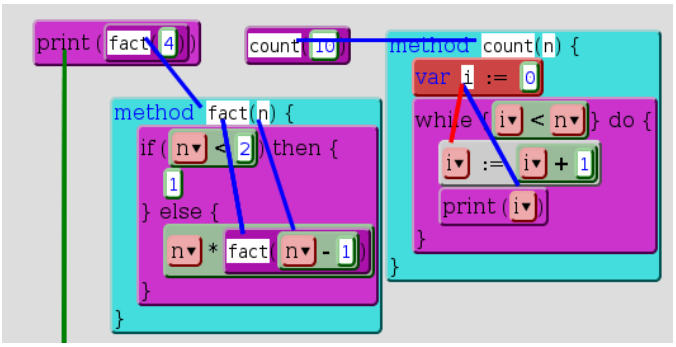


Figure 4. Composite image of multiple overlays at once in an alternative design.

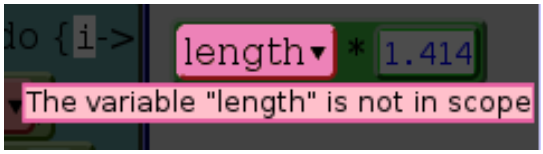


Figure 5. Marking a static scope error in the program where a variable reference has been moved out of its defined context.

to text as early as the first day, and draw out the process as long as necessary until they are truly comfortable working with textual code.

Throughout the process, both modes, and all of their past programs, remain available to the programmer. At all times, the user can see that what they were doing with blocks is exactly the same as what they are doing with text (or, indeed, the other way around). Both modalities clearly "count" as much as the other, and they are clearly both programming. Allowing movement in both directions necessitated some further trade-offs (particularly that the programmer can only switch views when there are no static errors in their program), but we considered it appropriate to the goal of the language. This textual-tiled combination was our original grounding conception for Tiled Grace.

4.3. Relationships and Dependencies

In a block language, and particularly one with arbitrary layout like Tiled Grace, it is possible for the declarations and uses of variables and methods to be dispersed around the screen where they may not be obvious, which could lead the user to break their program through being unaware of the dependencies between parts of the code. To preëmpt these incidences, we included two overlays to show the dependencies between these items.

The top left of Figure 3 shows the mouse pointer hovering over a variable declaration, with two uses of that variable highlighted. Similarly, hovering over a variable use site marks the declaration site. These indications occur anywhere in the program, for variables, constants, and method parameters. The top right shows that the "radius" field has been inherited from "circle" through a similar highlighting (the inheritance system of the underlying Grace language is object-based and blurs fields and variables [41]).

The bottom of the figure shows an unrelated method declaration, `circle`, with the mouse hovering over it. The overlay draws a line between the declaration and each use site of the method, wherever it is in the program. Again, if the user hovers the cursor over a call site, the line will indicate the declaration.

These indications are especially important to aid new users unfamiliar with the language or libraries. In our experiment, we would have people use the system with minimal training, and it was valuable that they should know the relationships between different parts of the task programs they were given. We considered an alternative dependency indicator, in Figure 4, that used lines for every indicator and showed all applicable dependencies at once. We found this overlay too busy and shifted to using the highlights from Figure 3 for everything but methods, and stopped showing possible assignment sites of variables altogether. We are not aware of a block language that successfully shows all of these dependencies, but we were inspired by the DrRacket editor for the textual Racket language [42] in the alternative approach.

4.4. Errors

While Tiled Grace was mostly modelled after Scratch, a key difference between Tiled Grace and Scratch, but much less so between, say, Tiled Grace and Alice, is that we made especial effort to provide error detection and reporting in the visual editor [5]. While in many block languages all programs can run regardless of missing or broken parts, we require that the program be well-formed to allow it to run, and prevent a wide range of errors from entering the program in the first place. To a large extent this choice is forced upon us by the need to match with a textual language, but we also believe that reporting errors early and often is beneficial.

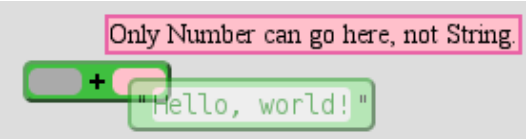


Figure 6. The display of a simple type error the user is making, where they try to place a string tile somewhere that only numbers are permitted.

While block-based editing prevents most syntax errors, the user may still omit filling in required components — for example, not specifying a variable name or leaving the hole on one side of an operator empty — or invalidate the program in other ways by moving a reference to a variable outside its scope or filling in an unsuitable value. We included a persistent graphical indicator of the current validity of the program in the interface: when it turns red, the program is somehow invalid. The user may hover over it to highlight all existing errors, which are labelled in situ with their cause (for example, an empty hole may have the message "Something needs to go in here"). These error sites are shown by desaturating all of the code area except the error sites, and overlaying an associated error message at the site, as seen in Figure 5. We investigated a number of approaches to indicating static errors, including overlaid arrows, persistent adjacent markers as used in App Inventor 2 [13], and a visible list of errors, but settled on this approach as a balance of space usage and clarity. The locality of error reporting and error messages is commonly better than is achieved in even advanced IDEs for conventional textual languages, which we feel is a key advantage that block-based languages can have.

Similarly, while type checking is commonly recognised as helpful in textual languages, translating that across to visual languages is challenging. We wanted to catch errors as early as possible, so it was important that we be able to detect and report type errors where possible, in a way that the user would see and understand.

We chose to use a variant on our error overlay approach to report errors as the user tries to perform the action that would cause an error, while also preventing the user from doing so. Any hole can be annotated (in the implementation) with the types it will accept. Any tile can be similarly annotated with the type of the object it represents. When the two do not match, the tile placement is not permitted. Most block languages have some variation of this "repulsion" behaviour for at least some of their tiles, but we hope that the simultaneous overlay makes clear why the tile cannot be placed where desired.

For example, a string tile is annotated with the type "String", and both holes in a + tile are annotated as accepting only "Number". When the programmer tries to place one into the other, as in Figure 6, the hole is marked in pink and an error message displayed nearby: the user will not be able to drop the tile into the hole. Type errors are in this way prevented from being introduced into the program in the first place, but the user also understands why they were not able to do what they wanted. This sort of live feedback is another aspect of block languages that is both clearly useful and difficult to replicate in conventional textual languages.

4.5. Shapes

We thought it would be helpful to indicate to users the appropriate placement of tiles before they move them. Scratch partially achieves this effect through its "jigsaw puzzle" pieces: holes and tiles of different types and roles have different physical shapes, so a boolean constant or expression will not fit into a numeric expression. While immediately understandable, the approach has flaws, notably that there is a limited range of sensible shapes that can be readily distinguished and consequential limit on the number of types that can be in the system. As well, "multi-type" holes are very difficult: in Scratch it is not possible to have an array of booleans, only of its combined string-number type. These constrictions make this approach problematic to implement in Grace, as it is a language with many extensible mostly-structural [43] types, and several places that can hold variables of any type (for example, variable declarations and equality tests).

Scratch uses shapes for both types and some grammatical categories, as do a number of other block languages: some blocks can only appear at the start of a stack (rounded top), some are statements (notched), and some are values (rounded). Tiled Grace does not do this: given the underlying language, every node can be legitimately adjacent to any other, and the only node that can never appear in any kind of value position is a method declaration, while variable declarations can appear in method bodies (which are expressions) but not some other expression sites. For most nodes, it is their (return) type that would limit their possible locations, and this would provide the same effect as grammatical restrictions for these nodes. The text-as-primary philosophy of Tiled Grace means that a wide range of possible programs must be representable, particularly given that dialects may extend or replace even basic control structures.

We considered colour-coding types, such that our any-type holes would be a neutral colour, while strings, numbers, booleans, other objects, and dialect definitions would have their own colours which could be matched on both tile and hole. Similarly to using shapes, however, the number of readily-distinguishable colours is a limit on the number of types that can exist, particularly if user-defined types (such as those of custom objects or classes) are possible.

Other block-based systems not aimed at conventional programming, such as Lerner *et al*'s Polymorphic Blocks system [44] and Vasek's TypeBlocks [45], face some similar obstacles. Polymorphic Blocks uses different shapes to represent different kinds of entity, but its equivalent of our holes, "ports", behave differently. A generic, untyped port is rectangular, but it may have connected ports elsewhere, each set of which is highlighted in the same colour. When a shaped item is connected to one of these ports, the matching ports all take on the same shape. Similarly to our animated transitions, Polymorphic Blocks animates each new shape moving from

source to destination. TypeBlocks has a similar philosophy with different shapes and entities.

Polymorphic Blocks supports generic parametric polymorphism in this way, but does not (yet) address the proliferation of shapes. Within the user experiment Lerner *et al* conducted, complex shapes are created only by nesting the small number of base shapes inside one another, with scaling down when required. Conceivably some version of this scaling can be applied more broadly, but we find it difficult to imagine applying it to complex object types. Designing Tiled Grace, and observing other work subsequently, has led us to the view that block languages must make a (fairly early) choice: use jigsaw-puzzle shapes and have a very restricted number of types, or use some sort of feedback during an attempted error but allow unbounded type construction. Neither approach is innately better or worse in general, but depends on the intended application domain of the language. Nonetheless, it is a degree of fragmentation of approach that we would have preferred to avoid.

5. Experiment

We ran a usability experiment trialling Tiled Grace with 33 participants [5]. In this section we describe the procedure of the experiment and summarise the results that are relevant to this paper. An anonymised dataset from the experiment was published [46], and in Section 6 we perform a new analysis of published instrumentation data from the experiment to examine how participants used the different editing modalities.

Participants were primarily students enrolled in early undergraduate Java courses in the School of Engineering and Computer Science at Victoria University of Wellington, selected so that they would have some existing familiarity with the idea of programming. This experiment focussed on usability and engagement, rather than learning, so true novices were not considered suitable subjects at this stage. Studying learning would require pedagogical studies of textual Grace to have been completed already in order to distinguish the effect of Tiled Grace, and we were mostly interested in whether the system design was practical at this point. The experimental design was guided by some key questions we wished to answer (as well as by practical considerations, particularly timing). We wished to find out whether users found the ability to switch views useful, and also whether they appreciated the explicit animation connecting the two, a particular novelty of our approach. We wanted to see whether the error reporting and type checking we had built was useful to users. As a tool that users do not enjoy will not be used, we wanted to measure engagement. Finally, we wanted users to explore different parts of the system so we could discover any unanticipated problems or successes.

The experiment took place in March–April 2014. Participants were asked to use Tiled Grace to write, modify, correct, and describe programs, while we measured their use of different features of the system. Participants also completed questionnaires about themselves and their use of the system. This experiment was approved by the Human Ethics Committee of Victoria University of Wellington.

The experiment focused on collecting data about usability, engagement, use of the various features, and user behaviour in this environment. We will first summarise relevant results that feed into our thoughts on the design of the system and of block programming systems here; for further details of these results, and other results from the experiment that we will not rely on here, see the original study [5, 47]. In Section 6 we present a novel analysis of people's revealed preferences for different modalities and different tasks, analysing the published data set from the same experiment.

5.1. Procedure

Each participant first completed a pre-questionnaire about themselves before being given a brief introduction to the system. The experimental system was instrumented and all interactions recorded. There were a total of six tasks in the body of the experiment, presented one at a time by the experimental system:

Task	Initial	Description
0	Tiled	Warmup – discarded
1	Tiled	Change Fibonacci to factorial
2	Tiled	Correct errors in this program
3	Tiled	Swap behaviours of two objects
4	Text	Describe program without running
5	Tiled	No specific goal – finish at will

The tasks were chosen to cause every participant to encounter both views and the error reporting at least once, and to have them both understand and modify code. Task 5 was intended to measure implicit engagement, giving no set task but telling participants that they could continue to use the system if they wished, and move on to the post-questionnaire when ready.

5.2. Summary

We will briefly summarise relevant results from the experiment [5, 47]:

- Participants showed high levels of engagement on multiple metrics, including implicit engagement with the system once tasks were complete and Likert-scale feedback on the post-questionnaire.
- The ability to switch views was widely used, with the median participant switching six or more times and 75% at least four.
- One quarter of participants spent more than half their time in text mode, and one half of them spent less than a third of their time in text mode.
- More-experienced participants viewed the system less favourably than less-experienced participants, as shown in Figure 7.
- The error reporting overlay was the aspect most often mentioned positively unprompted, by one third of participants.

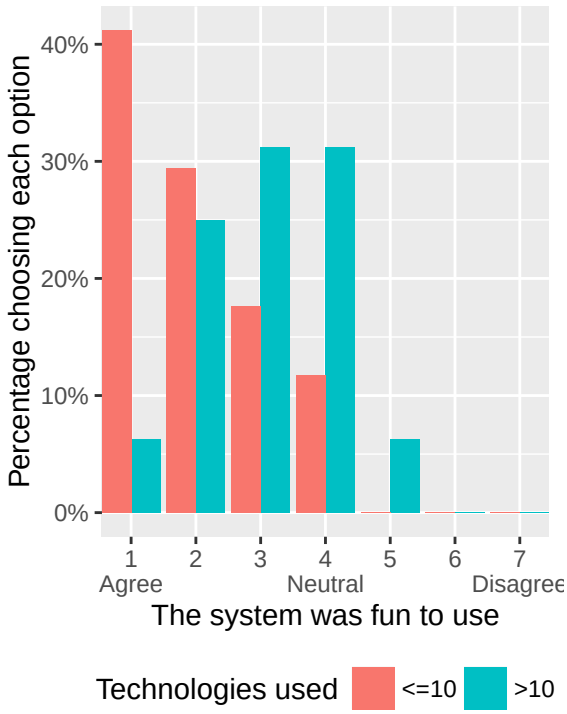


Figure 7. Participants' agreement with "The system was fun to use", one of a suite of engagement metrics in the experiment, split in half by a metric of experience relating to past programming and technological usage.

- Using both views in combination was also identified as helpful to understanding, mentioned by 9 of 33 participants and used by more.
- Many users found drag-and-drop frustrating, 40% mentioning it unprompted, and around 15% of participants had debilitating difficulty completing tasks with drag and drop.
- Most users did not switch to the tiled view in task 4, which only asked them to comprehend and describe the code.

This last point was not what had been expected while planning the tasks. Further analysis showed that of those 15 who did switch, half used tiles for more than 80% of the time spent on the task, and half for 35% or less, with nobody in between, and that while more-experienced users were more likely to switch, they were equally likely to be in either of those two groups.

This divergence of approach for a simple comprehension task suggested a difference in underlying preferences of modality between people, and led us to conduct the novel analysis presented in the next section, of how participants actually used each view during the editing tasks.

6. Preferred Modalities

The experiment with Tiled Grace provided a unique opportunity to examine which modality — blocks or text — users preferred for particular tasks. Spurred by discussions at the Blocks & Beyond workshop in 2015, we conducted an exploratory analysis of the collected data from the original experiment, which is publicly available [46], to see what trends might exist in the use of each mode.

We caution that, because of the overall sample size and the nature of breaking it down into further subgroups, these trends can only be suggestive of future research avenues. We did not have any particular hypothesis about what we might find, but these results will provide hypotheses for future research.

We focused on two areas specifically. First, we collected all changes made in "short" sessions in the tiled mode, which we defined as those logging ten non-automated events or fewer. We take these short sessions as representative of the user electing to perform specific actions in the tiled mode rather than as text. Secondly, we examined *all* text-editing sessions. In part, that is a technical limitation — unlike for the tiled mode, text operations were not discretely logged, only snapshots of the code — but we found it acceptable because text was not the default mode for any of the analysed tasks, so any use of the text mode indicated a deliberate choice by the user. We considered analysing all tiled sessions as well, but large sessions inevitably involve assembling whole programs and are not informative for this kind of analysis, as they reduce the measurements to counting the number of drags, new tiles, and so on, that are required to complete the tasks. As a result, however, some text sessions are much longer in wall-clock time than a short tiled session could reasonably be, and it is possible that more complex operations can be performed in them. A larger study might be able to tease out more detail from such data, and this exploratory analysis may suggest avenues to focus on. No participant spent all of their time on a task in the text view, while several spent all of their time in the tiled view. In both cases, we did not include tasks 0 (warmup) or 4 (comprehension only) in the analysis.

6.1. Editing as Text

We manually coded all 83 text-editing sessions by comparing the code before and after. Where the user made changes, and then undid them either themselves or through using our revert-changes option after a failed compilation, we recorded that fact mechanically.

Each session could be assigned multiple codes if more than one operation occurred during the session, but ancillary modifications as part of a broader code were not included (for example, "cut and paste" does not entail "delete"). A "before" and "after" snapshot of the code was mechanically obtained for each session and the two were compared by hand. The first author assigned each change a code or codes, in consultation with a colleague, after a first pass identifying candidate codes such that all present modifications were covered by a code. The initial set of candidate codes was taken from the codes

Table 2. Proportions and frequencies of codes of text-editing sessions. Each session could be assigned more than one code. Exp indicates the proportion of users in this code who were in the more-experienced half of the sample. 53% of all text sessions resulting in changes were by more-experienced users.

Freq.	Code	Prop.	Exp.
39	No change	47%	57%
4	Accepted offer to revert	5%	100%
4	Made and undid own changes	5%	67%
10	Change value of string/number	12%	57%
8	Cut and paste	10%	71%
8	Assemble complex code	10%	57%
8	Delete code	10%	43%
6	Change operator	7%	20%
6	Copy and paste	7%	83%
5	Subvert tiled error checking	6%	50%
3	Rename a variable	4%	33%

established for tiled sessions in the next section, and extended for text-only operations. Codes for operations that could only exist in tiles, such as drags, were also deleted.

47% of all text-editing sessions (39) made no change to the code, and seem to have been "just looking". We had earlier hypothesised that users may have found the ability to look at the code in two ways useful simply to break the monotony and to get an overview of the code, which we meant primarily as users switching from text to tiles to see the structure manifested graphically, but it appears that this may have been the case in both directions. This result may be a point in favour of "dual view" visual languages that display text and a visualisation simultaneously. 21 of 33 users (64%) had at least one empty text session.

43% of all text-editing sessions (36) had modified the code at the end. 15 of 33 users (45%) had at least one such session. The remainder of text sessions undid the changes they had made in one way or another. Table 2 shows the frequency and proportion in each code. Sessions may be assigned multiple codes.

The single most common modification was to *change the value of a string or a number*. Making such a change in the tiled view is among the easiest operations to perform as tiles, so switching to text to perform it seems a very deliberate choice. Most commonly, participants were making multiple such modifications at once, which may suggest a slight preference for textual interaction for repeated similar tasks.

The second most common operations were *cut and paste*, *deleting code*, and *assembling complex code*, in a three-way tie. By *assembling complex code* we mean producing a modification that would involve multiple drags and drops into holes. *Cut and paste* is exactly the operation that dragging and dropping tiles performs, so switching to text to perform it is not an obvious advantage. *Deletion* is also supported by the tiled interface, but deleting a single line from within a block of code was likely easier as text.

We believe that *changing operators* is an artefact of our implementation, rather than a meaningful result. The system sometimes interpreted attempts to select a different operator in an arithmetic tile as very short drag-and-drop sequences, which was a problem noted in the experimental results.

The most interesting task people performed in text mode was *subverting tiled error checking*. Although rare, it appears from subsequent analysis of the context that users frustrated with being unable to make logical, structural, or type errors in the visual interface switched to text in order to introduce the code they wanted (for example, adding two strings). This highlights an asymmetry in our system: the text and visual editors enforce slightly different rules on their code, which may have led to confusion. On switching back to tiles, these programs would have immediately reported errors.

6.2. Editing as Tiles

We categorised short and long tiled-editing sessions mechanically, discarding the long ones. Short sessions had ten or fewer logged interaction events. We also discarded short sessions at the beginning of Question 2, as this question presented a broken program that required a small degree of fixing before it was possible to switch views, and empty sessions at the end of tasks. Following this, 30% of all tiled sessions (70/234) were short.

37% of all short tiled-editing sessions (26) performed no operations at all, and seem to have been "just looking". These users may be viewing the structure of their textual program with a visualisation to help them to understand what they were doing. 12 of 33 users (36%) had one or more tiled sessions with no modifications.

The remaining 44 sessions performed some operation. In contrast to the text sessions, we also recorded interactions with the system that did not cause modifications to the program (no such interactions were available in text mode). Each session was coded for each operation that occurred within it, so some sessions were coded in multiple categories. Final coding was performed mechanically, after iterated manual inspection of uncoded segments created codes and rules to mechanise them. Table 3 shows the proportions of these sessions in each category. 19 of 33 users (58%) had one or more non-empty sessions.

As for text editing, the single most common operation to switch to tiled view to perform is *changing the value of a string or number tile*. It appears that this innocuous, but ubiquitous, operation is strongly polarising: it is a task that people will both switch to text to perform, and switch to tiles to perform.

The second most common operation is to *drag a single block into a hole*. This operation corresponds to textual cut-and-paste, moving code from one place to another.

Switching panes was counted if it occurred two or more times in combination with other operations other than creating a new tile, or was the only operation that occurred in the session. These sessions indicate users looking through the available functions to find the one they wanted, and are likely

Table 3. Frequencies and proportions of short tiled-editing sessions in each category. Exp indicates the proportion of users in this code who were in the more-experienced half of the sample. 37% of all non-empty short tiled sessions were by more-experienced users.

Freq.	Code	Prop.	Exp.
26	No change	37%	42%
13	Change value of string/number	19%	36%
9	Drag one block into a hole	13%	38%
7	Switch panes in toolbox	10%	67%
5	Meaningless drag	7%	50%
5	Rename a variable	7%	60%
3	See variables in scope	4%	33%
3	Assemble complex code	4%	0%
3	Fix code broken in text mode	4%	33%
1	Append one block to another	1%	0%
0	Delete code	0%	0%
0	Create new tile from toolbox	0%	0%

to be users primarily using text who wanted to know which methods were available. Some "no change" sessions may reflect the same use. The provision of a similar toolbox in the text mode, as in the work of Price & Barnes [48], would likely avoid these sessions.

Assembling complex code was naturally difficult within the restriction of a short session (measured in interaction events). The incidence of this code may not be informative.

Zero short sessions included dragging a tile out of the toolbox into the workspace.

6.3. Summary

Modifying the value of a string or number was the single-most common task performed in both modalities, in both cases commonly as the *only* operation performed in the session. This suggests to us that users have divergent views on the appropriate way to perform this action when given the choice, but we have no hypothesis as to why. We cross-referenced these codes with the original experiment's division of participants into more- and less-experienced halves; 62% (8/13) of such tiled modifications were by less-experienced users, compared with 40% (4/10) of the textual modifications. These proportions are roughly in line with the relative usage of the modes by each group found in the original experiment. There is an evident difference, but neither is overwhelming and given the sub-sample size nothing definitive can be said.

A plurality of sessions in both modes were empty, where the user performed no actions and simply looked at their code. These sessions were common among both more- and less-experienced users. We take this as partial validation of the hypothesis put forth previously [5, 47] that simply having two drastically different views available is found useful in itself.

Participants attempted to use the other view to work around limitations of the principal view they were using, for both good and ill. The interaction with operator tiles was a flawed design, and many users used the text view to make modifications they had difficulty with in the tiles, which was a positive use of the functionality. The enhanced error checking possible in the tiled view, which prevented many type and structural errors even being introduced, was an obstacle to some users who then used the text view to construct their broken programs, a negative use of time that could have been spent establishing *why* the code was not allowed. Neither technique was the intended use of the system, however, and both represent flaws or at least limitations in the approach taken by Tiled Grace.

Many participants opted to use textual cut and paste for operations that could easily be carried out by drag and drop. It may be that users are conditioned to perform such "moving" operations as text, which may also explain the high proportion of string or number modifications performed in the text mode.

Further research is required to confirm or refute all of these findings, particularly those with no a priori reason to expect them.

7. Discussion

A very common issue that has been encountered in introductory visual languages is that learners do not consider them "real" programming languages [30, 39, 40]. In some cases, the fact of being "easier" than text was interpreted to mean that the visual language did not "count" as programming, particularly when textual programming was later found challenging. Preconceived ideas about what is and is not programming, or what kinds of programming are or are not useful, can lead to block-based systems being regarded negatively by both current and prospective users. DiSalvo [49] found that learner perceptions of visual and textual programming systems varied according to the ultimate career goals of the user. Users with an interest in a programming career were more inclined towards a textual language than Alice, while those interested in media and design careers inclined the other way. For some, finding the textual language harder was in fact a point in its favour.

In Tiled Grace we aimed to avoid the perception that a block-based language was not "real" by having the block and textual language exactly match one another and presented equally. Experimentally, we did see users deploy both modes, and seem to understand the connection between the two. Similar to DiSalvo [49], and like Weintrop [50] and Matsuzawa [20] as well, however, we found that a number of participants very strongly preferred the textual mode and were even scornful of the tiled mode and its presence, to the point of using text almost exclusively even when the system design made it more difficult to complete tasks in that way. Three participants explicitly noted unprompted that they were predisposed to dislike GUIs. Other users exclusively used blocks. On one level, the fact that they were able to do so within the same system speaks to a more inclusive environment than a single-paradigm approach allows. On another, it seems that the presence of the other mode was at best neutral and possibly a net negative in overall perception for both groups of users, and might be off-putting. Neither group was more than 15% of users in our study. We did not ask about future goals in our experiment, but our more-experienced users (representing

the closest proxy we have to those whose self-conception is as programmers) were notably less positive about the system than less-experienced users. Our — somewhat informal — sense from observation and feedback in the experiment is that perceiving the block language as a toy, and thus the system as a whole as one too, played into this view. These perceptions may be a concern for the targeting and marketing of block languages.

Our experiences with Tiled Grace are paralleled by the experiment of Weintrop and Wilensky [27, 51] about multi-modal block-text environments. Their study involved high-school students in three conditions (blocks only, blocks and read-only JavaScript text, blocks and modifiable JavaScript text) who began in Snap! [22] and subsequently moved on to (textual) Java, over a ten-week course.

One facet of the study involved student performance over different concepts and modalities [52], having been exposed to both graphical Snap! and textual Java across the course and completing commutative assessments with parallel questions. For most concepts, students performed better with graphical code than textual. Students performed equally well on comprehension tasks regardless of modality, while in our experiment when given a pure comprehension task and a choice of views, most participants did not deviate from the default text view, despite overall preferring the tiled view in the experiment as a whole.

There may be no particular advantage in understanding code to either a block or textual view, despite blocks making the structure of the code explicit, at least for the relatively small programs in use in both experiments. What advantages do accrue from block editing would then all come from easier construction of programs, rather than easier understanding. Such a result does not seem to match with the self-reporting of participants in either study or elsewhere in the literature, and so needs further investigation.

When Weintrop's students were asked to compare Java and Snap!, those who expressed a view overwhelmingly said that block-based programming was easier than text, regardless of which condition they had originally been in. In our experiment, similarly, participants regarded the tiled view as somewhat easier to use, although not by the same 80% margin. Weintrop was able to conduct interviews with participants to attempt to establish the reasons behind these results, which can shed some light on our own findings.

Weintrop's analysis of the interviews in the study investigates (among other things), just *why* blocks were found easier. A number of points correspond to our study, and to how we found that participants used each mode in Section 6. One that did not, however, was that "blocks are easier to read" because the *language* of blocks was different and more "English". We believe that this limitation is quite significant: to find out whether blocks, themselves, are helpful, the languages should match as closely as possible. With Tiled Grace, where they match exactly, we did find that users regarded the tiled view as making it easier to deal with syntax, matching those from the Weintrop experiment who mentioned punctuation, balanced brackets, and other syntactic noise as reasons they found blocks easier. It is important to be careful not to conflate elements of a block language with elements of the block paradigm itself (albeit that this is very difficult to avoid with current systems).

Our blocks did not have different shapes for different data types, but were approximately colour-coded by topic (for example, variable declarations and variable references were similar shades). We had wished to make them shaped to complete the "jigsaw puzzle" metaphor, but were unable to do so with the wide range of types possible in a general-purpose language. The Weintrop study finds that these shapes were one of the key reasons that users said they found blocks easier to use: the shapes of a block and hole communicate whether they are compatible, while top and bottom connectors made sequencing explicit. Our type checker would disallow many invalid combinations, but only after the user had tried to perform it, and in some cases they would then switch the textual view (which had less stringent immediate checks) simply to create the code they wanted. It is possible that, had the blocks been obviously incompatible, users would not have attempted this to begin with. On the other hand, as Weintrop found, the puzzle-piece metaphor can lead to confusion among learners who expect there then to be "a" solution to a problem, as in a jigsaw, rather than many possibilities. This expectation did not noticeably appear in our experiment, but some participants did express that they thought they had completed some tasks "wrong".

Weintrop and Wilensky's other two reasons, that composing code was easier or more accessible as blocks, and that blocks were memory aids, were both borne out in our study. The pane-switching tiled sessions from Section 6, and likely some of the empty sessions, appear to be exactly using the block side as a memory aid. We also observed "bottom-up" construction of complex expressions to be common, often using an out-of-the-way corner of the workspace to build up the expression before moving it into place. Because Tiled Grace enforced scoping of variable-reference tiles (necessary, as the textual language has traditional lexical scoping, and in fact intended to help by offering a list of available names), assembling code in this way was sometimes not possible, to the frustration of the user. One of the trade-offs in integrating the textual and block languages that we had not considered was that this sort of "inside-out" construction, which is very natural and widely-reported [53] in block languages, would be stymied by error prevention in the textual language. Meerbaum-Salant *et al.* have argued that this style is in fact a "bad habit" [53], and that it has a longer-term detrimental effect on learners. It is not obvious whether this aspect of Tiled Grace is helping or hindering, and precisely what the long-term goals are may again be important.

A later experiment by Weintrop and Holbert [50] used Pencil Code as a switchable dual-mode environment, with the goal of finding out how each mode was used, as in our analysis in Section 6. Unlike in our results, the majority of switches to blocks were to add new blocks to the program, and empty block sessions were no more than 5.7% of the

total. Moving a block inside another was somewhat common in both experiments. While we considered short sessions as a whole, Weintrop and Holbert looked only at the first action taken after a change of modality. Given this, it is remarkable that so many more sessions created new tiles. Weintrop and Holbert note that two-thirds of the time a new block was added in this way, and 86.7% of the time a control block was added, it was the first time that block had been used. The nature of our experiment, where starter programs were already provided, may have limited the number of occasions to add a block in this way. The most significant difference between the subject populations for our experiment and theirs is that our users were adults who primarily had past programming experience, while Weintrop and Holbert's were a mix of novice learners in high school and graduate students outside of programming (the groups are not separated in this part of the analysis). It may also be that these populations are the cause of these different observations, or that the nature of the language affects user behaviour in some way (in particular, Tiled Grace had a significantly lower total number of distinct blocks available).

Similar to in our original experiment, Weintrop and Holbert find a wide range of levels of use of each modality, with a trend for increased text use to go along with higher degrees of experience. They also note some users who strongly prefer either blocks or text almost exclusively, as did we. These results are in line with the goals of Tiled Grace's design, and Weintrop and Holbert suggest that they provide support for the dual-modality approach as providing for "low-threshold/high-ceiling" programming environments.

Matsuzawa *et al.*'s experiment using both Java and Block, which translates to and from Java, in an introductory programming course finds a wide range of different levels of use of textual and block editing [20]. The BlockEditor system supports exporting to Java and importing from Java, but makes no particular explicit mapping between the two in itself; it does not appear that this caused any widespread trouble for learners, which may suggest that Tiled Grace's emphasis on making the mapping manifest through animation is unnecessary, or it may be a reflection of the teaching structure employed.

Across a fifteen-week course students tended to use less of the block view as time progressed, but with highly varied rates of change between students as well as individual fluctuations. The rate of backsliding and inter-student variation supports our goal of making the transition be a process, rather than an event, as many students would have been left behind given any particular transition point. It is notable, however, that after a large task in the eigth week the rate of block usage did drop dramatically, and stayed low, so it is possible that there is a distinguished point where text becomes preferable. It is also possible that using the text view out of necessity, when the block view of a large program has become unwieldy, acclimatises learners to it, and simple exposure is all that is required to cause the text modality to take over.

Experiments using TouchDevelop with secondary-school students [54] found that students were rapidly able to develop non-trivial mobile applications in that environment. Long-term users were able to produce advanced applications with no formal instruction other than sample code, while shorter sessions showed good performance regardless of programming background. One posited explanation is that, because TouchDevelop's tap-based interface surfaces the available actions in a given context on demand, it promotes experimentation with a wider range of options with immediate feedback. Similarly, users will less often need to search for the block they want to use if TouchDevelop presents what it expects they may need automatically. The "memory aid" activities from our and other systems should not need to occur in TouchDevelop, and some sort of context-aware suggestion mechanism would be an advantage to the user of a block language.

Our experiment found that a sizable proportion of participants (around one in six) had debilitating trouble using drag-and-drop interaction with the system, despite everyday use of mice and keyboards. Tiled Grace relies on the mouse pointer being over a drop target, which we had taken as the standard drag-and-drop behaviour, and these users found that task very difficult. Other contemporary block languages have similar behaviours: Scratch and Blockly use a point in the upper left of the bounding box of a tile instead of the mouse pointer, while Pencil Code uses a similar point in combination with a Euclidean distance metric to choose the closest target. Neither of these seems obviously more intuitive. We have not seen studies reporting on this significant of a difficulty dragging in block-based languages, but have anecdotally observed tiles going otherwise than where they were wanted in all of these systems. Past human-computer interaction research [55–57] has found that point-and-click interfaces may involve fewer errors and be faster than drag-and-drop, although recent research with children [58] has shown that they may both expect and prefer drag-and-drop interfaces. Ludi [59] has noted accessibility problems with contemporary block languages for users with motor or visual impairments. These are a significant issue that is fundamental to the interaction paradigm most of these languages currently use, and which is only exacerbated by the observations in our experiment. Other block-like structured editing paradigms, as in Kölling *et al*'s "frame-based" Stride [25] language, or the "point-and-tap" TouchDevelop [24] interface that requires no continuous action, may be more suitable. Of the drag-and-drop approaches, Pencil Code's appears the most usable, but still relies on free movement of the mouse.

When Powers, Ecott, and Hirshfield experimented with transitioning from Alice to Java (with BlueJ) in an introductory programming course [30] they observed that many students

> were intimidated by the textual language and syntax, and seemed to have a difficult time seeing how the Java code and the Alice code related

even when working with exactly corresponding Alice and Java code. In our experiment, which used exactly parallel languages in both textual and visual modes, and in Matsuzawa *et al*'s [20], which did not, understanding the relation between the two did not seem to be an especial problem, but intimidation

by text was evident. Students in Matsuzawa *et al*'s study with lower self-rating of their skill avoided the text view, but none of our self-rating questions showed a strong correlation with use of either mode; we did not have a generic "rate your programming skill" question, however. Weintrop's earlier experiment appears as though a similar trend may exist, but explicit data is not available. As our experiment was much shorter than any of the others, the trend may not have had time to emerge.

8. Lessons and Conclusion

We see mixed success and weakness in our experiences and results working with block-based languages. We will attempt to distill some key lessons from our experience with Tiled Grace that are applicable to block languages more broadly. These lessons draw from our and others' experimental results and observations, and from our experience designing and building an integrated block-text programming system.

A positive sign is that our experiment, and others', showed strong engagement with a predominantly block-based environment, and that **even when given the choice to use text users in an unfamiliar language largely preferred to use the blocks.** This held even though most of our users had some familiarity with text programming in other languages already.

One reason that some block systems have been found easier to use is that the *language* of the blocks is more accessible, or more "English", than conventional languages. This is a language-design element, rather than a property of block systems, and is a natural confound when assessing how helpful a system is, so **it is important to separate the effects of the language and the interaction paradigm when evaluating block systems**. Tiled Grace's use of identical block and text languages is one method of keeping this distinction clear, but designing textual languages that incorporate the benefits of block languages is another approach.

We also found an approach to reporting errors in block-based programs that was effective and well-regarded by experiment participants, including unprompted positive mentions and strong signs of effectiveness at communicating the issue identified. This approach could easily be applied to other languages in the same model, while conventional textual languages would find it difficult to provide the same level of immediacy. **Immediate in-situ feedback is effective and much easier in block languages than text.** We recommend incorporating some similar form of error reporting into all block-based languages, unless there is a strong pedagogical reason to turn ill-conceived programs into runtime debugging exercises.

Less positively, we found that **more-experienced users were substantially less favourable towards a block-based environment than less-experienced users**. While for purely novice systems this may not be an issue, it is a significant caution for systems aiming at broader markets or professional use. Even novices will become more experienced over time, and losing engagement is a problem for, at least, retention.

We believe that supporting people to move on to other paradigms (whether through our and Pencil Code's dual-mode approach or otherwise) is crucial to successfully deploying block languages for programming education (while general-purpose or domain-specific block languages may not wish to do so). The experience-engagement results of the previous paragraph are one reason why, but more important are the educational psychology aspects discussed in Section 4.1. Thus, **block languages *for programming education* must have an exit strategy.** Course structures predicated on simply starting in a block language and moving on to a more conventional language after a few months are fraught with danger unless significant care — and time — is put into providing explicit bridging instruction to help learners map concepts from one world to the other. Languages that do not facilitate this process are doing their users a disservice, but allowing and encouraging mixed use appears effective.

While it is well-known that experienced users can find the back-and-forth dragging of Scratch and other block languages frustrating, it has been less noted that **drag-and-drop visual editing is a significant problem for some users**, even those without physical limitations on doing so. In addition, drag and drop is much less accessible than text editing for anybody unable to use a mouse easily, or to see what is on screen. If block languages aim to democratise programming, they cannot do so by excluding already-marginalised people further. Block paradigms that are not dependent on drag and drop may be more suitable for everybody.

Finally, we found that users made heavy use of our view-switching ability simply to see "the other side": they did not always want to make changes there. These results emerged from the instrumentation in our experiment and from free-text feedback. It is not all-or-nothing: **providing multiple views of code helps users be more comfortable with it**, even if the code is not edited (or editable) in one view or another.

Block programming is currently undergoing substantial growth, but we should not lose sight of potential negative aspects. Long-term thinking is required in their design and use, and experimentation to determine which aspects of them are helpful, and which are ancillary or negative.

References

[1] A. P. Black, K. B. Bruce, M. Homer, J. Noble, A. Ruskin, and R. Yannow, "Seeking Grace: A new object-oriented language for novices," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: ACM, 2013, pp. 129–134.

[2] A. P. Black, K. B. Bruce, M. Homer, and J. Noble, "Grace: The absence of (inessential) difficulty," in *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! '12. New York, NY, USA: ACM, 2012, pp. 85–98.

[3] J. Noble, M. Homer, K. B. Bruce, and A. P. Black, "Designing grace: Can an introductory programming language support the teaching of software engineering?" in *Software Engineering Education and Training (CSEE&T), 2013 IEEE 26th Conference on*. IEEE, 2013, pp. 219–228.

[4] M. Homer and J. Noble, "A tile-based editor for a textual programming language," in *Proceedings of IEEE Working Conference on Software Visualization*, ser. VISSOFT'13, Sept 2013, pp. 1–4.

[5] M. Homer and J. Noble, "Combining tiled and textual views of code," in *Proceedings of IEEE Working Conference on Software Visualization*, ser. VISSOFT'14, Sept 2014.

[6] M. Homer, T. Jones, J. Noble, K. B. Bruce, and A. P. Black, "Graceful dialects," in *ECOOP 2014 — Object-Oriented Programming*, ser. Lecture Notes in Computer Science, R. Jones, Ed. Springer Berlin Heidelberg, 2014, vol. 8586, pp. 131–156. [Online]. http://dx.doi.org/10.1007/978-3-662-44202-9_6

[7] M. Homer, K. B. Bruce, J. Noble, and A. P. Black, "Modules as gradually-typed objects," in *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, ser. DYLA '13. New York, NY, USA: ACM, 2013, pp. 1:1–1:8.

[8] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, Nov. 2009.

[9] A. Kay, "Squeak Etoys authoring & media," Viewpoints Research Institute, Research Note, 2005.

[10] S. Cooper, W. Dann, and R. Pausch, "Teaching objects-first in introductory computer science," in *ACM SIGCSE Bulletin*, vol. 35, no. 1, 2003.

[11] Blockly Project, "Blockly web site," https://code.google.com/p/blockly/.

[12] D. Wolber, "App Inventor and real-world motivation," in *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '11. New York, NY, USA: ACM, 2011, pp. 601–606.

[13] F. Turbak, D. Wolber, and P. Medlock-Walton, "The design of naming features in app inventor 2," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, July 2014, pp. 129–132.

[14] K. Chadha, "Improving the usability of App Inventor through conversion between blocks and text," Honors Thesis, Wellesley College, 2014.

[15] K. Chadha and F. Turbak, "Improving App Inventor usability via conversion between blocks and text ," *Journal of Visual Languages & Computing*, vol. 25, no. 6, p. 1042–1043, 2014, Distributed Multimedia Systems (DMS2014) Part I.

[16] P. Guo, "Proposal to render Android App Inventor visual code blocks as pseudo-Python code," https://people.csail.mit.edu/pgbovine/android_to_python/.

[17] D. Bau and D. A. Bau, "A preview of Pencil Code: A tool for developing mastery of programming," in *Proceedings of the 2nd Workshop on Programming for Mobile Touch*, ser. PROMOTO '14. New York, NY, USA: ACM, 2014, pp. 21–24.

[18] D. Bau, "Droplet, a blocks-based editor for text code," *Journal of Computing Sciences in Colleges*, vol. 30, no. 6, pp. 138–144, Jun. 2015.

[19] D. Bau, D. A. Bau, M. Dawson, and C. S. Pickens, "Pencil code: Block code for a text world," in *Proceedings of the 14th International Conference on Interaction Design and Children*, ser. IDC '15. New York, NY, USA: ACM, 2015, pp. 445–448.

[20] Y. Matsuzawa, T. Ohata, M. Sugiura, and S. Sakai, "Language migration in non-CS introductory programming through mutual language translation environment," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '15. New York, NY, USA: ACM, 2015, pp. 185–190.

[21] J. Mönig, Y. Ohshima, and J. Maloney, "Blocks at your fingertips: Blurring the line between blocks and text in GP," in *IEEE Blocks and Beyond Workshop*, 2015.

[22] B. Harvey and J. Mönig, "Bringing "no ceiling" to Scratch: Can one language serve kids and computer scientists?" in *Constructionism 2010*.

[23] D. Blank, J. S. Kay, J. B. Marshall, K. O'Hara, and M. Russo, "Calico: A multi-programming-language, multi-context framework designed for computer science education," in *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '12. New York, NY, USA: ACM, 2012, pp. 63–68.

[24] R. N. Horspool, J. Bishop, A. Samuel, N. Tillmann, M. Moskal, J. de Halleux, and M. Fähndrich, *TouchDevelop: Programming on the Go*. Microsoft Research, 2013.

[25] M. Kölling, N. C. C. Brown, and A. Altadmri, "Frame-based editing: Easing the transition from blocks to text-based programming," in *Proceedings of the Workshop in Primary and Secondary Computing Education*, ser. WiPSCE '15. New York, NY, USA: ACM, 2015, pp. 29–38.

[26] N. C. C. Brown, A. Altadmri, and M. Kölling, "Frame-based editing: Combining the best of blocks and text programming," in *Fourth International Conference on Learning and Teaching in Computing and Engineering*, ser. LaTiCE '16, 2016.

[27] D. Weintrop and U. Wilensky, "To block or not to block, that is the question: Students' perceptions of blocks-based programming," in *Proceedings of the 14th International Conference on Interaction Design and Children*, ser. IDC '15. New York, NY, USA: ACM, 2015, pp. 199–208.

[28] Playful Invention Company, "Picocricket reference guide, version 1.2a," http://www.picocricket.com/pdfs/Reference_Guide_V1_2a.pdf.

[29] S. Dasgupta, S. M. Clements, A. Y. idlbi, C. Willis-Ford, and M. Resnick, "Extending Scratch: New pathways into programming," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2015.

[30] K. Powers, S. Ecott, and L. M. Hirshfield, "Through the looking glass: Teaching CS0 with Alice," *SIGCSE Bulletin*, vol. 39, no. 1, pp. 213–217, Mar. 2007.

[31] D. B. Palumbo, "Programming language/problem-solving research: a review of relevant issues," *Review of Educational Research*, vol. 60, no. 1, pp. 65–89, 1990.

[32] D. N. Perkins and G. Salomon, "Teaching for transfer," *Educational Leadership*, vol. 22, no. 32, 1988.

[33] A. Robins, "Transfer in cognition," *Connection Science*, vol. 8, no. 2, pp. 185–204, 1996.

[34] D. N. Perkins and F. Martin, "Fragile knowledge and neglected strategies in novice programmers," in *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. Norwood, NJ, USA: Ablex Publishing Corp., 1986, pp. 213–229.

[35] J. L. Dyck and R. E. Mayer, "Teaching for transfer of computer program comprehension skill," *Journal of Educational Psychology*, vol. 81, no. 1, 1989.

[36] V. R. Delclos, J. Littlefield, and J. D. Bransford, "Teaching thinking through Logo: The importance of method," *Roeper Review*, vol. 7, no. 3, 1985.

[37] D. H. Clements and D. F. Gullo, "Effects of computer programming on young children's cognition," *Journal of Educational Psychology*, vol. 76, no. 6, 1984.

[38] T. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, p. 131–174, 1996.

[39] C. Lewis, S. Esper, V. Bhattacharyya, N. Fa-Kaji, N. Dominguez, and A. Schlesinger, "Children's perceptions of what counts as a programming language," *Journal of Computing Sciences in Colleges*, vol. 29, no. 4, pp. 123–133, Apr. 2014.

[40] C. M. Lewis, "How programming environment shapes perception, learning and goals: Logo vs. Scratch," in *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '10. New York, NY, USA: ACM, 2010, pp. 346–350.

[41] T. Jones, M. Homer, J. Noble, and K. Bruce, "Object inheritance without classes," in *30th European Conference on Object-Oriented Programming (ECOOP)*, 2016.

[42] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen, "Languages as libraries," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 132–141.

[43] T. Jones, M. Homer, and J. Noble, "Brand objects for nominal typing," in *29th European Conference on Object-Oriented Programming (ECOOP)*, 2015.

[44] S. Lerner, S. R. Foster, and W. G. Griswold, "Polymorphic blocks: Formalism-inspired ui for structured connectors," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, ser. CHI '15. New York, NY, USA: ACM, 2015, pp. 3063–3072.

[45] M. Vasek, "Representing expressive types in blocks programming languages," Honors Thesis, Wellesley College, 2012.

[46] M. Homer, "Tiled Grace experiment data," Available as `tiled-grace-experiment.tar.bz2` attached to [47] in the research archive of Victoria University of Wellington at http://researcharchive.vuw.ac.nz/handle/10063/3654., 2014.

[47] M. Homer, "Graceful language extensions and interfaces," Ph.D. dissertation, Victoria University of Wellington, 2014.

[48] T. W. Price and T. Barnes, "Comparing textual and block interfaces in a novice programming environment," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ser. ICER '15. New York, NY, USA: ACM, 2015, pp. 91–99.

[49] B. DiSalvo, "Graphical qualities of educational technology: Using drag-and-drop and text-based programs for introductory computer science," *IEEE Computer Graphics and Applications*, vol. 34, no. 6, pp. 12–15, Nov 2014.

[50] D. Weintrop and N. Holbert, "From blocks to text and back: Programming patterns in a dual-modality environment," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '17. New York, NY, USA: ACM, 2017, pp. 633–638.

[51] D. Weintrop, "Modality matters: Understanding the effects of programming language representation in high school computer science classrooms," Ph.D. dissertation, Northwestern University, 2016.

[52] D. Weintrop and U. Wilensky, "Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ser. ICER '15. New York, NY, USA: ACM, 2015, pp. 101–110.

[53] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari, "Habits of programming in scratch," in *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '11. New York, NY, USA: ACM, 2011, pp. 168–172.

[54] N. Tillmann, M. Moskal, J. de Halleux, M. Fahndrich, J. Bishop, A. Samuel, and T. Xie, "The future of teaching programming is on mobile devices," in *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '12. New York, NY, USA: ACM, 2012, pp. 156–161.

[55] K. M. Inkpen, "Drag-and-drop versus point-and-click mouse interaction styles for children," *ACM Transactions on Computer-Human Interaction*, vol. 8, no. 1, pp. 1–33, Mar. 2001.

[56] D. J. Gillan, K. Holden, S. Adam, M. Rudisill, and L. Magee, "How does Fitts' law fit pointing and dragging?" in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '90. New York, NY, USA: ACM, 1990, pp. 227–234.

[57] I. S. MacKenzie, A. Sellen, and W. A. S. Buxton, "A comparison of input devices in element pointing and dragging tasks," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '91. New York, NY, USA: ACM, 1991, pp. 161–166.

[58] W. Barendregt and M. M. Bekker, "Children may expect drag-and-drop instead of point-and-click," in *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '11. New York, NY, USA: ACM, 2011, pp. 1297–1302.

[59] S. Ludi, "Position paper: Towards making block-based programming accessible for blind users," in *IEEE Blocks and Beyond Workshop*, 2015, pp. 67–69.

Frame-Based Editing

Michael Kölling, Neil C. C. Brown, Amjad Altadmri
Department of Informatics, King's College London, UK
{michael.kolling, neil.c.c.brown, amjad.altadmri}@kcl.ac.uk

Abstract *In introductory programming teaching, block-based editors have become very popular because they offer a number of strong advantages for beginning programmers: They avoid many syntax errors, can display all available instructions for visual selection and encourage experimentation with little requirement for recall. Among proficient programmers, however, text-based systems are strongly preferred due to several usability and productivity advantages for expert users. In this paper, we provide a comprehensive introduction to a novel editing paradigm, frame-based editing – including design, implementation, experimentation and analysis. We describe how the design of this paradigm combines many advantages of block-based and text-based systems, then we present and discuss an implementation of such a system for a new Java-like language called Stride, including the results of several evaluation studies. The resulting editing system has clear advantages for both novices and expert programmers: It improves program representation and error avoidance for beginners and can speed up program manipulation for experts. Stride can also serve as an ideal stepping stone from block-based to text-based languages in an educational context.*

1. Introduction

Syntax errors are a well-known – and largely unavoidable – problem in text-based programming. The severity of the problem varies widely: They can range from a small nuisance slightly slowing down an expert programmer's workflow to an insurmountable hurdle stopping a novice programmer in her tracks. A significant body of existing published work explores which errors are problematic (for example [1–3]) and how to alleviate these problems via additional tools [4, 5], but it is clear that text-based programming and syntax errors are inseparable.

In this paper, we will introduce a re-thinking of editing interactions in programming environments, which we term *frame-based editing*. A reduction in the number of syntax errors made by a programmer is one advantage, and we will use this goal as one motivation for our new design. We will see, however, that this is not the only benefit. Various other advantages, including improvements in readability, better navigation, and faster program manipulation also follow from our design.

For beginners, syntax errors present a serious and particularly annoying hurdle [4, 6]. Serious, because beginners often lack the skill to remove the error; syntax may still be mysterious, and what later becomes trivial is still the main focus of the programming activity [7]. Annoying, because syntax errors typically do not provide a path to any useful insight or learning experience. While the encounter of a semantic error may expose a misunderstanding and lead to a useful and meaningful learning experience, overcoming a syntax error does not usually teach an important concept of programming; it merely enforces an arbitrary rule to be memorised.

The problem is usually compounded by the dismal quality of error messages in many of our programming systems. Error messages are typically written by compiler writers, and little effort is made to include information useful to beginners. Many errors are reported from the viewpoint of the parser or type checker (such as the well-known "Illegal start of expression" or "Identifier expected" messages in common Java compilers), and in many cases, little useful information is given to a novice [1, 5]. Weinberg [8] summarises this succinctly: "*[H]ow truly sad it is that just at the very moment when the computer has something important to tell us, it starts speaking gibberish.*"

One instinctive goal might be to improve the quality (specificity and correctness) of error messages [9, 10]. However, we can do better: A more worthwhile goal is to avoid syntax errors in the first place, for the benefit of beginners and experts.

2. Blocks: Avoiding Errors in Programming

When thinking about novice programming, especially for young learners, it is useful to consider other successful areas of learning. When children play with Lego blocks, for example, they typically learn various techniques of construction without ever reading a manual and without any error messages involved in the process. Lego blocks have the inherent quality of allowing experimentation and fitting together only in well-defined ways. It is not possible to connect two Lego bricks erroneously – if they fit together at all, they fit correctly. There are no type errors in Lego bricks.

The equivalent of Lego bricks for programming are block-based languages, such as Scratch [11]. These languages provide statements of the programming language as direct manipulation "blocks", which can be snapped together in syntactically valid constellations.

Direct manipulation programming systems for beginners have become widely popular in the last 10 years. In these systems, language statements are visually represented as user interface entities that can be manipulated: dragged, dropped,

DOI reference number: 10.18293/VLSS2017-012

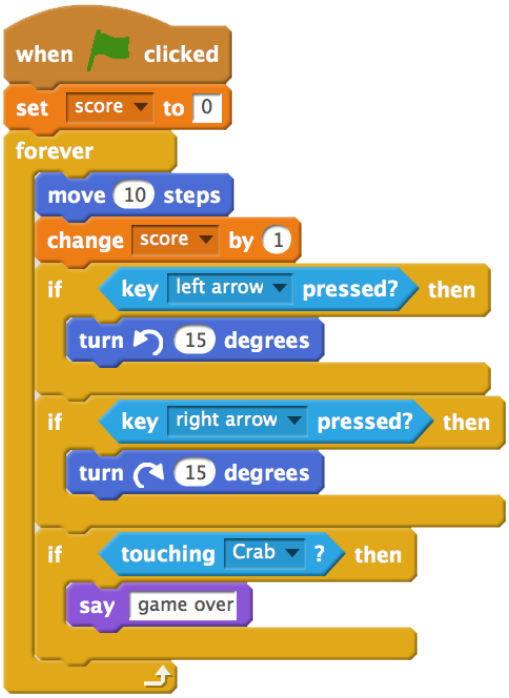


Figure 1. Block-based program notation in Scratch

snapped together, or double-clicked to activate. Due to the block-like appearance of these statements in many systems, they are often referred to as "block-based" languages.

The most popular of these systems in early programming education is Scratch (Figure 1); other notable examples include StarLogo TNG [12], Alice [13] and App Inventor [14]. While these systems differ in many aspects and significant details, they are similar enough for the purpose of our discussion here to be treated as one common class of system.

2.1. Benefits of Block-based Languages

Due to the visual and direct manipulation nature of programming elements, block-based editing achieves a more playful feel of programming, leading young learners to more experimentation and exploration [15]. All possible statements and expressions are represented on screen, supporting recognition rather than requiring recall for selection of statements.

What is more, most common syntax errors found in typical text-based languages are avoided; they simply cannot be made. It is not possible, for example, to forget to close the scope of a conditional statement – the statement is either present in its entirety or not at all. The syntax of statements cannot be mistyped, and statements can only be snapped together in syntactically valid combinations.

Indeed, the error prevention goes further than simple syntax: Where parameters are expected, statements are often created with reasonable default values already inserted. While the default value might be not what the programmer desired, the program is at least syntactically valid and will execute.

Type errors can also be avoided: statements expecting typed expressions can contain slots of specific geometric shapes,

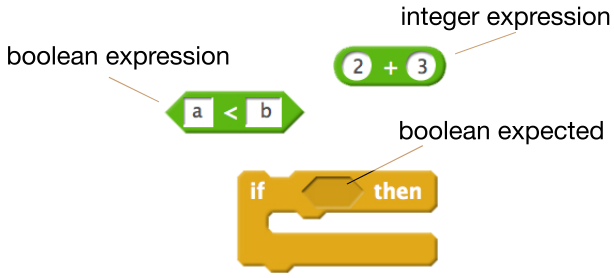


Figure 2. Shapes indicating types of expressions

where the shape denotes the expected type (see Figure 2). Expressions are represented in these shapes; only if the expression type matches the expected type will the shape fit, and the blocks snap together. In some systems, it is not possible to assemble a combination that would represent a type error. (More advanced use of shapes for types in block languages are also possible [16].)

For beginning programmers, these systems offer several tangible benefits: they reduce the rate of errors, allow a better exploration of the available language, make assembly of programs easier, almost always lead to executable code, and increase subjective satisfaction [17]. Many of these benefits are also clearly desirable for experienced developers, so the obvious question is: Why don't we all program like this?

2.2. Limitations of Block-based Languages

Experienced programmers clearly favour text-based languages over direct manipulation systems. The reason lies in several severe limitations of block-based languages: They suffer in readability, viscosity and navigation support.

When programs in block-based languages become large (and "large" in this context is reached very quickly – a few pages of program code already feels "large" in these systems), they become hard to read. The graphical nature – colour, shape and three-dimensional appearance with light and shadow effects – adds visual noise that can overwhelm the program structure and distract from program semantics.

Navigation in these systems is also comparatively poor. Quickly switching focus between declaration and use of an entity is typically not well supported, making reading and exploration of an existing program harder than in typical text-based environments. The ability to position code fragments arbitrarily in Scratch makes it difficult to systematically read a whole program, and little support is provided for organising the code, higher level structuring or obtaining an overview.

Viscosity – the resistance to change [18] – is high in these systems. Making changes to existing programs requires more effort and takes more time than in professional text-based environments. Changing large existing code bases – rather than the development of new, small programs – is the bread and butter work for most experienced programmers, and this is precisely where block-based systems do not offer adequate support.

3. Text-Based Programming

Text-based programming is the current standard in most programming systems for proficient users. Many benefits are obvious: Text is a very expressive, flexible medium that allows fairly clear and concise definitions of programs. Humans are very practised in reading, navigating and understanding text-based representations.

However, text-based systems have a number of limitations. Most of these encumber the programmer with work that could easily be automated by a more sophisticated system. Performing the work impedes productivity, sometimes just by requiring time to perform, and sometimes by adding cognitive load that distracts from the intrinsic complexity of the programming task. Common limitations include:

- Programmers must often type out program statements. Typing out keywords of the language is unnecessary work that slows down program entry – previous work has suggested an association between faster typing speed and increased programming performance [19, 20]. Often, the conceptual space of valid entities to be entered at any point in the program is fairly limited, and more efficient selection interactions could be devised.
- When entering program text, programmers must ensure syntactical correctness of statements (including correct spelling and punctuation). When the statement intended is recognised (e.g. once the programmer has entered the *while* keyword), it is unnecessary to make the human programmer responsible for correct orthography of the remainder of the construct.
- Layout, whitespace and indentation are typically under control of the programmer. Again, this is unnecessary. Many modern programming environments will offer help in automatically indenting correctly, but the indentation can still be "broken" (i.e. changed to contravene coding styles) by the programmer after the fact. This happens easily and often by accident. Since indentation and layout rules exist and follow simple algorithms, there is no reason why this cannot be fully automated, freeing the programmer from one more unproductive task.

The reason that these tasks require more work than would be ideally desirable is rooted in the fact that program representation is based on pure text. A one-dimensional sequence of free-form characters is arranged two-dimensionally on screen, and this serves as the basis for all program elements.

Pure text representation is a technology developed more than half a century ago for early computer terminals, and there is little reason today – other than historical inertia – to restrict program representation to this limited form. (This observation previously led to work on structure editors; later, in section 13, we will examine why these early attempts failed but block-based editors later succeeded.)

Few reasons exist that program statements have to be typed out manually in their entirety, that the programmer should be responsible for correct punctuation, or that characters (tabs and spaces) should be used for the arrangement of program components on a page. Some work has also suggested that the choice of syntax symbols is often arbitrary, and that a randomly chosen syntax is no less usable than existing ones [21].

Scope is represented in many programming languages by using a pair of brackets. This is undesirable for several reasons:

- A pair of brackets is not the best visual representation of the extent of a scope. Considering graphical elements as a possible part of a language, using drawn frames, boxes or colours offers a clearer, continuous representation which is easier to recognise and interpret than two isolated brackets.
- The fact that one can even omit a closing bracket – that it is technically possible to enter *half a statement* – serves no useful purpose. Any modern system should ensure that a statement is either *present* or *absent*, and offer interaction techniques that allow convenient entry and manipulation. Representing program statements as a sequence of characters which can all be edited individually – for example deleting a single character out of the middle of a language keyword – is an archaic accidental artefact that is hard to justify in today's code editors.

Modern IDEs have largely recognised this fact and offer a variety of support mechanisms to address some of these issues. Shortcuts, code completion, auto-indentation and automatic entry of matching brackets and quotes are all designed to alleviate the unnecessary busy work a programmer is tasked with. However, these mechanisms fail to solve the real problem. While they streamline the entry of the program text, the representation is still pure text, with all the resulting problems. Spelling can still be incorrect, parentheses can be deleted after entry to break the balance, indentation can be changed accidentally, and so on. Various possible benefits and improvements cannot be realised due to the reliance on pure text for program representation. In the following parts of the paper, we shall discuss some improvements that become possible when leaving behind pure text as the medium.

4. Blocks versus Text: A Brief Comparison

4.1. Criteria

In the previous sections, we have discussed some beneficial and some problematic aspects of direct manipulation (block-based) and text-based programming environments.

Three aspects emerge as the main areas of consideration:

- *Representation* includes the appearance of the program at various scales, from the visual appearance of a single instruction to the representation of larger structures such as control structures, classes, or modules. Representation is crucial for program comprehension and readability.
- *Manipulation* describes all aspects of program entry and editing, including ease of entry and deletion of program constructs, making changes ranging from small scale edits to large refactorings, and extending existing program source.
- *Error rate* refers to the rate of errors an average programmer makes, or the number of errors that can be made in a system.

As we have discussed, a significant number of syntactical or type errors can be avoided in some systems.

4.2. Comparison

Using these three areas of consideration, which class of system – block-based or text-based – is better? This question cannot be answered without taking the type of user into account, so we will ask this question individually for two relevant distinct user groups: Beginners and proficient programmers.

4.2.1. Novice Programmers

For novice programmers, block-based environments have a lot to offer:

- They provide a clearer, easier to interpret *representation* of individual program statements and their semantics;
- They allow easier *manipulation* of program elements, to a large extent because of the recognition-over-recall characteristic of entering program code; and
- They lead to a significantly lower *error rate*, eliminating many syntax errors outright.

For a typical thirteen-year-old novice, block-based systems win on all counts – a finding confirmed by several recent blocks versus text comparison studies [22, 23].

4.2.2. Proficient Programmers

For proficient programmers, text-based systems have some distinct advantages:

- Once a reader has been trained to read a programming language, text provides a more concise, more readable *representation* than blocks;
- *Manipulation* in standard text editors is faster and more flexible than in block-based systems – viscosity is significantly lower; but
- Text-based environments still allow a higher *error rate*, and even proficient programmers will make some errors which would not be possible in direct manipulation systems. Many of these errors will be slips and typographical errors, which are quickly fixed by experts; however, they still have the potential to interrupt workflow and cognitive processes.

Overall, for proficient programmers, typical text-based environments are clearly preferable.

4.3. Where is the Cut-off?

In the above discussion, we have – rather arbitrarily – distinguished only two groups: "novices" and "proficient programmers". This raises the questions: What about intermediate users? And at what point do programmers become sufficiently "proficient" to warrant a shift to text-based systems?

In fact, programmers reach the point where their proficiency outstrips the usability of typical block-based languages fairly quickly. We believe that a typical sixteen year-old, having programmed for two or three years with Scratch or similar systems, will normally have reached a level of expertise and expectation where she is more efficient and productive with a typical text-based system. For adults, with their higher ability of dealing with abstraction and notation, the time of usefulness of block-based systems is much shorter still (and may be near zero for some novices with good technical and abstraction background).

Despite their clear and distinct advantages in early stages of learning to program, current block-based languages do not manage to support programming activity for a significant length of time beyond the initial learning stages. Programmers outgrow these kinds of system fairly quickly, and their advantages are lost with the switch to traditional text-based systems. (We have discussed elsewhere the issues surrounding the transition from blocks to text programming in more detail [24]).

One could present the view that, at the time of sufficient maturity that a change to a text-based system is advisable, the additional help provided by direct manipulation systems is not needed anymore, and thus there is no problem. This, however, misses an opportunity. We strongly believe that proficient programmers can also profit from the advantages that block-based systems bring to the table: clearer representation, easier manipulation, and lower rates of errors. Programming for *all* users can be improved if the advantages of both kinds of systems can be combined.

5. Frames: A New Editing Paradigm

In the remainder of this paper, we present the concept of *frame-based editing*, a redesign of program editing with the goal of combining the advantages of text-based and direct manipulation editing systems.

Block-based languages provide the following main advantages:

- They make many syntax errors impossible, and thus reduce error rates.
- They make program statements visible and support recognition and experimentation.
- Some selected editing operations are quicker or easier due to the direct manipulation characteristics of the program elements.

Text-based systems have the following strengths:

- The representation is more readable for anyone but early novices.
- Lower viscosity; program manipulation is quicker.
- Navigation and exploration are more flexible.
- Programs can be entered, manipulated and navigated purely via the keyboard.

The goal is to combine the best of both worlds, to create an editing technology that sits in the space between block-based and text-based languages and that combines advantages of both systems.

The principle providing the foundation of this design is to approach the problem from an HCI perspective. Two views are fundamental:

- A programming environment is a user interface for manipulating a program.

- A programming environment is a user interface for understanding a program.

Reading and writing programs is equally important, and therefore both the representation and the manipulation aspects need to be considered equally.

When designing an interface for the representation and manipulation of programs, some elements are better represented graphically. Scope, for example, is a concept of extent in the program text, and can better be presented with graphical elements (such as boxes) than by using characters (such as brackets) in the text flow of the program.

6. Design of a Frame-Based Editor

Figure 3, overleaf, shows the interface of a frame-based editor for a new, Java-like language called Stride[1], integrated into the Greenfoot system since 2015 (and BlueJ since 2017). The editor uses some graphical elements (shapes and colours) to present aspects where graphics have advantages over characters. Overall, however, the presentation maintains the look of a program as essentially a textual, if coloured, document.

Greenfoot [25], the system our current implementation was first integrated into, is an introductory development environment aimed at beginning programmers. Previously, when it supported only the Java programming language, it was targeted at those aged from about 14 years old upwards. The new version, also supporting Stride, is aimed at an audience starting younger than that by two or three years. We will use this implementation as the prototype to discuss the concepts of frame-based editing.

While some specific design decisions are influenced by the concrete context (a novice user group likely to transition to Java), most of the aspects described here are independent of this context, and the design advantages would apply equally to professional environments. We will discuss this applicability to professional environments further in section 14.

6.1. Representation

Figure 4 shows the look of a segment of typical program code in Stride. We will discuss several aspects of our code representation, in turn.

6.1.1. Scope

Scopes are represented as frames: graphical boxes, rather than the customary pair of brackets or keywords. This is true for all scopes: classes, methods and control structures. The frames – like the scopes – may be nested.

The advantages are fairly obvious: Recognising the extent – beginning and end – of a scope is much easier and quicker in this representation. Programmers do not need to determine which closing bracket matches which opening bracket, and no additional confusion can be created by misleading indentation.

[1]The exact differences between Java and Stride are detailed in the appendix.

6.1.2. Indentation

In text programming, indentation is created using editable whitespace characters (tabs or spaces; the subject of a long-running debate, which also includes the exact number of spaces to be used), and programmers are responsible for creating and maintaining correct indentation. Both of these are archaic characteristics that have no place in modern editors. We will see that this fact – that all program elements are represented by text characters – forms the basis of problems with many elements of current systems.

Making programmers responsible for maintaining correct indentation – a task that can easily be automated – adds unnecessary work, both manual and cognitive, and may cause distraction from the actual task the user wants to achieve. The fact that almost all modern programming environments provide substantial help with this, in the form of auto-indentation of newly added lines and auto-formatters for whole documents, shows that designers of text editors are aware of the problem. But there is no reason to allow indentation to be later modified to be incorrect, or to require programmers to think about it at all.

Making the editor responsible for indentation also opens up a solution to another problem. Many programmers disagree over the desired depth of indent; anywhere from two to eight spaces is used. Frame-based code does not store the indent depth in the file – it is simply a graphical attribute in the editor[2]. Thus, it becomes possible for each programmer to define their own preferred indent level as a personal display preference without altering the shared code. And because our editor also manages the line-wrapping of code as a visual aspect (without modifying code to add line breaks, etc), the indentation of continuation lines is also automated.

One partial consequence of managing the indentation as an editor attribute is that we are free to use variable-width fonts. The main advantage of traditional monospace coding fonts is that they allow programmers to align code. This is no longer tied to font selection in frame-based editing. Some studies have suggested that variable-width fonts are more readable than fixed-width [26, 27], although these results are often hard to generalise as they are affected by specific choice of font face.

6.1.3. Whitespace

While indentation – leading horizontal whitespace – is maintained automatically in a frame-based editor, vertical whitespace – blank lines in traditional text editors – is partly automatic and partly under control of the programmer.

Spacing between fixed elements of a program (for example, the space between method declarations) is maintained by the system. There is little need for this to vary, so consistency can automatically be maintained.

Within a sequence of statements, vertical whitespace is sometimes used to separate logically distinct parts of a method.

[2]A similar argument is made for tabs over spaces, but continuation lines usually present a further problem.

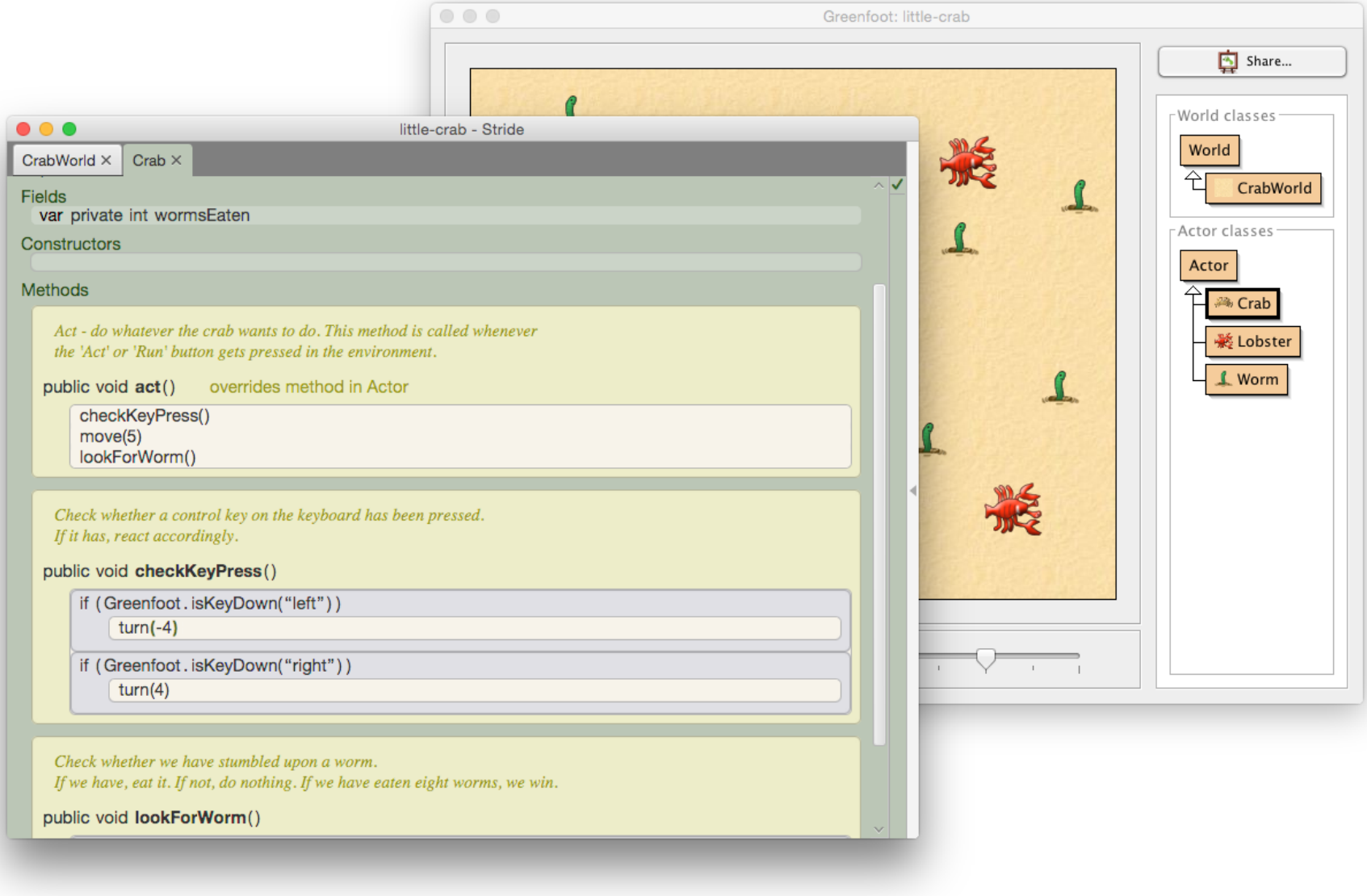


Figure 3. Frame-based editor interface, integrated into the Greenfoot system

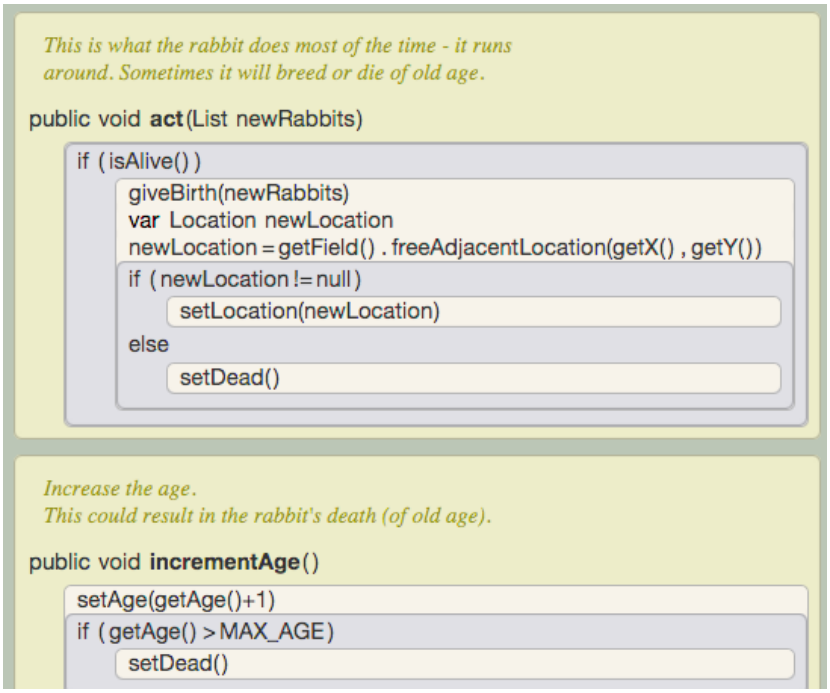


Figure 4. Code representation in a frame-based editor

This is a semantic consideration – not a syntactic one – and a programmer, therefore, has the option to enter vertical whitespace between statements, equivalent to inserting a blank line in a text editor.

6.1.4. Colour

Background colour is used to identify different frames, which represent different kinds of program elements. In our implementation, the outermost frame – the class – has a green background, methods are yellow, with other types of frames using various colours to distinguish themselves.

Simple statements, such as assignments or method calls, are also represented by frames (although these do not hold nested statements). These simple frames have a greyish-sandy background colour and no border drawn around them. This makes sequences of statements visually less busy than in most block-based editors.

Many frames have "slots" – holes that need to be filled in to complete the statement – such as the condition in an if-statement or the value in an assignment. These slots are white when they are empty. When slots are correctly filled in, they acquire the background colour of the frame, blending into their context. Thus white areas, standing out quite clearly, signify syntactically unfinished code (visible later on in Figure 7 and Figure 8).

Users can get used to these colours quite quickly, and they provide useful cues about program structure that are quicker and easier to recognise than groupings arranged using bracket characters.

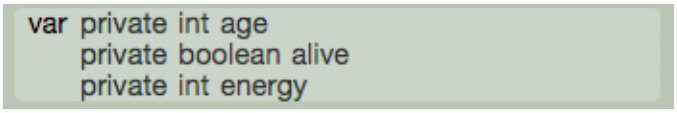


Figure 5. A group of variable declarations

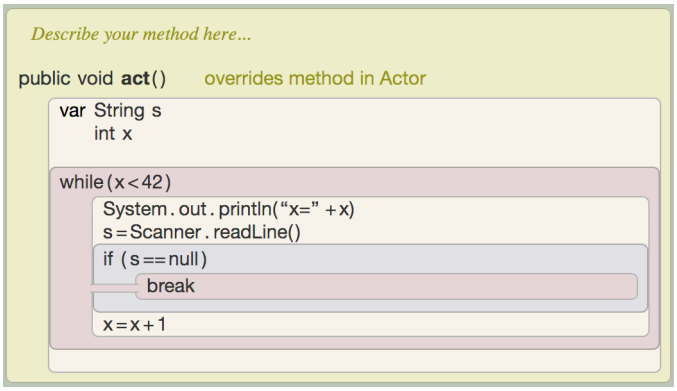


Figure 6. Representation of a break statement

6.1.5. Context Sensitive Display

The visual representation of statements can be context sensitive. For example, a variable declaration starts with the keyword "var" (Figure 5). However, if a variable is declared directly below another variable declaration, the keyword is not repeated; for visual simplicity, the keyword is shown only once for a variable group. The indentation of the rest of the frame is kept constant, to indicate a grouping of variable declarations.

Another context-adaptive example is the presentation of

Figure 7. An if-statement with empty slots

break statements, which exit the innermost loop or switch statement in which they are contained. The background colour of the break statement automatically matches the innermost enclosing loop or switch statement, representing its context, and a solid band of colour is drawn from that container's indent to the break statement (Figure 6).

This reflects again the underlying principle that the programmer is responsible for creating the structure of the program, but not for creating or maintaining its visual representation.

6.2. Manipulation

Manipulation of programs takes place mostly at the frame level. Users enter, remove or manipulate frames as a whole. Since frames represent statements (and other program elements) the main unit of manipulation is complete statements, not single characters. Character level editing exists only in text slots within frames (see section 6.2.2).

6.2.1. Insertion of Statements

Statements are inserted by inserting a frame. Every kind of statement has its corresponding frame, which can be inserted using a single command key when a frame cursor has focus (see section 6.2.3 and section 7.5).

Command keys are simple character keys on the keyboard – they do not need to be combined with a modifier key. Thus pressing the 'i' key when the frame cursor is focused will enter an if-statement (not the character 'i'). This is different from auto-completion of statements as supported in many traditional environments. There is no need to trigger any code completion system; one keypress is all that is needed. The command keys are not necessarily the first characters of a keyword. (Assignment, for example, can be inserted by typing an equals symbol.)

6.2.2. Slots

Some frames are complete just by inserting the frame itself (such as a *break* statement). However, most frames require additional information to be filled in to be complete; this information is provided in *slots*.

Frames can contain two different kinds of slots: *text slots* and *frame slots*. Text slots accept (almost) free-form text entry, whereas frame slots contain nested frames. A frame for an if-statement, for example, has two initially empty slots: a text

Figure 8. Optional text slots: invisible without focus (left) and visible when holding keyboard focus (right)

Figure 9. Cursors: a frame cursor (left) and a text cursor (right)

slot to specify the condition and a frame slot to hold the body of the statement (Figure 7).

Text slots have a white background when they are empty, expecting text entry. Two varieties of text slot exist: *compulsory* and *optional* text slots.

Compulsory text slots are always visible, and content must be supplied to create a syntactically valid program. The condition of an if-statement is an example.

Optional slots are only visible when the cursor navigates to their potential location. An example is a formal parameter in a method declaration. The parameter list always has optional slots at the end so that additional parameters may be entered. When the cursor is not at the location of the optional slot, it is invisible (Figure 8, left); however, when the cursor is moved to the location of the optional slot it becomes visible, gains focus and text can be entered (Figure 8, right). In the case of the formal parameter, two optional slots are present, one for the type and one for the parameter name. When one is filled in, the other slot becomes compulsory.

6.2.3. Frame Cursor versus Text Cursor

A focused frame editor always displays one cursor, and the cursor is always in a slot. Two different types of cursor exist, depending on what kind of slot has focus: When the cursor is in a frame slot, a frame cursor is shown (Figure 9, left); inside a text slot, the cursor changes to a text cursor (Figure 9, right). It is not possible to have a frame cursor and a text cursor at the same time.

Interpretation of input differs with the two different cursors: When the frame cursor is visible, key input is interpreted as commands, and corresponding frames are inserted. When the text cursor is visible, keys insert their own character literally, as in a traditional text editor.

Technically, this introduces two separate modes: a frame editing mode and a text-editing mode. These modes are entered by cursor movement, and visually distinguished by a different cursor representation. Whether this causes confusion to users was one of the important early questions in this design, and is discussed further below.

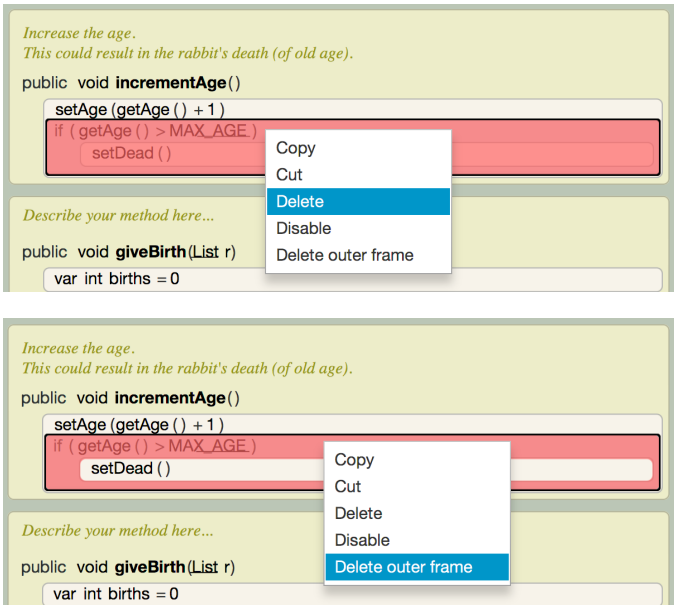


Figure 10. Preview of delete operations

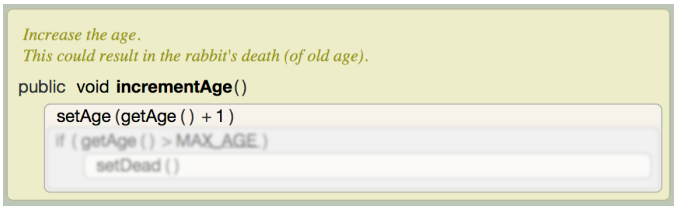


Figure 11. A disabled frame

Statements are frames, and entered in frame slots, while expressions are not frames, and are entered in text slots.

6.2.4. Deletion of Statements

As with insertion, deletion of a frame deletes the whole statement. Deletion can be achieved using the delete or backspace keys when the frame cursor is before or after the frame, respectively.

Another option to delete a frame is to right-click the frame with the mouse, and selecting a delete option from the frame's pop-up menu. Different delete options may be available: If, for example, the frame is an if-statement, the statement can be deleted while leaving the statements contained in its body present, or the statements in the body may be deleted with it. While a function is selected in the menu, a preview annotation in the source code hints at the effect of the selected function (Figure 10).

6.2.5. Disabling Frames

The example above, selecting 'Delete' from a frame's context menu, shows another advantage of elevating program elements to first class citizens in the interface. Since declarations and statements have become interface entities in their own right, they can have associated properties and functionality. Most obviously, they can have a context menu which offers operations on the frame. One of these operations is *disabling* a frame.

Disabling a frame (Figure 11) temporarily treats the frame as if it were deleted. In traditional systems, this is typically done by "commenting out" a block of text. (One study [28] found that 63% of all comment usages were disabling code, not providing an actual text comment.) Again, we see the richer possibilities of an interface not relying solely on characters for functionality: "Commenting out" a sequence of statements to temporarily disable them is technically a misuse of the comment symbol – unused code is not a comment. The purpose here is not to comment on other code, and a comment symbol is used merely because of the absence of other mechanisms.

If a comment symbol is entered at each line, entry and removal can be tedious, and individual lines can be missed. If a block comment is used, they typically cannot nest and so consideration must be given to whether the commented section already contains a block comment.

Existing text-based environments can alleviate the issue of the display by giving comments a different appearance (although there is no distinction between actual prose comments and disabled code). Our frame editor gives disabled frames a lightly blurred appearance (Blockly has similar functionality and changes the background appearance). Unlike in traditional editors, it is not possible to comment out part of a statement, for instance missing a closing bracket of a scope, and thus breaking program structure.

The explicit disable function in the frame editor is both visually clearer and less prone to syntax errors. A disabled frame can be re-enabled by the user when required. All frames inside a disabled frame are always disabled; you cannot re-enable a frame inside a disabled block (we believe this would make the program flow too hard to follow). However, inner frames can be easily dragged out and re-enabled.

6.2.6. Selection and Clipboard Operations

One theme in frame-based editing is the idea that most operations should act on whole frames (structures), and not parts of frames. This also applies to selection.

Selection of frames always selects a whole frame, or multiple adjacent frames inside the same enclosing frame slot. If the user selects a loop frame, for example, all of its content is automatically included in the selection. As a consequence, a frame selection must always begin and end within the same frame slot (scope).

Multiple selected frames can be dragged simultaneously as described in the next section, or the usual cut and copy operations can be performed. Frames may be pasted at a frame cursor location. So cut/copy/paste may be performed on single frames or multiple consecutive frames (at the same scope), and is also possible on the text content of individual slots.

When one or more frames are selected, inserting a control structure frame wraps the selection in that frame; the selection becomes the body of the inserted structure. This provides an

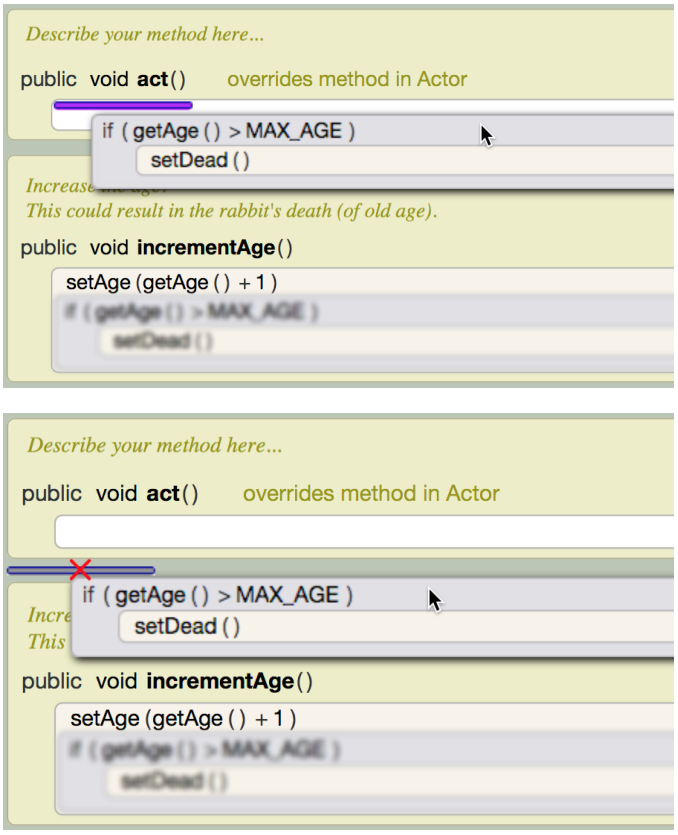


Figure 12. Cursor indicates valid (top) and invalid (bottom) drop targets. The drag source (at bottom of each) is blurred during drag.

easy way to add a loop or if-guard to existing code, as would be done in text by adding a header at the beginning, then adding a closing curly bracket at the end, then correcting the indent.

6.2.7. Drag and Drop

Treating statements and declarations as interface entities also naturally leads to the expectation of being able to perform drag-and-drop operations on them. Frames can be dragged with the mouse and dropped at alternative locations.

In traditional text editors, similar functionality is usually available: Text can be selected, and the selected text can be dragged and dropped to a different location.

Again, the different unit of manipulation (editing frames instead of editing characters) leads to various advantages in our editor compared to text:

- In text editors, arbitrary spans of text can be selected and dragged. These may include parts of statements, accidentally selected, and thus the drag operation may invalidate program structure. In the frame editor, only complete frames can be dragged. (This includes simple one-line statements – these are also frames.)
- Selecting a complete multi-line statement in a text editor typically requires careful targeting with the mouse, making this a high-overhead operation. It usually requires careful consideration of including whitespace, indentation or trailing return characters in the selection, resulting in different formatting for subtly different choices. No such consideration and fine targeting are required in a frame editor; the frame is a large target, and does not require selection before dragging.
- In text editors, dragged text may be dropped anywhere, again potentially breaking program structure. The vast majority of potential drop locations are syntactically invalid, yet no help is provided by the editor in identifying the few valid ones. In the frame editor, frames may be dropped only at locations where they maintain a syntactically correct structure. While frames are being dragged, the cursor indicates whether a potential target is valid or not (Figure 12).

It makes little sense to allow, for example, placing statements outside of a method body or method declarations at locations where they are invalid. Since many more potential edits invalidate legal structure than maintain it, disallowing invalid manipulation severely cuts down the space of possible user actions. Fewer possible actions lead to simpler and more expressive interfaces and fewer mistakes.

Dragging and dropping in frame-based editing also has advantages over block-based editing. In most block-based editors, it is awkward to drag out individual blocks or two adjacent blocks from a larger body. Generally, you must either drag the blocks separately, or (in the case of Scratch and many others), drag out the body, pick it apart with a further drag or two, then drag back the pieces you wanted to keep in place. Allowing a frame selection followed by a drag provides a much easier and clearer way to perform such a manipulation.

6.2.8. Changing Frame Type

One manipulation that is often highlighted as difficult in block-based or structure editors is that of changing the type of frame. A common example is changing while loops into if statements, and vice versa. There is also the issue of changing, for example, method calls into assignments if you wish to store the result (either because it was a mistake to discard it, or because you now need the result[3]), or vice versa if you decide not to. We have different mechanisms for these manipulations, which we will consider in turn.

We resolve the issue of changing method calls into assignments by direct text editing. If the user types an assignment symbol (the '=' character) in a method call frame then the frame is automatically converted into an assignment frame. Similarly, deleting the assignment symbol from an assignment frame (by placing the cursor before and pressing delete, or after and pressing backspace) converts in the opposite direction. This means existing text-manipulation patterns transfer directly. This does present a question as to whether it is worth conceiving of method call and assignment frames as different

[3]Many Java methods have a side-effect and return a value which may or may not be of interest. For example, the remove method on lists in Java removes the element from the list if present, and returns a boolean indicating whether the item was present or not. Depending on your code, you may or may not care whether it was present or not beforehand.

at all. We believe it is pedagogically useful to consider assignment different from a plain method call, even if technically in the editor, there is minimal difference.

The issue of transforming while to if statements is handled differently, because we do not allow editing of keywords, and thus text-manipulation cannot be used to make the change. Initially in the Stride editor, we were uncertain whether it was worth providing this transformation. Our intuition was that this transformation was rarely needed. However, we have access to a dataset to verify this. The Blackbox data set [29] records the editing behaviour of several million sessions of BlueJ, our beginners' Java IDE.

We looked in the Blackbox data set for the number of edits (and session counts) which introduced a new line of code containing an if statement, a new line of code containing a while statement, or changing if to while (and vice versa) without changing the accompanying condition[4]. We found that (to 3 significant figures):

- In total, there were 732,000,000 edits across 9,160,000 sessions.
- 2,030,000 edits (across 1,030,000 sessions) introduced a new line with an if statement header
- 46,900 edits (across 35,200 sessions) changed an if statement header to a while
- 376,000 edits (across 277,000 sessions) introduced a newline with a while loop header
- 40,600 edits (across 29,400 sessions) changed a while loop header to an if

This means that over 10% of the while loops written later get changed to if statements (minus those which are changed back and forth between while and if, something which is difficult to track). This was a much larger proportion that we had anticipated, and on this basis we introduced a context menu option to change an if statement to a while loop, and vice versa.

6.2.9. Localised History

Our frame-based editor has a standard, class-wide undo system: The standard Ctrl-Z shortcut undoes the last edit, whether it altered the value of a text slot or moved, added or deleted a frame.

A previous study of programmer behaviour, however, found that programmers rarely used the undo feature [28]. Programmers observed in this study preferred backspace for correcting recent typing mistakes. They were often unable to use undo for their other corrections because they had already made subsequent correct edits elsewhere before noticing the error; undo would have removed these first.

To solve this problem, we offer localised history. Each slot keeps a history of the three most recent content values. A submenu in the context menu offers these values for selection to restore an earlier state. This mechanism provides independent undo functions for logical elements of the code, even if other edits have been made elsewhere. Apart from providing more flexible undo, this function also supports easy experimentation: Values may be changed temporarily with the ability to restore previous values easily. Like several features in our frame-based editor, this feature is difficult to implement well in text-based editors [30], but is straightforward in frame-based editing.

6.2.10. Extending Frames

Many frames, such as while loops, have a fixed structure. Other frames, however, can be extended: An if-statement frame, for example, begins without an "else" clause, but can be extended to add it. Another example is the extension of a constructor definition to add a "super" or "this" call to invoke another constructor. Frame extension is triggered by command keys, just as frame insertion. For example, to add an "else" clause to an if-frame, the user should place the frame cursor inside an if-frame, and press the 'e' key; any frames after the cursor position will be used as the body of the new else clause. Adding "else if" clauses is done in a similar way, with the 'l' key. Deleting such extensions is done by placing the frame cursor at the top of the following frame slot and hitting backspace.

6.3. Navigation

The frame cursor allows navigation and manipulation to be performed with the keyboard, which is the preferred input method for navigation for practised programmers [31].

The frame cursor is always positioned between frames. Up and down navigation moves in steps of single lines by default (mimicking traditional editors). However, combining the use of cursor keys with a modifier key moves the cursor at the current scope level, jumping over compound frames in a single step. This adds a generic method of quick movement; if the cursor is outside of a method, for example, this command moves in increments of whole method definitions and provides quick navigation through a class.

The left and right cursor keys enter a slot in the neighbouring frame, whether it is a text slot or a frame slot, positioning the cursor at the beginning or end of the slot: The left key goes to the end of the last slot in the previous frame, while the right key goes to the beginning of the first slot in the next frame. The TAB and Shift-TAB keys can also be used to navigate slots.

Overall, this key binding largely mimics movement in traditional text editors, but adds generic navigation options for additional structure-based movement.

6.4. Overtyping

Most syntactic elements, such as parentheses, commas and spaces, are automatically displayed annotations and decorations; they do not need to be typed and cannot be edited. A new frame is created with these decorations, and only slots need to be filled in (Figure 13). Using a TAB character or right arrow advances to the next slot.

While this is a common interaction sequence for form fill-in, it goes against the habits of programmers, who are used to

[4]This stipulation avoids falsely counting cases where the user has happened to paste completely different line(s) of code over the previous code.

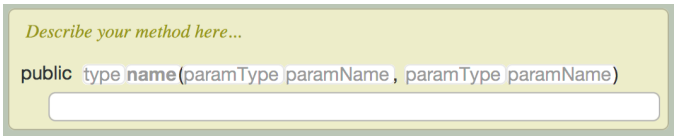


Figure 13. A method frame with empty slots

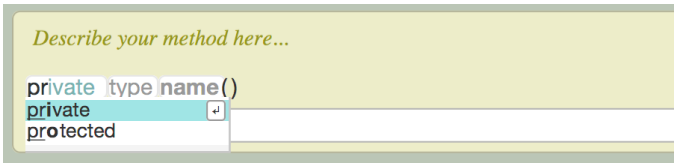


Figure 14. Text entry in a choice slot

typing the spaces and other syntactic elements. In the Stride editor, typing the syntax elements is permitted and advances the cursor to the next slot. The programmer can effectively "type over" a syntax element. For example, when entering the method return type in Figure 13, typing a space moves the cursor to the method name field. From there, typing an open parenthesis advances the cursor to the parameter list. Pressing space again moves from parameter type to parameter name. A comma in the parameter name slot creates the additional parameter slots and places the cursor into them.

As a result, an input key sequence that would type the method signature in a traditional text editor also works identically in the Stride editor. This supports the muscle memory of experienced programmers, who do not need to relearn all habits of program entry.

6.5. Editing in Text Slots

When filled in correctly, text slots appear as normal text on the background colour of their enclosing frame (see, for example, Figure 4). This ensures that lines of code can be read as flowing text without unnecessary visual overhead. Text slots that are either empty or have keyboard focus have a white background.

The Stride editor has three types of text slots: *identifier slots* that support entry of an identifier of the supported language, *choice slots* that allow entry of a limited set of fixed values and *expression slots* for the entry of expressions. Using specialised slots for different kinds of token in the syntax tree allows support for more efficient content entry as well as better avoidance and reporting of errors. We will give more detail on each type of text slot in turn.

6.5.1. Identifier Slots

Identifier slots allow the textual entry of program identifiers, but inhibit entry of characters that are syntactically invalid in this context, such as punctuation characters. Typing characters that follow the identifier slot, such as a space or the opening parenthesis after a method name, causes slot advance by overtyping as described above. Special characters can also cause the insertion of optional slots; typing, for example, a comma in a formal parameter name inserts an additional pair of slots for entry of an additional parameter.

6.5.2. Choice Slots

Choice slots allow selection of one of a small number of possible values. They are used, for example, for the access modifiers of method declarations (Figure 14) which have three possible choices.

Choice slots behave in ways similar to combo boxes in many interface toolkits: One value is always selected (they are never empty), and choices can be made using the mouse or cursor keys. Textual entry, however, is also possible. When the slot gains keyboard focus, a menu of choices is displayed, with the top choice selected. Typing narrows the choice list to those values starting with the characters entered (invalid characters are ignored), and using TAB, return, space, or a right arrow at any point confirms the current selection and advances focus to the next slot.

This method again ensures syntactically valid program structure while allowing overtyping of the whole text if desired, with the added possibility of much faster entry.

6.5.3. Expression Slots

Expression slots allow the entry of expressions, including arithmetic expressions, variables and function calls.

Expressions are structured. When an operator is entered in an expression slot (such as a plus symbol), the expression acquires a new structure, with an operator in the middle between two text fields for each of the operands. The operator itself is not part of the text fields; it can be deleted, merging the text fields again, but cannot be edited.

Multi-character operators are easily inserted. For example, typing "<" (the less than symbol) splits the current text field into two, with the less than operator between them. Typing "=" (the equals symbol) at the beginning of the second field will automatically merge with the less-than operator to make less-than-or-equal-to, just as would happen when typing text.

Entities that appear as pairs of symbols (such as parentheses, brackets and quotes) always appear in full: entering one half enters the other half automatically. While traditional text editors typically also do this for entry (e.g., typing an opening parenthesis automatically inserts the closing one), the link in the Stride editor is stronger: The pair of symbols remain linked and are processed during editing as a single operator. Deleting one also deletes the other, selection and drag-and-drop operations always operate on the complete subexpression, processing both or none of the bracket symbols. Existing expressions can be enclosed in parentheses by selecting the expression in question and pressing the opening parenthesis. This will enter both surrounding symbols.

Again, as before, various advantages flow from the fact that the user edits the structure of the code, not the representation:

- Some syntax errors become impossible to make, cutting the overall error rate.

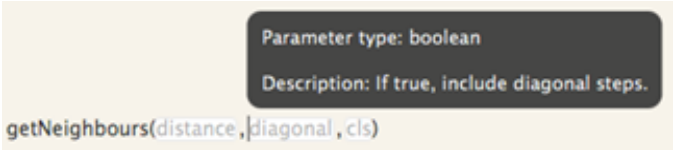


Figure 15. Prompt text and tooltip for parameters

- Edit operations, such as selection and drag-and-drop can guarantee to maintain syntactically valid structure.
- Errors that do occur can be associated more precisely with a particular token in the program source, because there are no structural syntax errors such as missing brackets.
- While structure is under the control of the programmer, representation can be automated. For example, spacing between symbols and arranging line breaks and indentation in long lines can be automated to be consistent and meaningful which can enhance readability.

Automatic adaptive spacing is used in the Stride editor to clarify precedence of operators (higher precedence operators use less adjacent space than lower precedence operators). This technique has previously been used in mathematical editors [32] and in print-outs of code [33] but not usually in WYSIWYG programing editors. It can lead to respacing of an expression as it is entered, which is not ideal but we believe is outweighed by the ensuing readability advantage.

All Java expressions can be entered in Stride's structured expression editor using exactly the same keypresses (although almost all the space characters are redundant due to the automatic spacing), meaning that those used to text will notice no difference on entering text, just in the display.

Again, through the use of overtyping and flexibility in navigation, text entry feels natural to seasoned programmers, with opportunities for faster entry when becoming familiar with the frame editor.

Our expressions are displayed and entered infix. We believe it is important that the entry and display align. While we did not choose to use prefix notation in Stride, in prefix-expression languages like Lisp at least the entry and display are also aligned. Many of the early structured editors displayed infix expressions but required prefix entry. This required that the user understand the abstract syntax tree behind the expressions being created – but novices do not yet understand that an expression is a structured tree. Therefore we felt it was important to be able to enter expressions infix, to match how they are displayed in the editor.

6.5.4. Prompts and Hints

One advantage of using interface elements other than pure text for the representation is the ability to display prompts to provide guidance for expected entry. We have seen examples in the condition of an if-statement (Figure 7) and the definition of a method signature (Figure 13). While these examples are mainly useful for beginners who have not yet memorised the syntax of those statements, there are other situations where these prompts remain helpful for experienced programmers. One such example is found in the actual parameter(s) of method calls (Figure 15). In this example, the prompt text shows the name of the formal parameter, and a tooltip may provide additional information, showing the type and parameter comment.

Many existing program editors also provide helpful content for actual parameters when entering method calls via an autocomplete mechanism. Some enter the formal parameter name (as in our prompts), while others guess at a possible intended value and enter the name of a nearby variable of matching type. None of these is ideal. In the first case, the resulting program text is almost certainly wrong, but appears to have been completed. In the second case, the program may compile and run, without leaving a hint that the programmer may not have considered and confirmed the default choice, potentially introducing semantic errors.

The problem, again, stems from the fact that the interface elements are just plain text: traditional text editors cannot display text to the user embedded in the program source that is not also part of the program, and therefore interpreted by the compiler. (Modern IDEs are starting to develop the use of prompts and pop-up windows for this purpose.)

In the Stride editor, using richer interface elements, we can display the prompt text to the user while still recognising the slot as unfilled, delivering helpful information and more accurate errors at the same time. For example, if you write a method in Java and forget the type or name of a parameter (e.g. "`public void setX(x) { }`"), you will get an unhelpful error "`<identifier> expected`". In Stride, either the type or name slot will be empty, showing a more helpful message such as "name cannot be empty".

7. Interface Elements

The Stride editor uses various additional interface elements to improve readability and provide additional information to a programmer.

7.1. Method Header Display

The signature of a method contains important information: its name, parameters, and the return type. This information is frequently useful while reading or editing any given method. However, if the method is longer than a few lines, the information often scrolls out of view. In the Stride editor, scrolling up leaves the header visible, sticking to the top of the window, and the body of the method appears to slide underneath it (Figure 16). Thus, the most important contextual information remains visible.

7.2. Long Scope Annotation

Another example where header information is useful is in the use of frames which contain conditionals in their definition, such as if-statements and loops. Figure 16 shows examples of these where the frame header has scrolled out of view; the header information is then displayed in the left margin of the frame, leaving it visible for the programmer.

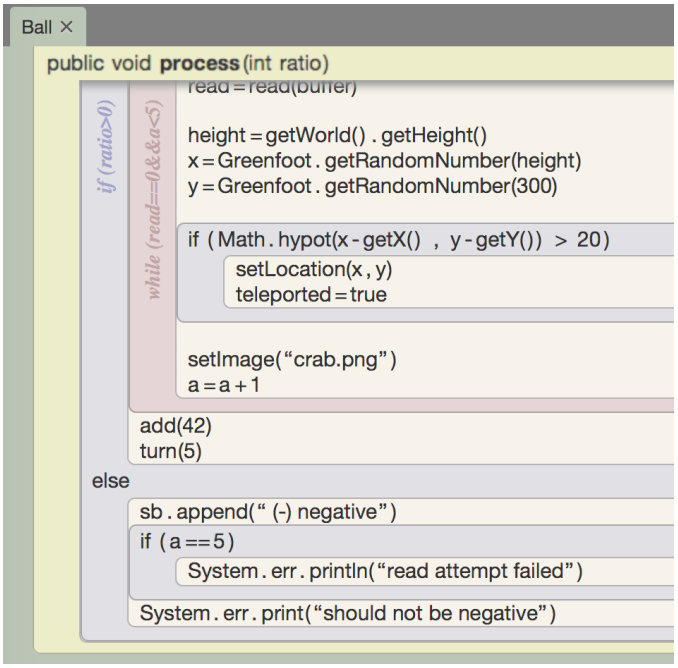


Figure 16. Method signature pinned to top of screen

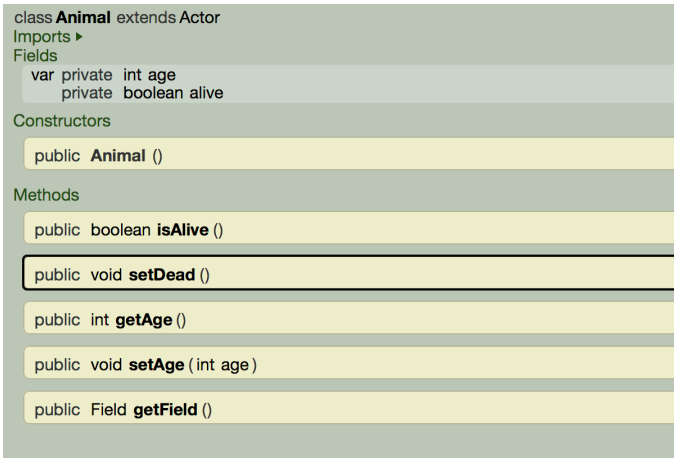


Figure 17. The bird's eye view

An alternative would have been to treat these frames identically to method frames and pin the header to the top of the display (or, indeed, to also use the left margin for method header displays). Pinning all open frame headers to the top was considered undesirable as it would have resulted in a potential stack of multiple headers, consuming more space and reducing readability.

The more significant context of the enclosing method, and the fact that there is only ever one enclosing method in Stride (which does not support Java's inner classes), justifies different treatment with more prominent presentation.

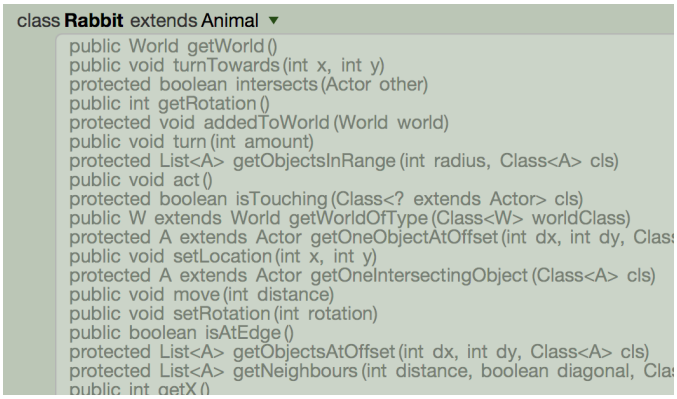


Figure 18. List of inherited methods

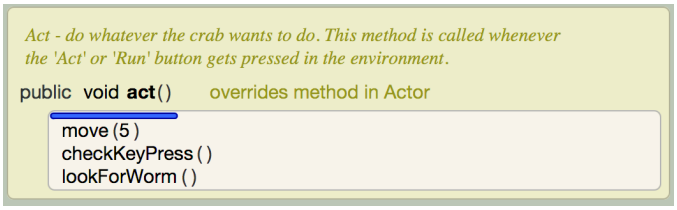


Figure 19. The override annotation

7.3. Bird's Eye View

The bird's eye view is an alternative (temporary) view of the class for quick orientation and navigation. Bound to a command key, it can easily be activated and displays only instance field, constructor and method frames, reduced to their signatures (Figure 17). A second press toggles the display or hiding of documentation (useful to see more details about methods, but reduces the conciseness of the method list) and a third press exits bird's eye view.

As a result, users can get an easy and quick overview over available methods, and easily navigate the class. Up- and down-arrows select method frames, and pressing the Return key or clicking with the mouse returns to the standard display with the selected method in view. (A similar view-only operation is available in Blockly to collapse all visible blocks, but without the functionality to easily navigate between the collapsed blocks.)

7.4. Inherited Methods

Inherited methods are available for invocation in subclasses. Traditionally, the available inherited methods are not easily visible without referring to documentation outside of the class under construction. This causes problems for learners who would use methods from the superclass without understanding where the methods were declared.

In the Stride editor, a small arrow next to the superclass name in the 'extends' declaration allows a list of inherited methods to be unfolded (Figure 18).

This list serves two purposes: It provides easily accessible in-editor documentation of the methods available, and it allows

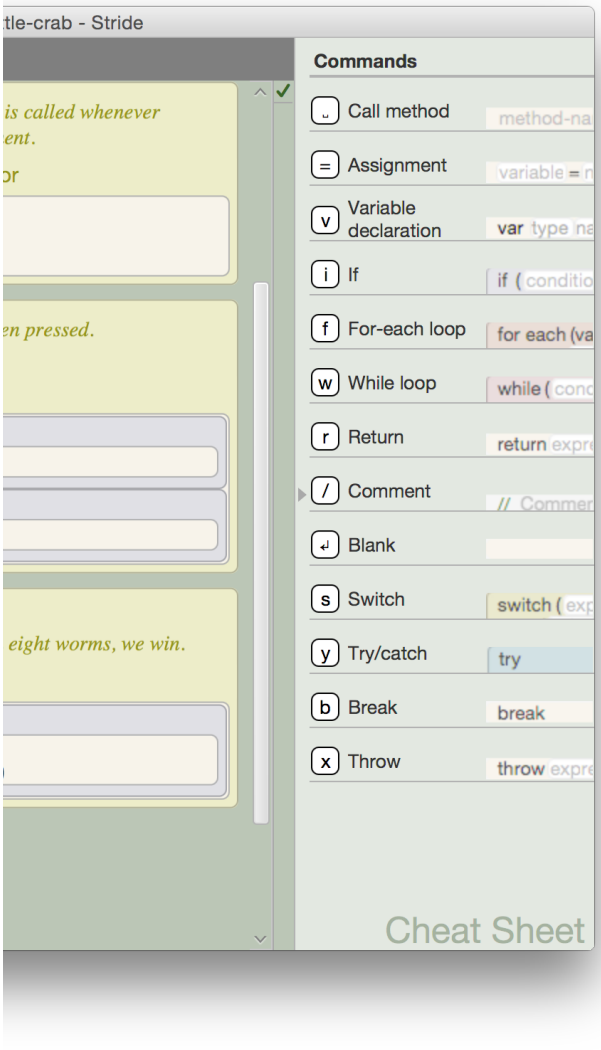


Figure 20. The "Cheat Sheet", which appears on the right of the editor

the user, via a command in a contextual menu, to override an inherited method. Selecting the override function on an inherited method inserts the appropriate method definition into the current class. It should stop the availability of methods seeming 'magic' to beginners, by allowing all available method calls to be seen in a class, either directly declared or shown in the inherited list.

Methods can also be overridden in the traditional way: by simply defining a method with a matching signature. In this case an override annotation is automatically added to the method definition display (Figure 19), serving as information to the reader. This contrasts with override annotations in Java, which the programmer must enter and manage themselves.

7.5. The Cheat Sheet

A small arrow on the right-hand side of the editor allows a separate window pane to be displayed, showing the "Cheat Sheet" (Figure 20). Viewing or hiding this pane is also possible via a shortcut key, and is shown by default for new users.

The Cheat Sheet lists all frames that can be inserted at the current cursor position, together with their command keys. Using the command key or clicking on the frame in the Cheat Sheet inserts the frame into the program text.

The Cheat Sheet serves a similar purpose as the block catalogue in block-based languages: It supports recognition over recall and encourages experimentation for users who are not familiar with the whole range of options. One significant difference is that block catalogues in most block-based languages show all available operations (method calls) separately, while Stride offers only different types of frames. Thus, a method call is shown as a single option, encompassing all possible method calls. Selecting a specific method from those available is left to the code completion mechanism, described below.

As a result of this choice, the number of available options at any point is limited and a complete set of options can be displayed in a relatively small amount of space. (Figure 20 shows the complete list of all available statement frame types that can be inserted at the a typical frame cursor position inside a method.)

The Cheat Sheet is context sensitive: At any time, only valid options are shown. If, for example, the cursor is between method definitions, only commands to insert methods or comments are offered. One potential disadvantage is that if a student wants to insert a frame not available at the current context (e.g. a method, but their cursor is already inside a method) then they will not find what they are looking for. However, we expect that this will only be an issue for beginners, and students will soon learn which elements are available in which context.

8. Context-Aware Editing

In a traditional text editor, the entire text area is one interface element. If the editor aims to offer context-sensitive support, the editor must infer the type of structure currently being edited from the representation. In a frame-based editor, each part of the code (i.e. each node in the syntax tree) is represented by a separate interface element. The programmer edits the structure, not the representation, so the context of the element edited is always known. As a result, frame-based editors can much more easily offer context-aware editing: we describe several examples of this here.

8.1. Contextual Code Completion

Code completion allows a programmer to insert code more quickly than typing it in full, by selecting "completions" from a generated list, usually chosen by matching the already typed prefix to possible suggestions (e.g. typing "get" would offer the completion "getNeighbours", Figure 21). Text-based editors often offer completion of variable and method names.

The Stride editor also offers completion of variable and method names – but only in expression slots. If the user is editing a type slot (e.g. a method return type or variable

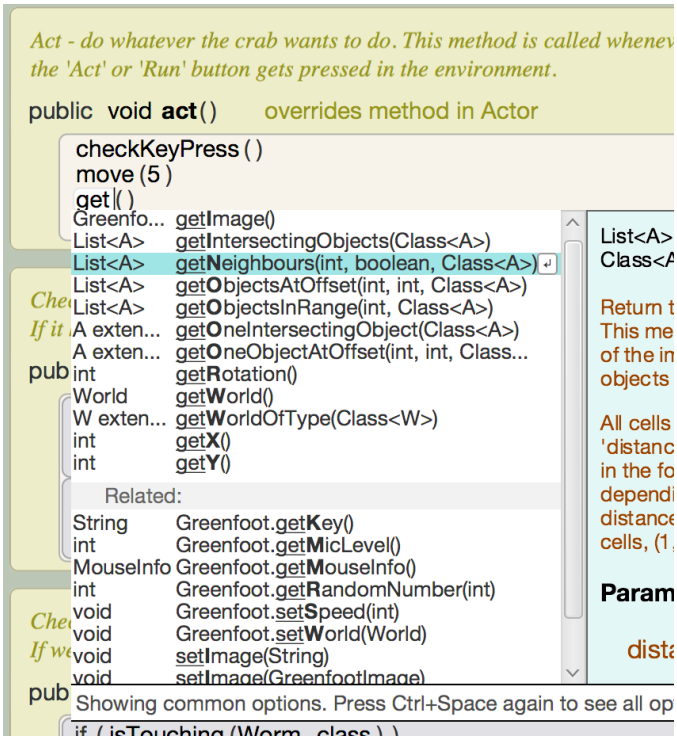


Figure 21. Code completion in Stride

declaration type), code completion offers type completions instead (e.g. "St" will offer "String" as a completion). If the cursor is within a string literal, code completion offers a list of frequently used string literals (in Greenfoot's case: image and sound filenames in the current Greenfoot scenario) as choices. If the user is editing a method name slot and requests code completion, names of methods in parent classes are offered for overriding. In a "throws" clause, only subtypes of type Throwable are offered, and so on.

In short: The fact that the Stride editor maintains code structure more explicitly enables better contextual support for code entry.

8.2. Error Messages

In a text-based editor, a single character (present or missing) can cause a parse error that affects the parsing of the rest of the file. Too many closing curly brackets or an unterminated string literal can affect the parsing of the remaining code. Thus, relevant syntax error messages can be hard to determine and can be affected by distant mistakes.

In a frame-based editor, the location of the source of a syntax error can be much more precisely determined. There is no possibility of unclosed scope, unterminated string literals or missing semicolons. Each syntax error can be attributed to a single slot in a frame.

For example, entering the text "wait(,,);" into Eclipse shows the error "wait cannot be resolved to a type", as the editor struggles to decide if this is intended to be a method call or a variable declaration. In a frame-based editor, this choice is made explicitly upfront, and entering "wait(,,)" into a method call frame will display two errors stating "parameter cannot be empty".

In fact, because so many invalid inputs are disallowed in frame-based editing, the majority of remaining *syntax errors* are simply "X cannot be blank", positioned exactly at the offending text slot.

8.3. Suggested Fixes

Some IDEs offer suggested fixes to a program when they detect an error. Our Stride editor does the same, and usually with less technical effort. In order for a text-based IDE to offer the fix "You are attempting to assign to undeclared variable x; fix by declaring here?", an IDE must parse the source, determine that this is an assignment statement, check whether the left-hand side is declared, offer this fix, and then manipulate the text to change to a variable declaration. In our frame-based editor, we implicitly know that an assignment frame is an assignment; we must still check if the left-hand side is undeclared, but then implementing the fix simply requires swapping the assignment frame for a variable declaration frame.

These examples demonstrate that offering improved help and support for programmers, while not impossible in text-based editors, is technically easier to implement in a frame-based editor. As a result, we are able, in some areas, to offer better functionality in the first release of a new editor than existing professional IDEs currently offer after many years of development.

8.4. Context-Awareness vs Error Tolerance

There is sometimes a tension between context awareness and error tolerance. As a simple example, when the user is entering content in a type slot, we could only allow entry of existing known types. However, it may be that the user intends to introduce a new type (and thus precluding its entry would be frustrating) or they want it to be incomplete (e.g. they have a List but need to look up its inner type). So we allow the erroneous type to be entered even though we know it to be wrong.

Another example is the context awareness of the frame cursor, where we make different choices in different contexts. We do not allow arbitrary code to be inserted outside methods: we know this to be wrong, and if the user wants to enter an arbitrary code fragment, they can always add it inside a nearby method. More subtle is the case of break frames: we know that they are only valid inside a loop or switch frame. But this means the context relies not only on the immediate parent, but the grandparent or further. Additionally, the user may write the break ahead of enclosing the current code in a loop. We thus allow break frames to be entered anywhere a statement can, and an error will be issued if this is an invalid placement.

9. Implementation

Frame-based editors must store program code and attribute information, such as the enabled/disabled state of frames. It

would be possible to store the frame structure as standard text (e.g. as Java code). An advantage then would be the interoperability on the same source files with other editors. We do not do this for several reasons.

Firstly, it requires extra technical effort to read and write the code from/to Java compared to storing it in a structured format. Secondly, if the file was externally edited it could be in an invalid state that is unrepresentable in our frame editor. Thirdly, any details such as white space in the Java code would be difficult or impossible to preserve during frame-based editing. Fourthly, Stride only supports a subset of Java, which makes it awkward if the Java code contains features unsupported by Stride (see appendix).

In short, storing the code as Java text would expose many of the problems that frame-based editing eliminates. Thus, the Stride editor stores the code in a simple structured form, using XML.

One historical advantage of text is that it was portable between tools. But this is not an inherent advantage of text-based programming: the advantage of being able to edit code in multiple editors, and for code written in Emacs to be analysable in Eclipse is not to do with text, it is because there is a standard format for Java code (defined by the Java language standard). There is no reason that Stride code stored as XML could not also be edited, analysed or compiled by other tools (including existing text editors, although this would be slightly unwieldy). Alternatively, Droplet [34] has shown that frame-based or block-based editors can use text as the canonical representation if desired.

We transform the program to Java source code for compilation, and then use a standard Java compiler. Any semantic errors are passed back into the editor for display to the user. (There should be no syntax errors generated from frame-based code, as syntax errors are detected during the Java generation phase and short-circuit the process.)

To support the transition of learners from Stride to Java, we also allow to preview Stride code as Java (using an animated transformation) directly in the Stride editor, and to convert the Stride code to Java for further editing in standard Java form in a traditional text editor. Greenfoot supports development in either Java or Stride, and classes in the two languages are interoperable. This transition is also behind the syntax choices of Stride: with a frame-based editor, the choices of keywords or syntax (e.g. brackets around the if-condition) are a choice solely based on visual design and comprehension rather than ease of entry. We retain similarity to Java to allow an easier transition from Stride to Java.

10. Evaluation A: CogTool

To evaluate the effectiveness of the design features of the Stride editor, we have carried out a study comparing task times of common editing tasks performed in a frame based editor and seven other commonly available and popular programming systems.

The editors used for this study were an earlier prototype of our Stride editor to represent frame-based editing, Scratch, Alice, and StarLogo TNG as examples of traditional block-based systems, the Lego Mindstorms NXT [35] editor as an example of a visual editor with an alternative design, and NetBeans (Java), Greenfoot (Java), and IDLE (Python) as representatives of common text-based editing.

The results of this evaluation have been presented in an earlier paper [36] and are summarised here.

10.1. Cognitive Modelling

To compare the editing tasks performed in each editor, we used CogTool [37], a software tool that automates the creation of cognitive models for the purpose of evaluating and comparing interactions in competing systems. Cognitive models extend on keystroke-level interaction models by including "mental" operators, such as eye movement, reading time and thinking time in addition to explicit interaction events (such as key presses, mouse movements and mouse clicks). It should be noted that CogTool is primarily intended for measuring expert use, and aspects such as novice learning processes are not addressed.

Cognitive models more accurately reflect interaction complexity and time than keystroke-level models, but are more difficult to correctly construct by hand [38]. It can be difficult to judge which mental operators to include at what time, and errors are easily introduced. CogTool automates the creation of the cognitive models, improving accuracy considerably [38]. CogTool records the interaction sequence and uses the "Adaptive Control of Thought – Rational" (ACT-R) architecture, a computer model of human cognition [39] to generate a model of the task.

10.2. The Experiment

Green and Blackwell [40] define six "cognitive activities" in relation to programming activity: "incrementation" (adding new code), transcription (copying a design into code, or copying code from somewhere else), modification, exploratory design, searching, and exploratory understanding. For our experiment, we primarily covered the incrementation and modification categories. These cover common editing tasks, including inserting and modifying statements, deletion and restructuring of code.

Forty-six common editing tasks were defined and carried out in each of the target systems. CogTool was used to record and analyse the resulting interactions. For simplicity of presentation, the tasks were grouped into five categories: adding new statements (n=6), modifying part of a statement (n=8), deletion (n=12), moving code to another location in the program (n=13) and replacing code with another statement (n=7).

10.3. Results

The recording of the sessions allowed us to observe the number of steps involved in each task, and the cognitive models generate simulated tasks times. The mean task times

Table 1. Mean editing times in seconds for various editing tasks. Lower times are better. Best time highlighted.

	Scratch	Alice	Mindstorms	StarLogo	Python	NetBeans	Greenfoot	Frame editor
Insert	4.87	6.56	15.95	12.50	3.95	5.09	3.80	**1.64**
Modify	5.61	7.05	9.10	8.29	5.44	5.53	5.84	**5.01**
Delete	5.44	2.56	6.51	5.59	5.53	7.82	6.53	**2.42**
Move	5.48	**3.09**	3.82	4.97	5.18	6.01	12.20	4.84
Replace	9.80	8.90	18.55	11.55	5.16	5.10	4.69	**2.29**

produced by the model for each system, grouped by task type, are presented in Table 1.

An analysis of variance (ANOVA) shows that the differences between systems are significant in all groups except "Modify". Mean task times are lower for the frame editor in all groups except "Move". These results support our hypothesis that frame-based editing can improve on the efficiency of common editing tasks compared to existing editors.

11. Evaluation B: Frames vs Text in Middle School

We aided in an evaluation of frame-based editing against text-based editing among middle-school students. The results have previously been published elsewhere [41] but are briefly summarised here.

18 middle-school students were set a task in Java in Greenfoot, while 14 other students were set the same task in Stride in Greenfoot. This allowed the task and IDE to be kept constant between the conditions, with the only difference being the editor paradigm: text-based Java or frame-based Stride. Students were given a 25-minute introduction followed by the 60-minute task.

The students in both conditions rated the activity as low frustration and high satisfaction. No differences in satisfaction were found between Java and Stride; there exists, however, a potential of a ceiling effect. Students in the Stride condition advanced through the task instructions faster than the Java side and completed more objectives with less idle time than Java. Less time was spent making syntactic edits in Stride than in Java, and less time was spent in Stride with non-compilable code. Stride users encountered issues with remembering to press the command key to insert a frame rather than just typing immediately.

12. Evaluation C: Experienced User Study

We conducted a small user study with participants who had programmed before to collect their opinions on using the new editing paradigm. The results of this study have not previously been published.

12.1. Method

We recruited 23 participants by advertising on the postgraduate student mailing lists of the Computing, and Engineering and Digital Arts departments at the University of Kent. The study was approved under the ethics procedure at the University of Kent.

- What, if anything, did you find more difficult in a frame-based editor compared to a text-based editor which you might usually use?
- What, if anything, did you find easier in a frame-based editor compared to a text-based editor which you might usually use?
- Did you complete task 1? Did you have any particular difficulties doing so?
- Did you complete task 2? Did you have any particular difficulties doing so?
- Did you complete task 3? Did you have any particular difficulties doing so?
- If the frame-based editor had all the advanced features of a professional IDE (e.g. easy project organisation and navigation, refactoring support, version control, etc), would you consider using a frame-based editor over a text-based editor? Whether yes or no, please explain your reasons.
- Briefly describe your previous experience in programming: what language(s) you commonly use, how many years you have been programming, and what editor(s) you usually use.
- Do you have any comments you want to make about the frame-based editor which did not fit into the previous questions?

Figure 22. The free-text questions asked in the electronic survey.

In one parallel session, the participants were seated at computers and given a 5-minute verbal introduction and demonstration of the Stride editor on a projected screen. Participants then worked through three sets of tasks in order. Task 1 involved detailed steps to carry out various editor interactions (e.g. inserting frames, deleting frames, editing frames). Task 2 asked participants to enter large pieces of provided code into the editor, and task 3 (for those few who finished all of task 2) was to program extensions to the Greenfoot project they had entered in task 2. Participants worked on the tasks for around 50 minutes, with most near the end of task 2 at the end of the allotted time. Subsequently, they were asked to fill in an electronic survey containing some open questions with free-form text responses (shown in Figure 22) and some questions to be answered using a 7-point Likert scale (shown in Figure 23).

12.2. Results

Participants were asked (as free text) to provide details on their programming experience. 21 of the 23 participants indicated how long they had been programming: the median was 7 years. The most popular programming languages were C++ and Java (17 of 23 participants in each case) and the most popular editors mentioned were Visual Studio (7 participants) and Eclipse (5 participants). We did not collect data on na-

Which editor...	Mean (1 d.p.)	Distribution
...is faster to program in	3.6	
...requires fewer keypresses	4.7	
...makes code easier to read	5.6	
...makes it easier to insert new code	4.4	
...makes it easier to edit existing code	3.4	
...makes it easier to delete existing code	5.6	

Figure 23. The results of comparative questions on a 7-point Likert scale, where 1 meant text-based editor, and 7 meant frame-based editor. Thus, 4 is neutral, numbers lower than 4 favoured the text-based editor, above 4 favoured the frame-based editor.

tionality but we observed that many participants did not have English as a first language. We do not believe this affected their ability to follow the instructions (they were all studying at an English-speaking university), but it did lead to some slightly broken English in the responses.

The results for the Likert scale questions are shown in Figure 23 with details of the distribution of the responses on the 7-point scale.

12.2.1. Negative Comments

To analyse the open text responses, we created categories for the responses and tagged the answers given by the participants, which we will detail now, beginning with the negative issues.

Learning Curve: The issue of the learning curve was raised by 9 participants. By far the most frequently cited issue was remembering to press a command key to insert a frame before typing the content: "The shortcuts to start a new frame take a bit of getting used to (in the beginning I'd often open a frame by mistake by starting typing my line of code)." One participant noted: "especially with the assignation [i.e. assignment frame] because we are use to write the left side first before telling it's an assignation".

Navigation: Four participants mentioned frame cursor navigation being difficult, and another four mentioned some specific cases where they found it difficult to navigate between certain slots using the keyboard. We plan to address these issues in the software; it is our belief that they are solvable usability issues rather than inherent aspects of the design (as, for example, the command keys are).

Expression Editing: Seven participants mentioned that editing text at the slot level could be more awkward than in a text editor. Two highlighted the difficulties of editing incorrect brackets when the brackets must remain paired (which prevents deletion of a single bracket to move it elsewhere). This is a cost of the advantage of never having mismatched brackets. We plan to investigate whether there are usability additions (e.g. allowing the drag of an opening or closing bracket to move it) that can mitigate this problem.

Software Issues: Some responses mentioned issues with the software, such as the code completion interface being slow, one or two small bugs, or the lack of a "saved" indicator when the work is auto-saved. Although we intend to address any problems with the specific software, they do not relate to the editing paradigm design, so we will not discuss them further here.

Two participants also offered the viewpoint that frame-based editing was not as advantageous as it perhaps would have been when text was first used with primitive text editors, because many IDEs now offer what they viewed as similar functionality: auto-indentation, forms of scope highlighting, text selection and mouse-dragging.

12.2.2. Positive Comments

The positive comments were categorised as follows.

Easier to read: As shown in the Likert scales in Figure 23, most of the participants found the frame-based editor easier to read, and this was confirmed by several mentions in the free-text responses (e.g. "It's more structured and easier to follow your code."), although none went into great detail as to why (two mentioned the colouring and scope highlighting).

Easier to insert new code: The Likert scales for easier to insert new code show a slight favouring of frames, and the fewer keypresses most likely relates to inserting new frames too (since the command-keys is one of the major differences to text). This is confirmed by many text responses, e.g. "I like the speed with which you can get code created. I like not having to worry about saving, semi colons, braces and parentheses."

Easier to delete new code: This was one of the Likert items that most favoured frame-based editing, and was backed up by several text responses, e.g. "It is nice being able to treat a whole frame as one unit for the purposes of moving / deleting etc. Since it is broken up into frames you won't get errors with missing / extra closing braces after such tidying up of code."

12.3. Study Conclusions

The main problem participants highlighted was that of the learning curve, and in particular, remembering to press a command key to insert a new frame, rather than just beginning to type. It is impossible to tell from a 1-hour session how long this problem persists, but many participants seemed to view the issue as a temporary initial hurdle rather than a permanent problem.

The participants were almost equally split as to their preference between frames and text. We view this as a very positive result once you consider the context: after only one hour with a new paradigm which has a difficult learning curve, many participants (all of whom were quite experienced in

programming) indicated that they would consider using frames over text.

To avoid overloading participants, we did not make mention of all the frame-based editor features listed in this paper. Several of the more advanced features – such as bird's eye view, the long scope annotation, the inherited methods pane, tooltips, and overtyping – were not explicitly mentioned. Some participants may have discovered a few of them, but we focused primarily on the core interactions and features (the frame cursor, frame insertion and deletion, etc). It may be that participants would have been more positive about the possibilities for long-term use if they had also seen the more advanced features.

Participants praised the readability of frames, and the speed/ease with which you can create new code, manipulate existing frames and delete whole frames. The main sticking point appears to be the editing and navigation of slots and their content. Given that this hybrid approach of text and frames (rather than pure-text of mainstream programming, or only-blocks of Scratch et al.) is novel, there may still be room to improve the text aspect so that any usability issues are minimised or removed.

12.4. Threats to Validity

Participants knew that the experimenters had created the Stride editor software being tested, which created a risk of response bias. To mitigate this, we assured all the participants that their responses were anonymous, and we avoided collecting detailed demographic data (age, sex, etc), and used an online form, to emphasise the anonymity of the responses.

It is possible that being non-native speakers may have hampered the performance of some students both in terms of the software (two students mentioned the QWERTY keyboard layout of our PCs as being a problem for them) and the instructions. However, all students achieved a similar amount of progress through the tasks (all completed task 1 successfully, and most were near the end of task 2) and since we are not using task progress as a measure for our analysis, this does not seem to present a serious issue.

13. Related Work

There are two strands of previous work which relate to frame-based editing: the work on structure editors from the 1980s and early 90s, and work on block-based editing in the past ten years.

These strands of work suffered different fates. Structure editing had a period of limited popularity but – apart from a few Lisp editors – is barely used today. Block-based editing has been a resounding success and is now the predominant form of programming for younger age groups, via systems such as Scratch, Snap!, Blockly editors and many others. Given that our work relates to both, it is instructive to investigate the fate of structure editors, and the relation between the two strands of work.

13.1. Structure Editing: First Attempts

Structure editing started to receive attention in the late 1970s, beginning with systems such as the Cornell Program Synthesizer [42]. Work on structure editors continued throughout the 1980s, producing systems such as GENIE/GNOME [43] and Boxer [44] that saw some success, as well as many other systems. There is a noticeable pattern in the literature, with excited descriptions of new structure editors in the early 1980s giving way to critical retrospectives of structure editors' failure in the late 1980s and early 1990s: "Despite [structure editors'] potential, they have not been successful, as evidenced by limited use. In general, they are perceived as being too difficult to use and the benefits of their use are outweighed by the difficulties." [45].

So why did structure editors fail? We believe there are four main reasons. Firstly, we should consider the computing environments in the early 1980s. Text-mode displays were the norm, mice would only become popular later in the 1980s, and high resolution full-colour graphical displays would only become widespread in the 1990s. Also crucial was that the discipline of human-computer interaction (HCI) was in its infancy in the 1980s. Interface design has improved immensely in the 30 years since. All of these contextual issues meant that the interfaces constructed for structure editors were hampered in what they could achieve.

Secondly, there was a lack of flexibility in the structured approach. Over time, the creators of structure editors noted that structure editing could be awkward and overly restrictive at the lowest levels in the syntax tree. Expressions were difficult to manipulate, and were entered differently to their display in many cases. They were entered prefix, with the operator specified before the operands because the operator is higher in the syntax tree than the operands, but displayed infix as is standard in text editors (outside Lisp). Expressions also caused issues with cursor movement: the cursor was generally moved around the syntax tree which was not visible at the expression level. For example, moving from 1 to 4 in the expression (1+2)*(3+4) required moving "up" two levels (once to the addition operator, once more to encompass the whole bracket), then across one level (to the other bracket), then "down", and right. But these concepts of "up" and "down" are abstract and not visualised; the user must understand syntax trees before being able to navigate their program. As Miller et al. [46] pointed out, structure editors were already quite different from text editors in that the selection in a structure editor is almost always a node and thus a range of text, rather than a point (between characters) as users were used to in text editors. These problems led to many systems providing hybrid editors, where code could be entered either in text mode or in structure mode. As per Minör [47], this could create difficulties for users in obtaining a consistent mental model of the lexical structure of their program code. Welsh and Toleman [48] suggested that users do not think of their program as structured all the way down, which relates to our decision to provide different editing at different levels of the syntax tree.

Thirdly, we believe a further issue was that programming instruction in the 1980s was primarily university-based. First-year university students learning structure editing would be required to switch to text-based editing (ready for exit into industry) within the next year or two. Thus structure editing had only a small window available: it had to provide sufficient benefits that it was worth using for only a year or two, before students had to make a switch which would be unnecessary if text-based tools were simply used throughout. As we will discuss in section 13.3, this is a different environment than the one in which block-based editing would achieve great success.

Fourthly, there were elements of over-design in the editors. Many, if not most, of the structure editors were not a single editor, but rather editor generators. Given a formal language grammar, the generator would automatically produce an editor. As later work in this area pointed out [45, 49], this led to awkward editing interactions. With present-day interface design sensibilities, for example, we would not expect that given a description of a database table, we could automatically generate a perfectly usable web form/user workflow to populate the table. In our own work we have made several decisions specific to the Stride language which would not generalise to other languages: We believe an automated editor generator is highly unlikely to produce very usable editors in each case. (We note that similarly, Scratch and Snap! are editors for specific languages, although Blockly provides a counter-example as it is a framework for creating block languages.)

Crucially, none of these reasons for the failure of structure editors strike at their core argument: that text is an inappropriate way to model structured program code. (Note: we use structured here to refer to the lexical structure of code, not the semantic structuring that "structured programming" referred to in the 1960s). Instead, the arguments were that the structure editors were too unusable. The open question going into the 1990s was: did structure editors fail because no-one had yet worked out a way to produce a more usable editor than a text editor, or would this never be possible?

A fascinating glimpse of the potential of structure editors was provided near the end of the first wave of interest, in a 1992 special journal issue on structure editors. Minör [47] describes there a possible direct manipulation structure editor, and shows a prototype interface (reproduced here in Figure 24, overleaf). We would now identify this as looking like an early prototype of the today's block-based editors. So the structured programming era ended by pointing towards block-based editing, just as interest in it died. Over a decade later, these ideas would be independently reinvented by later work such as Agentsheets [50] and Scratch to great success, as discussed in section 13.3.

13.2. Structure editing: Recent Work

There were isolated pieces of further work on structure editors after the 1990s, followed by several recent examples of work in the area. In 2006, Ko and Myers [31] described Barista, a structured code editor generator (see earlier comment) which allowed editing in text or structured mode as discussed earlier. Barista was a prototype, with a mouse-focused interface for adding new blocks and little support for keyboard navigation or manipulation.

A recent effort at structure editing was presented by Osenkov for C# [51]. The indentation level is under editor control, as in our frame-based editor, and keywords are entered through a classic code completion menu. At times (when the cursor is at the beginning of what we call a frame slot) the cursor shares similarities with a frame cursor. The type of syntactic elements, however, is not explicitly declared by the programmer on entry (as it is for frames through the use of a command key), but rather is deduced from the textual representation once the user has written enough text to resolve the ambiguity between different possible constructs. Thus the C# structure editor does not automate the creation of boilerplate to the same extent, nor does it prevent subsequent editing of keywords.

JetBrains MPS [52] is a framework/generator for creating structure editors. MPS does not have the concept of frames as first-class entities which can be picked up and manipulated, and is closer to classic structure editors than to block-based languages. With no frame cursor, navigation and selection revolve around what we call text slots in our editor, meaning that direct manipulation of existing code is not as easy, especially with the mouse. MPS also lacks the visual presentation of frames.

The Envision editor [53] is structural, but moves much further away from the traditional text-based presentation with methods arbitrarily placed on a two-dimensional grid and the use of icons instead of keywords, and provides alternative visualisation displays such as a flowchart view. It is designed to be possible to visualise large code-bases on a single large two-dimensional display. Envision focuses on keyboard-based interactions, in contrast to our editor, which supports keyboard- and mouse-based interactions.

An alternative approach for touchscreen devices has been created by Almusaly and Metoyer that uses specialised buttons to generate common patterns in code [54]. Like our work, this greatly reduces the number of keypresses required to enter core program code. The keyboard is primarily focused on the entry of code, and does not have the structured frame manipulation that our editor provides.

Other editors support structure editing modes. One example is the ParEdit mode in Emacs, which prevents unbalanced parentheses and curly brackets. Gomolka and Humm [55] described a structure editor for Lisp which also prevented unbalanced parentheses. These tools match frame-based editing's advantage of avoiding unbalanced parentheses/scopes, but do not address issues such as easy automatic keyword insertion, illegal edits of syntactic structure or typing of syntactic symbols. Lisp advocates may argue that this is because control structures should not be special cases in the language; we would argue that by making them known special cases, we gain significant advantages in editing speed and readability.

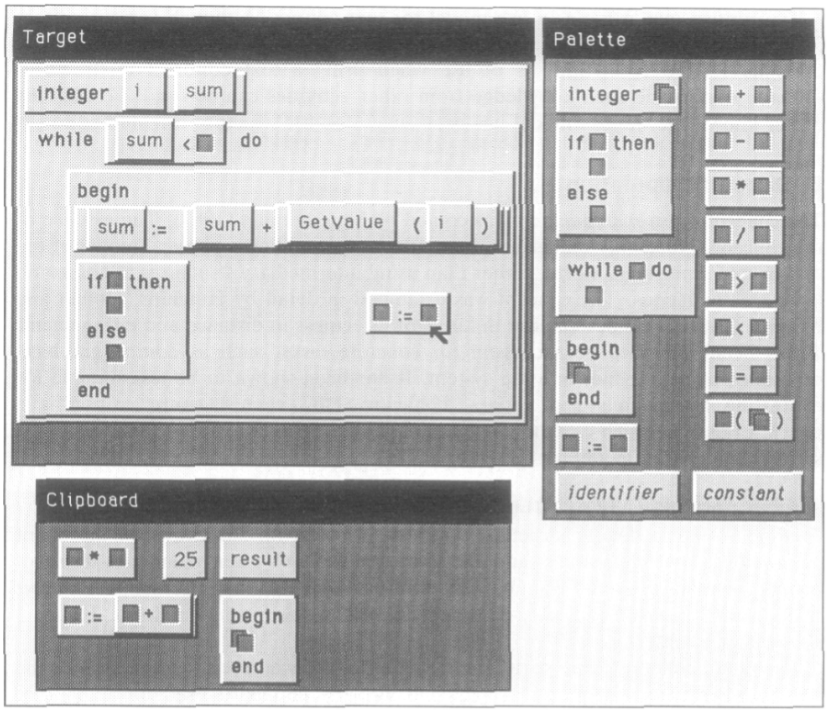


Figure 24. Editor interface proposed by Minör, 1992.

13.3. Block-based Editing

Block-based editing was in a sense a reinvention or re-imagining of many of the ideas of structure editing. It is, therefore, instructive to consider why it succeeded where structure editing failed. We believe there are two main reasons.

Firstly, block-based editing had a more usable interface, informed by fifteen years further development in HCI, and also the ability to use high-resolution, full-colour displays with a fast graphical system. This eliminated many previous constraints on designers of program editors.

Secondly, some of the concerns about structure editors do not apply in blocks' target audience of young children. There are few worries about the transition into text, or compatibility with other tools or many of the other issues which dogged structure editing in contexts such as universities or industrial use. Thus, all blocks had to do was to provide an easier interface for that age group – and the elimination of syntax was an even bigger advantage for young children (who tend to struggle with the precision required in text-based syntax) than older learners.

Meerbaum-Salant et al. [56] examined some of the downsides of block-based editing, highlighting an extremely bottom-up approach, and an issue with "extremely fine-grained programming", where users split their programs into particularly small fragments, making them hard to read. This arises from the way that Scratch and most other block-based editors allow execution of single blocks, and also allow programs to be split into very small event handlers. Neither of these characteristics are present in our frame-based editor: code must be contained in one single coherent class, thus not allowing either of these issues to occur. (There may be other bad habits which frame-based editing permits – something to investigate in future.)

Many block-based program editors have been designed, most quite similar in design to Scratch, the system we have used as our primary example in this paper. We will focus here on work performed to extend block-based programming to be more powerful, or closer to text-based programming.

Alice 3 [57] is a block-based environment that is closer to text programming than many others. It uses method call syntax similar to Java rather than phrases resembling natural sentences, and code is organised into methods. However, like many other block editors, it lacks support for keyboard navigation and entry and does not include many of the other possible representation improvements presented here.

Alice 3 has a text (Java) preview mode, as do other editors such as Tiled Grace [58] and Droplet [34]. The latter two editors allow for a two-way translation between blocks and text representation of the exact same code, with Droplet even preserving the indentation pattern of the original text in its blocks view. This idea of a hybrid editor (editing code either as blocks or as text) has similarities to the hybrid structure editors described earlier, which allowed editing as structure or as text. It is our belief that a successful editing paradigm should not need such a hybrid mode; we seek to merge blocks and text in one representation rather than requiring two distinct modes.

A significant recent effort in block-based editing to add keyboard support is GP [59]. GP is close to Scratch and Snap!, but seeks to improve accessibility, add keyboard support

and generally improve the design of block-based programming languages. The designers have added a block cursor very similar to our frame-based editing cursor, which allows navigation along the same lines. They have chosen a more search-focused paradigm to entering new blocks (since GP has many more blocks available than Stride has frames) rather than our command keys, but GP and Stride are evolving along similar lines (thanks in part to discussions between the two teams).

There have been attempts to apply block-based programming to mainstream programming languages, such as C [60]. This acts as a proof of concept for the transferability of block programming to other languages, but does not add any new editing features to mainstream block editors.

TouchDevelop [61] has several modes, from a classic block-based view through a hybrid view to a mostly text-based representation. The hybrid view includes elements from classic structure editing and frame-based editing; some fields are free text entry like our text slots, while others are similar to the choice slots or code completion, with a grid-based touch-oriented restricted list of selections. It employs a classic text cursor instead of a frame cursor, but syntactic constructs (such as if-statements) can be selected by typing, in a way that is slightly faster than IDE template code completion.

TouchDevelop allows manipulation by drag-and-drop similar to that for frames and similar schemes for frame selection and surrounding a selection with compound statements. Being touch-oriented, the keyboard navigation and manipulation are less flexible than in frame-based editing, resembling classic structure editing. For example, it is not possible to enter a non-existent variable name in an expression, intending to declare it later – an intermediate error state that we explicitly choose to allow in frame-based editing. (This is a significant difference in programmer flexibility: A study of programmer behaviour [28] found that around one-fifth of all name edits resulted in the use of an as-yet undeclared name.) TouchDevelop does not contain the structured expression slots of frame-based editing, nor many of the informational display elements described here, but its hybrid mode, and GP, are probably the most similar of the existing systems to frame-based editing.

13.4. Contribution and Novelty of Frame-Based Editing

The related work described here has a complex correspondence to frame-based editing. Several individual features of frame-based editing are present in isolation in other editors, while we believe others are entirely novel. The contributions of frame-based editing encompass these new features and new combinations of existing features into a robust editor publicly released and available for serious use:

- The horizontal frame cursor, as originally developed by McKay and Kölling, is a new feature for block-like programming (as a target for keyboard-based insertion and selection rather than just as a drop target), which enables or simplifies a variety of keyboard based editing interactions, and adds new interactions such as wrapping a selection into a control structure.
- The combination of keyboard overtyping, bracket balancing and automatic spacing in expression slots (and prompt text for parameters), while partially available in existing IDEs, is a novel way to enter expressions.
- Keyboard-enabled frame extensions are a new way to manipulate blocks that can be arbitrarily extended with additional body elements.
- The display of inherited methods as a foldout display within the body of the code itself (rather than as a pop-up display) is not present in other editors.
- The contextual code completion is not available outside professional text-based IDEs, and here goes beyond them, e.g. restricting the types available for catch clauses.
- The Cheat Sheet provides help for textual entry not previously available for keyboard-driven programming.
- The localised per-frame/per-slot history is a novel way to support returning to earlier versions of individual elements of program code.

14. Discussion

Our frame editor aims to fuse benefits of blocks and text editors to increase usability, whether for writing new code, reading or manipulating existing code. We believe there are several advantages for novice programmers compared to block-based and text-based programming:

- Easier to enter new code: Frames save typing much text-based boilerplate, while keyboard commands allow faster entry than block-based programming. Expressions can be entered via a keyboard rather than through mouse-controlled drag-and-drop operations.
- Fewer errors: Many syntax errors, including those related to balancing brackets, punctuation and semicolons, are eliminated.
- Better error messages: Those errors that do remain have clearer messages and more accurate error locations than in text-based programming.
- Readability: Frames are more easily readable than text at the statement level (for example by depicting scope more clearly). At the same time, its look is designed to have lower visual overhead and offer better visual text flow than current block-based editors. This makes programs more readable than in block systems for larger segments of code.

We believe that expert programmers also benefit, particularly in contrast to text-based programming:

- Faster entry: The number of keystrokes required to enter code is significantly lower than in traditional text editors. In addition, there is no responsibility on the programmer to spend time maintaining the layout of code.
- Readability aids: The highlighting of scopes, improved display of method headers and annotations of long scopes all serve to aid readability of standard programs.
- Better navigation: The Stride editor offers quicker navigation for some use cases than many traditional text editors.

This includes easy navigation at the method level and the availability of the bird's eye view.

Many of these beliefs have been backed up in our user studies. Middle-school students spent less time with non-compilable code (in effect, had fewer errors) in Stride and were able to proceed faster than students using Java on the more advanced parts of a task. Expert programmers who tried the system agreed that it made program code more readable, and that inserting new code and deleting existing code was faster in Stride than in text-based systems.

14.1. Separation of Code and Representation

In a frame-based editor, the structure and content of the code and its representation on screen are decoupled.

This removes long-standing problems in programming teams which include members with different visual preferences. Arguments about the preferred depth of indentation or the best placement of curly brackets disappear.

In traditional systems, if one team member changes the number of spaces used for indentation, version control systems would register changes for all lines of code (creating possible conflicts with other edits) even though the code structure is unchanged. Layout changes are indistinguishable from code changes. Thus, the same layout preferences must be imposed on all team members.

In a frame editor, the visual appearance (including depth of indentation and other layout details, including colours) can be a personal preference of any team member. The code structure is shared with the team; the visual representation can be separately customised by any individual programmer without affecting others.

14.2. Edit Flexibility and Tolerance of Error States

The Stride editor allows entry of many erroneous statements and expressions, even where they could feasibly be prevented. When entering a method call, for example, it would have been possible to provide an interface mechanism that only allows entry of existing method names, thus avoiding possible mistyping of identifiers. We chose not to implement this restriction.

While such a mechanism has the potential to prevent some errors, it would also prevent some valid use cases and potential work patterns. A programmer might, for example, conceive of a new method while implementing an algorithm, write a call to this new method first, with the plan to add the definition of that method later. An editor that only allows calls to existing methods would not allow this sequence; our Stride editor does.

Similarly, during the editing of an expression containing multiple operators, the code might be temporarily in a syntactically incorrect state. An editor could force a user to fix a recognised problem before allowing entry of additional code, ensuring that any sub-expression is syntactically correct before allowing the cursor to move away. (The Visual Basic editor used to have this restriction: An attempt to move the cursor away from an incorrect line caused a modal error dialog to appear; this was perceived as deeply frustrating by many users.)

We consider these error prevention possibilities too restrictive. Users must be allowed the flexibility to choose their order of work, including the ability to leave parts of work half finished and incorrect to work on or consult other parts of the system.

We believe that the over-restrictive nature of the systems is a further significant reason that previous attempts at popularising structure editors failed [45]. Structure editors managed to prevent many errors, but most locked users into fixed workflows, resulting in frustrated programmers reverting to free form editors.

In the Stride editor, we do not prevent entry of many incorrect code segments, choosing instead to subtly indicate erroneous code (via a red underline) without blocking the programmer from additional manipulations.

Deciding the exact line to draw – how much help (and restriction) to provide, and how much freedom (and errors) to allow – provides an interesting and complex design space. The concrete decisions made in each individual editor implementation will strongly colour users' views and opinions, determine acceptance or rejection of the tool, and ultimately contribute to success or failure of achieving user acceptance.

Whether we have achieved an acceptable balance in the implementation of the Stride editor remains to be seen. We considered many cases and attempted to make reasonable choices, but whether we have succeeded will only be known after a longer period of use by a larger user base.

14.3. Modal Entry and Learnability

Text editing in a frame editor is modal. Pressing a key in a text slot has a different effect from pressing the same key when the cursor is in a frame slot. The former enters the key verbatim, while the latter is interpreted as a command key that may enter a frame.

Modal interfaces are known for their potential to introduce confusion and misinterpretation in interface operations. Users may misread the mode and their expectations of command entry effects may be at variance with the actual system state and behaviour. Thus, in many situations, it may be beneficial to avoid modal interfaces if possible.

Some examples exist, however, of very successful and accepted modal interfaces. Paint programs, for instance, often offer a variety of paint tools, and the current choice of tool represents a mode. This mode is typically indicated by the shape of the mouse cursor, which may take the form of a pencil, an eraser, or a paint bucket.

Modal interfaces can be successful if the existence of the modes mirrors a user's mental model and the current mode is sufficiently clearly indicated for users to be aware of it at all relevant times.

In the Stride editor, the mode is indicated by the shape of the input cursor. This mirrors the successful indicators in paint programs: Because the indicator is, by definition, at the

point of the user's focus, it is implicitly noticed and hard to overlook.

The text cursor takes a very traditional shape, indicating text entry as in common editors. The frame cursor has a very different, distinct and prominent shape and colour, signalling different behaviour.

Whether the cursors as visual mode indicators are sufficient will remain to be seen with increased use of the Stride editor. Initial observations in user trials are promising: Users seem to adapt to the two-mode editing model easily and quickly, without the need for formal introduction or explanation.

One misinterpretation remains longer than others after a short time of use of Stride: When entering method calls, novice Stride users often forget to issue the method call command (space) before starting to type the method name. Just starting to type the name seems a habit that is harder to break than with other commands, perhaps because no modern language uses a keyword before method calls, whereas other constructs (if, while, etc) do begin with a keyword. Whether this habit disappears after some time remains to be seen; currently, there are not enough longtime Stride users to assert this either way.

It is our intention to conduct further detailed user studies to assess learnability and usability issues. One of our existing studies (evaluation B) already turned up one example issue: the ability to delete an entire frame by placing the frame cursor before/after it and hitting delete/backspace respectively was used accidentally by students who then removed a large piece of their program without meaning to. The students did not generally think to undo, preferring instead to re-enter the code. We have now added an overlay after deleting a large piece of code which offers to undo it. We believe that other issues will be on this scale: minor to medium usability issues which can be designed against, rather than serious flaws in the interaction model.

14.4. Code Folding

Some text-based program editors include support for code folding: Selected segments of code (usually constructs which in our editor are represented by frames) can be temporarily "folded in" to reduce them to their header, similar to the bird's eye view described earlier. Frame-based editing and code folding clearly can work well together – we could easily provide a clickable control (and a shortcut key) which reduces any given frame to a single line. Blockly already does this for block-based editing, for example.

In Stride, we have chosen not to do so. The main reason for this is our personal belief that code folding is not very useful when good navigation functionality is provided. However, it is clear that code folding can easily be implemented in a frame-based editor.

14.5. Accessibility

Making programming accessible, for example to blind programmers, is a difficult challenge [62]. Furthermore, it has been noted as a particular issue for block-based languages [63] (and was a motivation for keyboard support in GP [59]), due to their focus on a visual display and mouse-based manipulation – items less amenable to supporting screen-reader technology than text-based display and keyboard-based manipulation.

It is a shame that the visual aspect and mouse-focus has prevented accessibility support, as the fundamental model of block-based (and frame-based) editors is more suited to accessibility. The reduction in syntax means that blind users do not have to worry about entering or correcting the syntax; with a screen reader, the users must either hear "semi-colon" read out all the time, or if it is not read out, will find it harder to correct when it is missing. The inherent chunking of the code into semantic units should make it easier to provide screen readers with relevant information, compared to the line-based approach that existing screen readers take.

Frame-based editing has several features that should allow it to be more accessible than existing block-based editors. The ability to perform all edit operations (entry and navigation) using the keyboard instead of the mouse allows use of the editor even if the user cannot operate the mouse (physically, or due to poor sight). The way that expressions are entered by typing them rather than entering a series of blocks should allow for faster use by blind users than block-based entry, even with keyboard support for blocks (as in GP). The decision to have all program code in a single long sequence of named methods allows easier navigation and organisation for a blind user than multiple scattered unnamed fragments as in many block-based languages.

14.6. Frame-Based Editing of Other Languages

In this paper, we have presented a frame-based editor for a language very similar to Java (for compatibility, both technical and pedagogical, with our Greenfoot system). However, the principles of frame-based editing are independent of this language. It is possible to edit any typical programming language (or structured tree format, such as XML or HTML) with a frame-based editor.

We note a few features that may make a language more or less amenable to frames-based editing, starting with issues we encountered while aiming for Java compatibility:

- Java's overloading of less-than/greater-than symbols as angle brackets in generic types is not ideal, especially when combined with the casting syntax. In a type slot, this does not pose a problem because here these characters always denote a generic type. But parsing expressions proves more problematic: Because type casts are not identified by a keyword, types can appear in expressions. Thus we must allow having mismatched angle brackets because we must treat these characters as less-than/greater-than symbols, the more flexible form.
- Java generally makes little use of whitespace to separate lexical tokens, which allows us to not require the user to enter spaces in expressions. We can let the user enter 1+2 or x/3 and let the editor space the expression. Because most operators have a different alphabet to operands, we can easily distinguish one from the other. For keyword operators,

such as “new” and “instanceof”, spaces are still required between the operator and its arguments. To alleviate this problem, the instanceof keyword has been replaced with a symbolic operator in Stride (“<:”). The “new” operator is treated as a special case. In general, keyword operators are more awkward than symbols for frame-based editors.

- Java’s lambda expressions remain an open issue for us. A lambda expression can appear in any expression and contain arbitrary segments of code. This means that an arbitrarily complex frame structure could appear in an expression slot. This may become complicated to display and manipulate. (For the same reason we also do not support anonymous inner classes). Devising a manipulation system for this is not impossible, but complicates the interface. A stricter model of programming, where statements contain expressions and no further nesting is possible (avoiding arbitrary levels of nesting of statements inside expressions) is easier to implement in frame-based editors.
- Frame-based editing works well where there is an expression/statement divide, or more generally, a setting where there is a high-level structure with program units, and a lower-level expression editing. Although in theory everything in Haskell is an expression and there are no statements, it would actually be a good fit as you could have frame slots for monadic ‘do’ blocks, ‘case’ statements, ‘where’ clauses, individual functions and so on. In Lisp-like languages where every construct is an S-expression, frame-based editing would be almost non-applicable because there would be no apparent way to distinguish what is a frame vs what is a structured slot.
- Frame-based editing can also be used for non-programming languages. On similar lines to the previous point, it would work well for HTML and XML where each element could be a frame, attributes could be slots, and text content and sub-elements could go in frame slots. Because each element has a name (and potentially type information in the DTD) you could have context-sensitive input assistance. In contrast, JSON is closer to Lisp S-expressions and would work much less well. Languages with markup which have tags rather than begin/end hierarchal structure (e.g. LaTeX’s section commands) would need to be converted to nested frames to work well.

15. Conclusion

In this project, we set out to build a system that combines advantages of block-based and text-based systems.

We have designed and implemented an editor for a new, Java-like language called Stride that incorporates this goal. The Stride editor demonstrates that such a system can be designed and that it presents a number of advantages over each of the competitor systems.

Our initial user studies, on middle-school students and experienced programmers, suggest that Stride has several advantages over text-based programming. Users reported that they found it more readable, and easier to insert new code and delete existing code. When given the same task, students using Stride progressed further than students using Java within the same environment. We continue to iterate and improve the software in order to address possible weaknesses, such as the navigation between slots and editing of structured expressions.

The Greenfoot system including the Stride editor is available for free download at http://www.greenfoot.org/. The same Stride editor has also recently been incorporated into BlueJ, which is also available for free download (http://www.bluej.org/).

15.1. Future Work

There are several avenues for future work. The first is to perform further user studies and evaluation with the editor. Computer science education unfortunately produces many new tools, but little evidence of their effectiveness. We have begun to address this already with the evaluations reported in this paper, but we would like further evaluations, carried out either by ourselves or by others.

A second avenue for future work is to create frame-based editors for other languages, and/or integrate frame-based editors into other tools (such as professional IDEs). (We have frequently been asked if there is a frame-based editor for Python, or whether the editor is available as an Eclipse plugin, etc.) We do not have the resources available to perform this extra technical work, but we would welcome such work by others.

A third avenue for future work is to build on the new editor model. We have so far implemented editor interactions in frame-based editing, but editors in modern IDEs support several other modes of use, such as debugging and version control. We believe there is potential to integrate these other modes of use into a frame-based editor in a better way than they are supported in current plain-text-focused editors.

Appendix

Appendix: Java vs Stride

In one sense it is difficult to precisely define the difference between Java and Stride. The characters which form necessary keywords and syntax in Java are often mere decoration in Stride, which we have kept looking similar to Java. For example, whether Stride shows round brackets surrounding the condition in a while loop is a cosmetic decision, whereas those characters are essential for Java code to be parsed correctly. On similar lines, Stride has no curly brackets around its scopes, whereas Java requires them, Stride displays a var keyword which Java does not use.

Broadly, however, the languages are near-identical. Stride currently lacks several Java constructs: inner classes, anonymous inner classes, synchronised blocks, several advanced modifiers (such as volatile), enums, and interfaces, although we are likely to add the latter two. Stride uses `<:` instead of `instanceof` due to the difficulty of distinguishing variable names from letter-based symbol names, and also disallows

floating-point literals of the form `.5`, requiring `0.5` instead. Semantically, Stride is an unaltered subset of Java: what Stride includes from Java is identical to Java.

Acknowledgments

Some of the initial design work of the frame editor, and the CogTool work presented in section 10 were carried out by Fraser McKay as part of his PhD work. The study described in section 11 was primarily carried out by Thomas Price, in conjunction with Dragan Lipovac and Tiffany Barnes. We are grateful for their contributions.

References

[1] P. Denny, A. Luxton-Reilly, and E. Tempero, "All syntax errors are not equal," in *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '12. New York, NY, USA: ACM, 2012, pp. 75–80.

[2] A. Altadmri and N. C. C. Brown, "37 million compilations: Investigating novice programming mistakes in large-scale student data," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '15. New York, NY, USA: ACM, 2015, pp. 522–527.

[3] D. McCall and M. Kölling, "Meaningful categorisation of novice programmer errors," in *2014 Frontiers In Education Conference*, October 2014.

[4] S. K. Kummerfeld and J. Kay, "The neglected battle fields of syntax errors," in *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20*, ser. ACE '03. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2003, pp. 105–111.

[5] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: Improving error reporting with language models," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 252–261.

[6] M. C. Jadud, "An exploration of novice compilation behaviour in bluej," Ph.D. dissertation, University of Kent, 2006.

[7] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx, "Understanding the syntax barrier for novices," in *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '11. New York, NY, USA: ACM, 2011, pp. 208–212.

[8] G. M. Weinberg, *The Psychology of Computer Programming (2nd edition)*. Dorset House Publishing, 1998.

[9] G. Marceau, K. Fisler, and S. Krishnamurthi, "Measuring the effectiveness of error messages designed for novice programmers," in *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '11. New York, NY, USA: ACM, 2011, pp. 499–504.

[10] P. Denny, A. Luxton-Reilly, and D. Carpenter, "Enhancing syntax error messages appears ineffectual," in *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '14. New York, NY, USA: ACM, 2014, pp. 273–278.

[11] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch programming language and environment," *ACM Transactions on Computing Education*, vol. 10, no. 4, pp. 16:1–16:15, Nov. 2010.

[12] K. Wang, C. McCaffrey, D. Wendel, and E. Klopfer, "3D game design with programming blocks in StarLogo TNG," in *Proceedings of the 7th International Conference on Learning Sciences*, ser. ICLS '06. International Society of the Learning Sciences, 2006, pp. 1008–1009.

[13] S. Cooper, W. Dann, and R. Pausch, "Alice: A 3-D tool for introductory programming concepts," *Journal of Computing Sciences in Colleges*, vol. 15, no. 5, pp. 107–116, Apr. 2000.

[14] D. Wolber, H. Abelson, E. Spertus, and L. Looney, *App Inventor 2: Create Your Own Android Apps*. O'Reilly Media, Inc., 2014.

[15] J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk, "Programming by choice: Urban youth learning programming with Scratch," *SIGCSE Bull.*, vol. 40, no. 1, pp. 367–371, Mar. 2008.

[16] M. Vasek, "Representing Expressive Types in Blocks Programming Languages," Undergraduate thesis, Wellesley College, 2012.

[17] C. D. Hundhausen, S. F. Farley, and J. L. Brown, "Can direct manipulation lower the barriers to computer programming and promote transfer of training?: An experimental study," *ACM Transactions on Computer-Human Interaction*, vol. 16, no. 3, pp. 13:1–13:40, Sep. 2009.

[18] T. R. G. Green, "Cognitive dimensions of notations," *People and Computers V*, pp. 443–460, 1989.

[19] R. C. Thomas, A. Karahasanovic, and G. E. Kennedy, "An investigation into keystroke latency metrics as an indicator of programming performance," in *Proceedings of the 7th Australasian Conference on Computing Education - Volume 42*, ser. ACE '05. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2005, pp. 127–134.

[20] J. Leinonen, K. Longi, A. Klami, and A. Vihavainen, "Automatic inference of programming performance and experience from typing patterns," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE '16. New York, NY, USA: ACM, 2016, pp. 132–137.

[21] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," *ACM Transactions on Computing Education*, vol. 13, no. 4, pp. 19:1–19:40, Nov. 2013.

[22] T. W. Price and T. Barnes, "Comparing textual and block interfaces in a novice programming environment," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ser. ICER '15. New York, NY, USA: ACM, 2015, pp. 91–99.

[23] D. Weintrop and U. Wilensky, "Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ser. ICER '15. New York, NY, USA: ACM, 2015, pp. 101–110.

[24] M. Kölling, N. C. C. Brown, and A. Altadmri, "Frame-based editing: Easing the transition from blocks to text-based programming," in *Proceedings of the Workshop in Primary and Secondary Computing Education*, ser. WiPSCE '15. New York, NY, USA: ACM, 2015, pp. 29–38.

[25] M. Kölling, "The Greenfoot programming environment," *ACM Transactions on Computing Education*, vol. 10, no. 4, pp. 14:1–14:21, Nov. 2010.

[26] J. S. Mansfield, G. E. Legge, and M. C. Bane, "Psychophysics of reading. XV: Font effects in normal and low vision." *Investigative Ophthalmology & Visual Science*, vol. 37, no. 8, pp. 1492–1501, 1996.

[27] M. Bernard, B. Lida, S. Riley, T. Hackler, and K. Janzen, "A comparison of popular online fonts: Which size and type is best," *Usability News*, vol. 4, no. 1, 2002.

[28] A. J. Ko, H. H. Aung, and B. A. Myers, "Design requirements for more flexible structured editors from a study of programmers' text editing," in *CHI '05 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '05. New York, NY, USA: ACM, 2005, pp. 1557–1560.

[29] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, "Blackbox: A large scale repository of novice programmers' activity," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '14. New York, NY, USA: ACM, 2014, pp. 223–228.

[30] Y. S. Yoon and B. A. Myers, "Supporting selective undo in a code editor," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 223–233.

[31] A. J. Ko and B. A. Myers, "Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '06. New York, NY, USA: ACM, 2006, pp. 387–396.

[32] P. H. F. M. Verhoeven, "The design of the mathspad editor," Ph.D. dissertation, Technische Universiteit Eindhoven, 2000.

[33] R. Baecker and A. Marcus, "Design principles for the enhanced presentation of computer program source text," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '86. New York, NY, USA: ACM, 1986, pp. 51–58.

[34] D. Bau, "Droplet, a blocks-based editor for text code," *Journal of Computing Sciences in Colleges*, vol. 30, no. 6, pp. 138–144, Jun. 2015.

[35] "Lego mindstorms," Retrieved on 17 Dec 2015 from http://www.lego.com/en-us/mindstorms, 2015.

[36] F. McKay and M. Kölling, "Predictive modelling for HCI problems in novice program editors," in *Proceedings of the 27th International BCS Human Computer Interaction Conference*, ser. BCS-HCI '13. Swinton, UK, UK: British Computer Society, 2013, pp. 35:1–35:6.

[37] B. E. John, K. Prevas, D. D. Salvucci, and K. Koedinger, "Predictive human performance modeling made easy," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '04. New York, NY, USA: ACM, 2004, pp. 455–462.

[38] B. E. John, "Reducing the variability between novice modelers: Results of a tool for human performance modeling produced through human-centered design," in *Proceedings of the 19th Annual Conference on Behavior Representation in Modeling and Simulation (BRIMS)*, 2010, pp. 22–25.

[39] J. R. Anderson, D. Bothell, M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin, "An integrated theory of the mind." *Psychological review*, vol. 111, no. 4, p. 1036, 2004.

[40] T. R. G. Green and A. Blackwell, "Design for usability using cognitive dimensions," 1998, tutorial session at British Computer Society conference on Human Computer Interaction HCI98.

[41] T. W. Price, N. C. C. Brown, D. Lipovac, T. Barnes, and M. Kölling, "Evaluation of a frame-based programming editor," in *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ser. ICER '16. New York, NY, USA: ACM, 2016, pp. 33–42.

[42] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A syntax-directed programming environment," *Communications of the ACM*, vol. 24, no. 9, pp. 563–573, Sep. 1981.

[43] D. B. Garlan and P. L. Miller, "GNOME: An introductory programming environment based on a family of structure editors," in *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ser. SDE 1. New York, NY, USA: ACM, 1984, pp. 65–72.

[44] A. diSessa, "Twenty reasons why you should use Boxer (instead of Logo)," in *Learning & Exploring with Logo: Proceedings of the Sixth European Logo Conference, Budapest, Hungary*, 1997, pp. 7–27.

[45] L. R. Neal, "Cognition-sensitive design and user modeling for syntax-directed editors," *SIGCHI Bulletin*, vol. 18, no. 4, pp. 99–102, May 1986.

[46] P. Miller, J. Pane, G. Meter, and S. Vorthmann, "Evolution of novice programming environments: The structure editors of Carnegie Mellon University," *Interactive Learning Environments*, vol. 4, no. 2, pp. 140–158, 1994.

[47] S. Minör, "Interacting with structure-oriented editors," *International Journal of Man-Machine Studies*, vol. 37, no. 4, pp. 399–418, Oct. 1992.

[48] J. Welsh and M. Toleman, "Conceptual issues in language-based editor design," *International Journal of Man-Machine Studies*, vol. 37, no. 4, pp. 419–430, Oct. 1992.

[49] B. S. Lerner, "Automated customization of structure editors," *International Journal of Man-Machine Studies*, vol. 37, no. 4, pp. 529–563, Oct. 1992.

[50] A. Repenning and J. Ambach, "Tactile programming: a unified manipulation paradigm supporting program comprehension, composition and sharing," in *Proceedings 1996 IEEE Symposium on Visual Languages*, Sep 1996, pp. 102–109.

[51] K. Osenkov, "Designing, implementing and integrating a structured C# code editor," *Brandenburg University of Technology, Cottbus*, 2007.

[52] M. Voelter, J. Siegmund, T. Berger, and B. Kolb, "Towards user-friendly projectional editors," in *International Conference on Software Language Engineering*. Springer, 2014, pp. 41–61.

[53] D. Asenov and P. Müller, "Envision: A fast and flexible visual code editor with fluid interactions (overview)," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2014, pp. 9–12.

[54] I. Almusaly and R. Metoyer, "A syntax-directed keyboard extension for writing source code on touchscreen devices," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2015, pp. 195–202.

[55] A. Gomolka and B. Humm, "Structure editors: Old hat or future vision?" in *Evaluation of Novel Approaches to Software Engineering*, ser. Communications in Computer and Information Science, L. Maciaszek and K. Zhang, Eds. Springer Berlin Heidelberg, 2013, vol. 275, pp. 82–97. [Online]. http://dx.doi.org/10.1007/978-3-642-32341-6_6

[56] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari, "Habits of programming in Scratch," in *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '11. New York, NY, USA: ACM, 2011, pp. 168–172.

[57] W. Dann, D. Cosgrove, D. Slater, D. Culyba, and S. Cooper, "Mediated transfer: Alice 3 to Java," in *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '12. New York, NY, USA: ACM, 2012, pp. 141–146.

[58] M. Homer and J. Noble, "Combining tiled and textual views of code," in *Proceedings of the 2014 Second IEEE Working Conference on Software Visualization*, ser. VISSOFT '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1–10.

[59] J. Mönig, Y. Ohshima, and J. Maloney, "Blocks at your fingertips: Blurring the line between blocks and text in GP," in *IEEE Blocks and Beyond Workshop*, 2015.

[60] S. Federici, "A minimal, extensible, drag-and-drop implementation of the C programming language," in *Proceedings of the 2011 Conference on Information Technology Education*, ser. SIGITE '11. New York, NY, USA: ACM, 2011, pp. 191–196.

[61] N. Tillmann, M. Moskal, J. de Halleux, M. Fahndrich, and S. Burckhardt, "Touchdevelop: App development on mobile devices," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 39:1–39:2.

[62] A. M. Stefik, C. Hundhausen, and D. Smith, "On the design of an educational infrastructure for the blind and visually impaired in computer science," in *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '11. New York, NY, USA: ACM, 2011, pp. 571–576.

[63] A. Wagner, J. Gray, D. Marghitu, and A. Stefik, "Raising the awareness of accessibility needs in block languages (abstract only)," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE '16. New York, NY, USA: ACM, 2016, pp. 497–497.

Moving Beyond Syntax: Lessons from 20 Years of Blocks Programing in AgentSheets

Alexander Repenning

University of Colorado
Boulder, Colorado 80309
ralex@cs.colorado.edu

School of Education
University of Applied Sciences and Arts Northwestern Switzerland
(PH FHNW) Brugg-Windisch, Switzerland
alexander.repenning@fhnw.ch

Abstract *The blocks programming community has been preoccupied with identifying syntactic obstacles that keep novices from learning to program. Unfortunately, this focus is now holding back research from systematically investigating various technological affordances that can make programming more accessible. Employing approaches from program analysis, program visualization, and real-time interfaces can push blocks programming beyond syntax towards the support of semantics and even pragmatics. Syntactic support could be compared to checking spelling and grammar in word processing. Spell checking is relatively simple to implement and immediately useful, but provides essentially no support to create meaningful text. Over the last 25 years, I have worked to empower students to create their own games, simulations, and robots. In this time I have explored, combined, and evaluated a number of programming paradigms. Every paradigm including data flow, programming by example, and programming through analogies brings its own set of affordances and obstacles. Twenty years ago, AgentSheets combined four key affordances of blocks programming, and since then has evolved into a highly accessible Computational Thinking Tool. This article describes the journey to overcome first syntactic, then semantic, and most recently pragmatic, obstacles in computer science education.*

1. Introduction: Programming is "hard and boring"

The statement "programming is hard and boring" made by a young girl when asked what she was thinking about programming approximately 20 years ago, does not suggest a workable trade-off but instead a heartbreaking lose-lose proposition. Disappointingly, a recent report by Google [1] exploring why women do not choose Computer Science as a field of study listed the top two adjectives describing women's perception of programming as "hard" and "boring." These persisting concerns can be interpreted as a two-dimensional research space called the *Cognitive/Affective Challenges Computer Science Education* space [2] (Figure 1). The "hard" part is a *cognitive challenge* requiring programming to become more accessible. The "boring" part is an *affective challenge* requiring programming to become more exciting. In other words, the big question is how does one transform "hard and boring" into "accessible and exciting?"

DOI reference number: 10.18293/VLSS2017-010

The research described here is my 20-year journey through the Cognitive/Affective space. In the lower left of this space is the "compute prime numbers" using C++ and Emacs activity which, by the vast majority of kids, is considered to be hard and boring. In the upper right corner is the elusive holy grail of Computer Science education providing activities that are easy, or at least accessible, and exciting. This journey started in the lower left corner and is gradually moving towards the upper right corner. The path of this journey is not straight. It includes setbacks and detours. Also, while progress has been made, the journey is far from over.

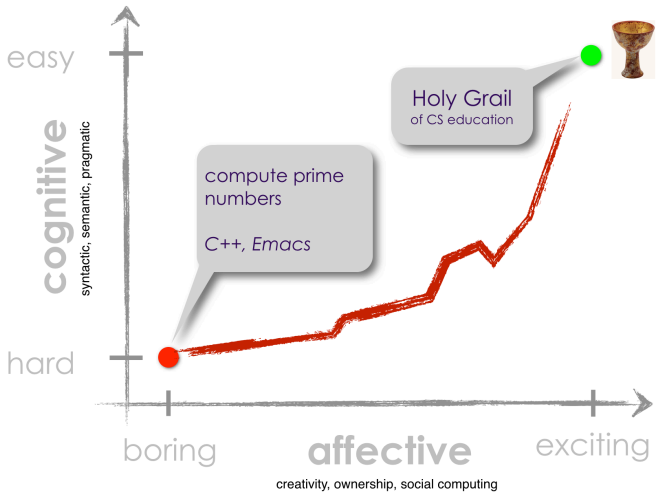


Figure 1. The Cognitive/Affective Challenges Computer Science Education Space.

To explore the affective challenge (Figure 1, horizontal axis) and better understand the reasons why kids would, or would not, want to program, the first question is "What would kids really want to program?" Traditional introductions to programming based on examples such as computing prime numbers are not particularly compelling to most kids. I have developed the Retention of Flow instrument [3, 4] to actually measure motivation. This instrument was applied to our 3D Frogger Hour of Code tutorial and showed that a large

percentage of kids want to and can build games even with very limited time [5]. But what if kids could program games, robots and maybe even simulations? A key to overcome affective challenges and broaden participation with respect to gender and ethnicity is the support of creativity, ownership and social computing [6]. To better understand the rationale behind AgentSheets, it may be helpful to travel back in time to clarify what it was supposed to be used for.

I was fascinated by the affordances of spreadsheets. A simple grid based structure, containing numbers and strings, combined with a formula language resulted in the awesome power to enable an unparalleled fleet of end-user programmers [7] to create sophisticated computational artifacts. These artifacts, in turn, were dealing with an extremely rich set of problems ranging from somewhat dry business applications such as tax forms to highly entertaining topics such as games. What turned gradually into an obsession with grids was nourished even further with events taking place around the same time.

In 1988, as a beginning PhD student, I was in charge of helping scientists to use the Connection Machine (CM2), an intriguing looking, massively parallel supercomputer with up to 65,536 CPUs connected up as a 12-dimensional hypercube. The SIMD architecture of the Connection Machine 2 (CM2) was perfect to compute solutions to problems that can be reduced to cellular automata [8] or Mandelbrot sets in real time. However, even the intriguing look – a huge black cube with a massive panel of wildly blinking red LEDs featured five years later in the movie Jurassic Park – could not overcome difficult programing obstacles. The scientists of the National Center for Atmospheric Research (NCAR) that I worked with had concrete needs to run sophisticated weather models. At first sight the CM2 appeared to be a dream come true. Unfortunately, it was not clear to the scientists how they would benefit from a 12-dimensional hypercube. But perhaps even more of an obstacle was that the programming models they were used to (most of the models they had at the time were written in Fortran) did not map well onto the *Lisp-based programming model featured by the CM2. This mismatch was not limited to the tasks attempted by the NCAR scientists. In 1994, Thinking Machines, the organization behind the Connection Machines, went out of business.

While the CM2 and *Lisp did not become commercial successes, they helped to shape a new parallel mindset to think about problems differently. AgentSheets did not attempt to replicate the 12-dimensional hypercube topology of the CM2, but it did create a highly usable 3-dimensional 3D abstraction based on rows, columns, and stacks. Similarly, StarLogo [9], which also came into existence as a *Lisp prototype on the CM2, also became an end-user modeling environment for parallel processes.

Another milestone in my obsession with grids was the 1989 game SimCity. In my mind, the computational notions intrinsic to spreadsheets, cellular automata, and SimCity-like games started to fuse into a single massively parallel, visual, end-user programmable computation idea that became AgentSheets.

Each idea had its own affordances and obstacles. My goal became to create a framework that could become a synergetic combination overcoming one idea's obstacle with another idea's affordance. For instance, allowing spreadsheet cells to contain animated icons similar to the ones found in SimCity, rather than limiting the content to text, could enable end-users to create more exciting applications such as games and simulations. Cells could contain programmable objects, called agents. These agents could do much more than just computing numbers. They could move around, change their appearance, interact with the user through keyboard and mouse commands, play sounds, use text to speech, react to voice commands, and many more things. Cells could contain multiple agents that could be stacked up. The grid containing these agents became the *AgentSheet* (Figure 2).

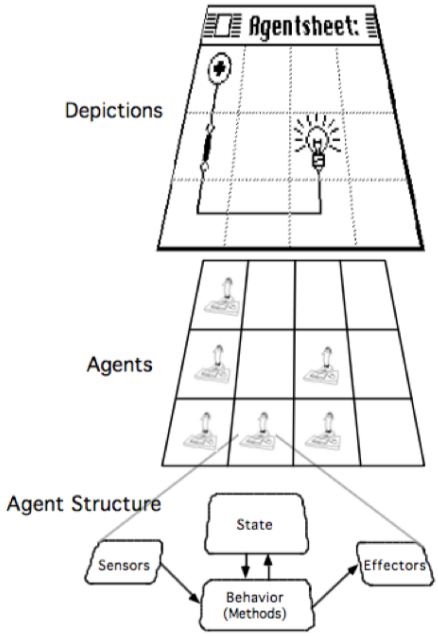


Figure 2. An AgentSheet is a grid containing agents with states including depictions.

Experimenting with agents representing objects such as people, animals, building, wires, switches, bulbs, pipes, buildings, roads, and cars, it became clear that there was a vast universe of exciting applications that could be built with AgentSheets (Figure 3). The target audience of AgentSheets had shifted from scientists to children. The main reason for this shift was that the exploration of the cognition of computing required untainted minds. In contrast to the children, the scientists had strong preconceptions on the very nature of computation based on their experiences with current programming languages such as Fortran. I felt that if I wanted to explore the cognitive challenges of programming then I should start with an audience that did not have any preconceived notions of programming flavored by previous programming experiences. My research platform was based on Common Lisp, a language that is highly malleable for creating new programming languages. The programming language I designed, AgenTalk, was an object-oriented programming

language including Lisp-style syntax to express the behavior of agents. AgenTalk was clearly powerful enough to build a huge variety of applications including SimCity-like games, agent-based simulations, cellular automata and even numerical applications such as spreadsheets. Unfortunately, yet not very surprisingly in hindsight, AgenTalk was too difficult to understand even for the many eager children who wanted to create their own games.

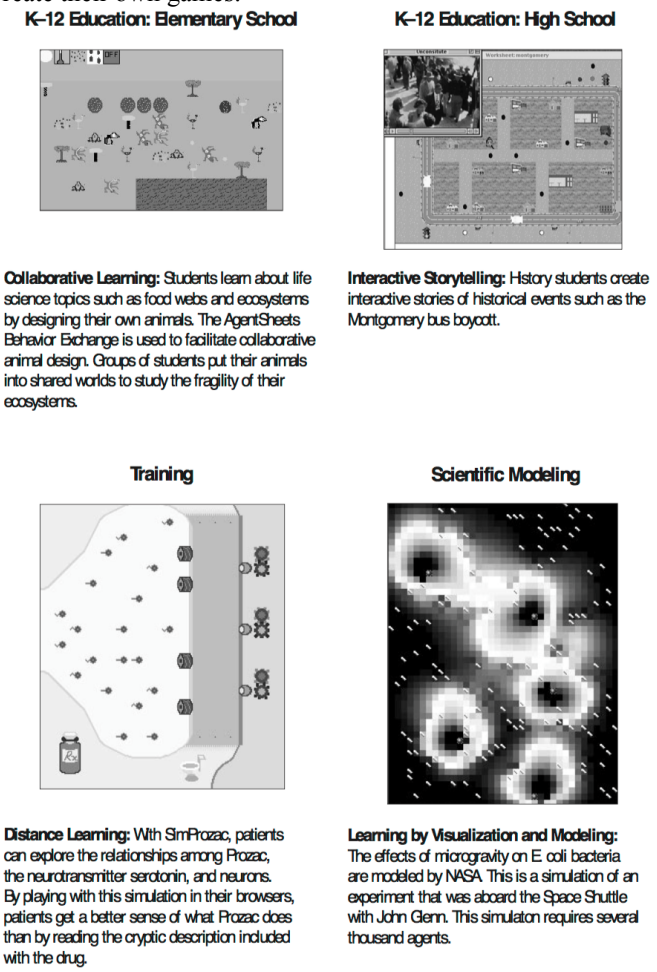


Figure 3. AgentSheets Example applications.

To make programming more accessible and exciting [10], it is necessary to understand complex interactions between affective challenges and cognitive challenges. Kids may be quite excited to build a game, simulation, or robot, but if the tools are too complex then there is a good chance kids will give up because the return on investment is not clear. AgentSheets had turned into a simple but quite promising game and simulation authoring tool. However, AgentSheets was in dire need of a more accessible end-user programming approach that addressed cognitive challenges. Cognitive challenges can be broken down into *three main obstacles*:

1. ***Syntactic***: is about the arrangement of programming language components into well-formed programs.
2. ***Semantic:*** is about helping users with the comprehension of the meaning of programs.
3. ***Pragmatic*** is about practical concerns of programming languages, including the comprehension of programs in the context of specific situations.

At that time it was not at all clear to me that what I initially considered just a minor syntactic challenge should keep me busy for the next twenty years. In my initial obsession with the syntactic obstacle, it took me a long time to recognize, let alone to overcome, the semantic and pragmatic obstacles. Unfortunately, too much of the current research and development of blocks programming is still focused on the syntactic level of challenges. My aim in this paper is to strongly encourage blocks programming researchers to shift away from syntactic towards semantic and pragmatic programming challenges. In my projects, this shift in emphasis has been buttressed by some longitudinal research that unfolded continuously over 20 years, from informal observations in small afterschool programs to large scale national and even international implementations, including the use of sophisticated evaluation instruments. In this paper, I share the lessons that I've learned in my journey, with the hope that they will be useful in other projects.

This paper contains six more sections. Section 2 explores syntactic obstacles through the lens of the AgentSheets genesis. Section 3 defines four key affordances associated with blocks programming, and Section 4 puts this research into a much wider context of related work by considering these four affordances. Section 5 looks at techniques to overcome semantic and pragmatic obstacles. Section 6 outlines a vision for future research called Computational Thinking Tools. Computational Thinking Tools support Computer Science education by carefully balancing cognitive and affective challenges through the support of the Computational Thinking Process (see Figure 19 later). Section 7 concludes the paper.

2. Syntactic Challenges and Beyond

Before I settled on the current form of drag-and-drop blocks programming for AgentSheets, I explored a number of programming paradigms to overcome syntactic obstacles. These obstacles are rooted in the simplicity of creating a syntactically wrong program [11]. Being, for instance, just "one semicolon away from total disaster" with many traditional programming languages can be the source of extreme frustration, particularly for novices. This section illustrates syntactic obstacles by briefly discussing some of the milestones of AgentSheets transitioning from text based programming to blocks programming.

The first approach to overcome syntactic obstacles in AgentSheets was rooted in graphical rewrite rules [12-14]. Initially, AgentSheets [15] was built with a text-based object-oriented extension of Common Lisp called OPUS [16]. A Zipf distribution analysis [17] of OPUS methods used in AgentSheets project revealed that most of the methods used were about making agents move, e.g., a car moving on a road, or changing their appearance, e.g., a person changing from a happy to a sad face. This analysis discovered power laws in

natural language word frequency similar to the frequency of tools used by a blacksmith [18]. The distribution suggested that graphical rewrite rules [12-14] would be a good match because they support the most frequent uses of actions (movement and change) well. Moreover, by combining graphical rewrite rules with programming-by-example mechanisms, these rules could be automatically generated to circumvent any kind of syntactic obstacle. For instance, a train could be programmed to move on a train track simply by selecting it in the scene and moving it one step on the track (Figure 4). The first usability test was so successful that kids had to be forced to stop and go home from the lab. Several iterations of agent-based graphical rewrite rules were explored to enable the creation of more complex games and simulations. Collaboration between the University of Colorado and Apple Computer resulted in several prototypes based on the SK8 programming environment [19]. The Apple team created a SK8 prototype called KidSim [20], which later turned into Stagecast Creator.

However, an effect that I later described as "trapped by affordances" [21] described a shallow learning curve followed by a sudden, steep incline. While it had become tremendously simple to get started, this approach essentially dead ended at a certain level of project complexity when users were trying to do more than just having agents move around and change their appearance. Graphical rewrite rules were powerful enough to create very basic games or animations, but my original goal was to create a framework that could also be used for more sophisticated games and simulations. Graphical rewrite rules fell short of this vision.

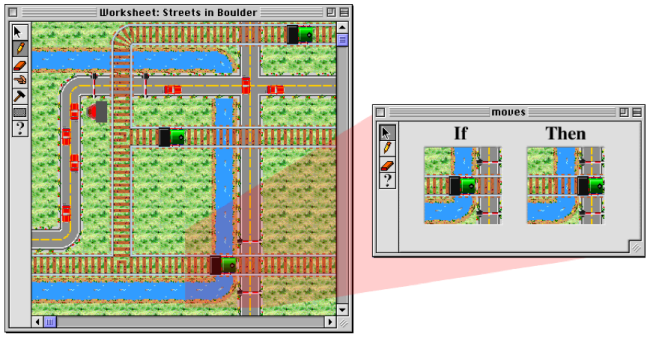


Figure 4. AgentSheets Graphical Rewrite Rule. Double clicking an agent would create a local copy of the agent's situation. Users could demonstrate actions such as moving the train on a train track to the right.

A first step towards the exploration of semantics, with the goal to overcome the syntactic obstacles experienced with graphical rewrite rules, resulted in the creation of semantic graphical rewrite rules [22]. Semantic graphical rewrite rules enabled users to annotate agents with semantic information that could be used to generalize the interpretation of a rule in order to avoid huge numbers of permutations. For instance, a horizontal piece of road, similar to a wire or a pipe, could be annotated to mean that this horizontal symbol represents a connector connecting things on the left with things on the right and vice versa. AgentSheets can, syntactically and semantically, transform agent depictions into all the necessary permutations necessary to facilitate generalization. In a SimCity-like simulation, the user would only have to draw a single horizontal piece of road to have AgentSheets automatically generate all the 16 permutations of road pieces (straight pieces, turns, T-sections and intersections). The 2^4 permutations are the result having or not having a connection in each direction (up, down, left, right). The transformation of the agent depictions applies sophisticated image warping, including the bending of icons [23], to the artwork initially provided by the user. The transformed icons can be further annotated by users. For instance, the dead end road pieces in Figure 5 were annotated with road signs. Also, Figure 5 only shows 15 our of the 16 road pieces. The road piece connecting nothing, i.e., road piece zero, may as well be left off. AgentSheets will also apply the semantic equivalents of these syntactic transformations to agents, e.g., a horizontal piece of road connecting the left with the right, when transformed into a vertical piece of road, will connect the top with the bottom. The net effect of this idea was that the user would only have to draw a single piece of road which could be turned into a complex road system, and then program a car with a single rule to follow that road. In other words, the design and programming of a project that would have taken multiple hours to complete could be compressed into a 5-minute task thanks to semantics. These ideas are of course not limited to roads but apply to any kind of object representing conductivity, such a wires conducting electricity or rivers conducting water. Programming by analogous examples [24] went one step further by allowing users to express analogies such as "a train moves on a train track like a car moves on a road" to map sophisticated interactions from one context to another.

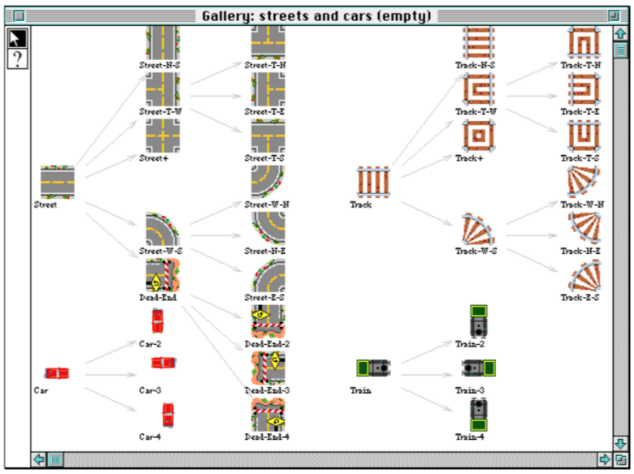


Figure 5. AgentSheets can syntactically and semantically bend, intersect, and rotate transform depictions to interpret rules semantically.

Gradually the notion of blocks as programming language components emerged in AgentSheets. In LEGOSheets the programming language components became more tangible by representing end-user editable [7] rules that users could rearrange and modify with direct manipulation [25] interfaces. LEGOsheets [22] was an AgentSheets derivative based on

spreadsheet-like cells interfacing with sensors and effectors. The programming language used in LEGOSheets became the first visual programming language for the MIT programmable brick. The LEGO Company later created the Mindstorm system based on the MIT programmable brick. LEGOSheets rules are associated with effectors such as motors. To express a rule, a user creates a spreadsheet-like mathematical formula referring to sensors.. Clicking on a sensor adds a symbolic reference to the rule of the effector to be programmed.

The approaches described above reduced syntactic obstacles through the direct manipulation of objects, the agents, instead of typing in text. Unfortunately, not every operation that agents are able to perform could be demonstrated through programing by example approaches. A different approach making all the operations agents can perform accessible to an end-user would be to provide these operations as objects – or blocks – that users could explicitly manipulate. These blocks should be encapsulated objects providing direct manipulation user interfaces [25] facilitating simple end-user editing. That is, users should be able to move them around, duplicate them, and, if they represent operations, control all of their parameters with highly accessible user interfaces. For instance, a color parameter should not be a piece of text that can be mistyped but should be a *type interactor*, called color, bringing up a color selection widget enabling users to pick a color from a color palette. The idea of programming language primitives as blocks already existed. Blox Pascal [26], for instance, already used the notion of puzzle pieces (Figure 6) to represent syntactic relationships between primitives.

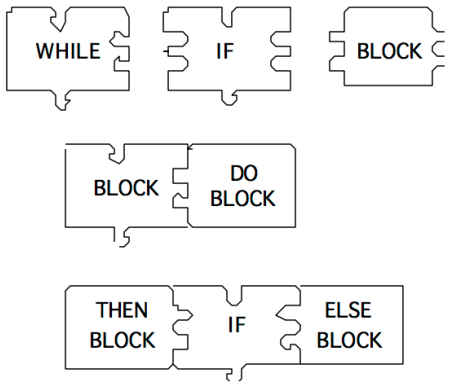


Figure 6. Puzzle shaped Blocks in 1984 Blox Pascal.

Under the title of Tactile Programming [27], AgentSheets introduced a form of blocks programming in 1995 (Figure 7) by combining four affordances defined in the next section. As a tool providing blocks programming to create games and simulations, it made a significant step in moving away from "hard and boring" toward "accessible and exciting." Similar block approaches were later found in Squeak eToys [28, 29], Alice [30], and ten years later in Scratch [31]. Unlike the programing approaches discussed above, blocks programming has stayed with AgentSheets for over 20 years now. AgentCubes [10, 32-35], featuring innovative 3D end-user modeling approaches empowering kids to create their own 3D worlds, includes sophisticated parallel execution and animation models for blocks programming. AgentCubes online is an early Web-based 3D game and simulation authoring tool merging end-user 3D modeling [36] with end-user programming. Common to these tools are three core principles that shaped the creation of blocks programming in AgentSheets back in 1995 [27]:

1. ***Composition***: A drag-and-drop-based approach was employed to aggregate individual programming language primitives, called commands, into a whole program. This was perhaps the most evident affordance of blocks programming. AgentSheets' aim was not to become a general purpose programming environment but a Computational Thinking Tool[1] [37]. To that end, the puzzle piece idea was replaced with a combination of color-coded language primitives, e.g., conditions versus actions, and syntactic drag-and-drop feedback. For instance, the user would get a clear signal through an animated cursor that a condition could not be dragged into the THEN part of an IF/THEN statement. While dragging a block the mouse cursor turns into a green positive indicator when a block fits or into a red negative indicator if it does not fit at the current location. An important concept that is integral to Tactile Programming is that blocks can be composed from any source including from websites.

2. ***Comprehension***: A programming block should be able to explain itself to a user, similar to the way that Rehearsal World [38] could provide explanations for parts of a programming-by-example program. As a programming object, a block can establish connections to objects in the project, i.e., agents. For instance, users can drag actions such as a *move (right)* action onto a frog agent to make it move to the right. This is not an act of programming but a process supporting comprehension. What does this action do to this agent? Likewise, conditions can be tested to learn if they are true or false. Explanation implies that every block can produce an animated description of what it will be doing based on its parameter settings or subcomponents. For instance, the *move (right)* explanation would produce a spoken explanation, using text to speech, highlighting first the "I move" and then saying, while simultaneously making the arrow right parameter blink, "to the right." This would make it very clear how each parameter contributes to the precise meaning of a command. Explaining an IF/THEN statement would explain all of its conditions and actions. Explaining a method would explain all of its statements. These ideas are explained in section 5.2.

3. ***Sharing***: Each command is a sharable object with a canonical textual representation allowing objects to be turned into text and text into objects. Current versions

[1] The original term used was Thought Amplifier, which was not well received.

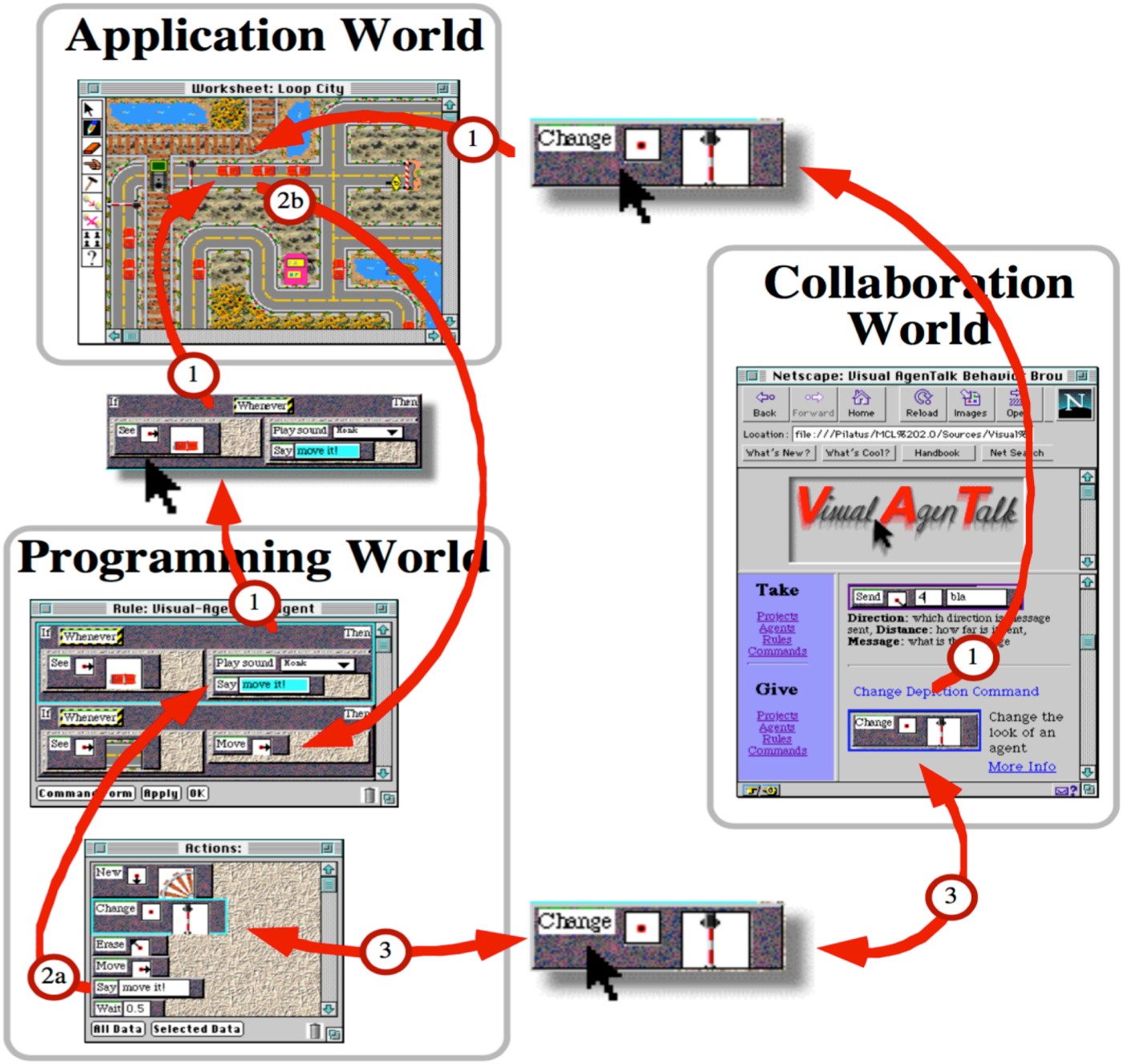


Figure 7. 1996 Figure with original caption: (1) Comprehension: test the functionality of commands and rules by moving them from the programming world or collaboration world into the application world (2a) Direct Composition: select commands and compose them into rules. (2b) Composition by Example: compose rules by manipulating the application world (3) Share: share with a community of users entire simulations, agents, rules and commands through the World Wide Web.

of AgentSheets and AgentCubes use an XML representation. Using some browser exploits – the Web had existed for only 2 years at the time – any project, any program, any agent could be directly shared by dragging it into the AgentSheets Behavior Exchange webpage [39, 40] or dragging it out of there. This enabled a high agile style of sharing but it was greeted with a lot of skepticism in schools, as the practice of easily sharing products, particularly with identifiable authors, was not compatible with common school practice.

3. Four Key Affordances for Blocks Programming

The next section puts the AgentSheets exploration of syntactic obstacles into a much wider context of related work relevant to end-user programming. Reflecting back now 20 years, the Composition/Comprehension/Sharing framework captured important aspects relevant to blocks programming. However, to meaningfully discuss related work, it makes sense to identify a minimal set of affordances that need to be provided in order to be considered a modern blocks programming system. The notion

of blocks as representations of programming objects alone is not sufficiently discriminatory as blocks can be found in most visual programming languages. The value of using the notion of blocks programming as a mere synonym for visual programming would not be clear.

When I review the beginnings of AgentSheets in the context of other visual programming work that was going on at the time, four affordances stand out as being particularly important. I may not even have recognized their full importance at the time, but do so in hindsight. These affordances have turned out to be key aspects of today's blocks programming environments. As part of the Composition/Comprehension/Sharing framework, sharing is a powerful idea [39, 40], with important consequences for the cognitive as well as the affective challenges, but it does not have to be part of the minimal requirements for blocks programing languages. AgentSheets combined these four key affordances into a highly accessible visual programming paradigm. These affordances continue to be at the core of popular blocks programming languages [41] such as Scratch [31] and Blockly [42].

1. ***Blocks are end-user composable***. Simple end-user manipulation techniques, frequently drag-and-drop style manipulations, are used to compose blocks into programs represented as linear, multidimensional, hierarchical or other kinds of organizations. The block manipulation can be based on two- or three-dimensional mouse, gesture or virtual reality interfaces. To be usable by end users, the composition process must include some scaffolding mechanisms supporting the syntactically correct composition of blocks into programs. Examples of such scaffolding mechanisms include context aware menus (e.g., Alice), animated cursors (e.g., AgentSheets/AgentCubes), animated insertion points, enabled/disabled screen regions, and block shapes/colors (e.g., Scratch) suggesting syntactic compatibility.
2. ***Blocks are end-user editable***. As interactive objects, blocks are not just static entities such as icons on a computer screen or physical objects such as plastic cards but dynamic objects that contain end-user editable information. To minimize syntactic challenges, blocks will typically employ direct manipulation interfaces to implement edit operations. For instance, a color value would become end-user editable by using a color picker (e.g, AgentSheets and eToys) to select a color from a palette instead of using an editable text field to enter color values.
3. ***Blocks can be nested to represent tree structures***. Blocks may be composed recursively into tree structures to contain blocks, which, in turn, may contain more blocks. In AgentSheets, a method block contains rule blocks, containing IF and THEN blocks, containing condition and action blocks. In Scratch loops contain instructions.
4. ***Blocks are arranged geometrically to define syntax***. The semantics of block combinations emerges from where blocks are connected by having the blocks touch each other directly or be placed in particular positions relative to one another (block geometry) rather than being linked indirectly by additional explicit graphical connectors like lines (block topology). The definition of geometry may be aided by jigsaw puzzle appearance like in Blox Pascal [26] or Scratch, but does not have to be. This distinguishes modern blocks languages from a style of visual languages that Lieberman calls "icons on strings" [43], epitomized by dataflow languages such as LabView.

Particularly when keeping an eye on educational applications, the "end user" aspect of these affordances is incredibly important for modern blocks programming languages. With the one common goal to make programming more accessible, blocks programming languages need to provide some evidence of efficacy to validate "end user" compliance. Minimally, systems should provide evidence of end-user usability. In the case of AgentSheets, validation has gone much further. Related to cognitive challenges, the Computational Thinking Pattern Analysis research instrument [44-48] has shown that users, by building games with AgentSheets, can acquire important Computational Thinking abstractions, which they can later leverage to build scientific simulations [49]. Howland has explored similar Computational Thinking transfer in the context of the FLIP game-programming tool [50]. Related to affective challenges, the Retention of Flow research instrument [3] has measured motivational levels in Hour of Code activities based on AgentCubes online and shown that the "Make a 3D Frogger" activity has even exceeded motivational levels of high production activities such as the code.org Hour of Code Angry Birds activity [4]. Finally, but perhaps most importantly for educational applications, the Scalable Game Design project [51] has shown with large national studies (student n > 10,000), that teachers can be sustainably trained [52] to use AgentSheets and AgentCubes to the point that they can teach students to build sophisticated games and simulations.

4. Related Work

This section discusses the genesis of modern blocks programming languages through the lens of the four affordances above, which address the core problem of syntactic challenges. In the context of a variety of concrete programming languages, the roles of these affordances will become more apparent. The notion of blocks as objects to be used for programming emerged early on and evolved gradually, raising and, to some degree, answering questions such as: What is a block, what does it look like, how does it get manipulated by a user, how do blocks relate to each other? Most visual languages [53], with their aim to make programming more accessible, include some notion of blocks.

The idea of blocks as visual programming components can be traced to early interactive computer systems. In 1962, on

one of the first transistor-based computers, a TX-2, Ivan Sutherland developed the revolutionary Sketchpad CAD (computer-aided design) program to interactively sketch two-dimensional shapes [54]. Only two years later, also on a TX-2, his brother William "Bert" Sutherland employed the idea of sketching as the root of a two-dimensional programming language [55] implementing an electric circuit metaphor. That language was based on data flow and included two-dimensional components representing mathematical functions such as addition and multiplication. These components can certainly be considered blocks that were, to some degree, end-user composable (affordance #1) but the blocks were not editable (affordance #2), were not nestable (affordance #3), and their semantics did not emerge from the block geometry but rather from the explicit connection of blocks. The Grail project [56] expanded on this by adding a basic ability to edit (affordance #2) blocks through tablet input. Remarkably, for that time, input was pen based including letter recognition. Later, at a time when through the release of the Apple Macintosh mice as user interface devices had just started to become more widely available, Minsky already demonstrated the use of "finger on screen" gestures [57] as a manipulation interface which she used to create a visual programming language layered on top of Logo.

Blocks can represent programming components at syntactic and semantic levels. For instance the logic objects in Minsky's system represent AND, OR and NOT gates that feature well-established semantics rooted in integrated circuits. In this case, the shape of a component represents its semantics, i.e., its meaning and has nothing to do with its syntax, i.e., how it can be combined with other blocks into a well-formed structure. An AND gate functions differently from an OR gate and everybody with an electrical engineering background is able to instantly tell this difference based on the shape of the block. The Blox Pascal (Figure 6) system [58], in contrast, was perhaps the first system shifting radically from a *shapes representing semantics* to a *shapes representing syntax* visualization model. It employed the notion of jigsaw puzzle pieces to present visual clues on how components can be combined. In other words, Blox Pascal uses the shape of blocks to represent syntax. Most modern blocks programming systems, including Scratch, are using the shape of blocks to represent syntax.

This shift from shapes representing semantics to shapes representing syntax fundamentally changed the notion of visual programming semantics to one where the semantics of programs emerged solely on the geometry of blocks (affordance #4) and not on the use of explicit graphical clues such as lines connecting blocks. In contrast to "icons on strings" [43], each modern blocks program consisting of connected blocks has a canonical gestalt [59]. In AgentSheets/AgentCubes, blocks vertically aligned imply top-down sequence. Actions in a THEN part of a rule will be executed from top to bottom. The geometry of blocks in icons-on-strings languages is essentially irrelevant. Blocks can be placed everywhere and then connected with lines. Of course, most programmers will try to strategically position blocks to keep connections short and to avoid spaghetti code by minimizing the number of lines crossing over each other.

Affordance #1 (blocks are end-user composable) and affordance #4 (blocks are arranged geometrically to define semantics) in modern blocks programming languages work hand in hand. That is, modern blocks programming languages provide manipulation mechanisms including a feedback system to support the construction of syntactically correct geometry. Blox Pascal-like programming languages employ the jigsaw puzzle piece notion to indicate how to properly combine blocks. As Glinert [58] and later Lewis [60] indicated, the jigsaw approach is limited by lacking flexibility for connecting blocks. Each connector can only fit one matching counter piece. Polymorphic syntax compatibility is difficult to implement with static shapes. Some [58] have suggested dynamic shape shifting approaches but without providing implementations. Lerner et al. [61] and Vasek [62] have implemented polymorphic block connector shapes. AgentSheets employs a dynamic cursor approach that shows a green positive cursor where blocks can be added and a red negative cursor where they cannot. This approach is further supported by strategically positioning blocks palettes. In AgentCubes, the conditions palette (see later in Figure 12) is immediately next to where conditions go and, likewise, the actions palette is immediately next to where actions go.

There are many drag-and-drop programming systems using some kind of blocks, but they only implement a subset of affordances #1-#4. TORTIS by Perlman [63] was an early system that came very close to modern blocks programming. In addition to featuring direct manipulation interfaces consisting of physical button boxes to control a mechanical turtle, TORTIS featured a so-called slot machine for programming. Slot machines were boxes representing procedures defined by the arrangement of plastic cards. These cards can be considered blocks in the sense that they represent program instructions such as move forward or turn. TORTIS featured blocks that can be composed physically (affordance #1), that have a limited sense of nesting (cards could not contain other cards but a card could be a placeholder for another box containing more cards: affordance #3), and the sequence of program steps was determined by their geometry (affordance #4). However, instructions were not editable (affordance #2). ChipWits [64] was a robot control game providing powerful control flow based on graphical instruction tiles to program robots. The tiles were drag-and-drop composable (affordance #1), but individual tiles were not editable (affordance #2) nor was there a nested notion of tiles (affordance #3), and the program control flow was determined by explicit arrows and not the geometric location of blocks (affordance #4). Logoblocks implemented a Logo-based visual programming language to control simple robots [65]. Logoblocks did provide blocks that were drag-and-drop composable (affordance #1), did have nested blocks, e.g., the REPEAT block (affordance #3), and featured blocks that were arranged geometrically (affordance #4). However, it had a limited notion of block editability (affordance #2).

Two systems stand out with respect to blocks that are recursive (affordance #3). Boxer, a programming system aimed at "nonprogrammers" [66], focused on boxes as nested containers of code, data or images. Boxes in Boxer are blocks that can be composed through drag and drop (affordance #1), can be nested (affordance #3), and are arranged geometrically (affordance #4). The only shortcoming with respect to modern blocks programming languages was its lack of end-user editing (affordance #2). Users could edit the content of a programming block, but in order to do so they had to know textual programming. Similarly, Janus [67] had a very strong sense of recursion. However, it was not focused so much on the recursive construction of user created programs but the animated execution of recursive algorithms.

Following AgentSheets, a growing list of modern blocks programming languages emerged providing all four affordances. In chronological order of their creation, not necessarily in order of their publication, some of the important systems are briefly listed here. eToys is a blocks programming extension to Squeak [29] that emerged in 1997. Around the same time the Alice system provided accessible programming for kids [30]. Scratch became a popular programming tool for creating and sharing animations [68]. Blockly [42], an open source blocks programming language, was used in Hour of Code tutorials and is now used in a number research projects creating custom blocks programming languages.

Programming by Example (PBE) is the idea that programs can be automatically created by observing users manipulating worlds instead of writing programs [69]. The notion of blocks is somewhat secondary to PBE as, at least initially, the idea was that programs that were generated through user manipulations should be hidden from users. Some tried to avoid the need for explicit program representation by keeping PBE demonstrations and resulting programs short. Rehearsal World [38] provided a very simple programming-by-example approach in which users would demonstrate one step. For instance, they could start recording, select a button and then describe the action to associate with that button. Others, including Halbert [70], explored approaches to make recorded programs explicitly available to users. His specific aim was to make PBE more useful by enabling users later to add control structure to a recorded program. Providing affordances #1 and #2, one of the few PBE systems that came close to a modern blocks programming language was Pygmalion [71]. It did provide the notion of block through its representation of icons as placeholders for programs. Users could enter data, typically numbers, which then they could operate on through the explicit application of operators and record the computation. Conceptually speaking, nothing would prevent PBE systems now from being combined with modern blocks programming languages to provide all four affordances. However, perhaps due to the perception that modern blocks programming languages are already highly accessible, PBE research appears to have lost some momentum particularly for educational applications.

Not qualifying as modern blocks programming languages because their semantics do not emerge from the geometry of blocks (affordance #4), there are numerous visual programming languages based on the icons-on-strings approach. The majority of these languages employed connections between blocks to express either data or control flow. Data flow has been particularly popular. Especially early versions of data-flow-based visual programming languages tried to aim at a really wide scope of application, proposing data flow as a general purpose programming model. Prograph, for instance, is a general purpose data flow visual programming language including strong typing and other properties of object orientation such as multiple inheritance [72]. VisaVis [73], employing an implicit type system, demonstrated that it was able to express algorithms such as Quicksort more compactly than Prograph. Other systems developed a more task specific [74] perspective of programming. DataVis [75], for instance, was a data-flow-based visual programming environment helping users to create visualizations of scientific data. In spite of their overall limited use in Computer Science education, some icons-on-strings programming languages are very popular. LabView, for instance, is used by many professional programmers working on embedded systems and robots [76].

Some visual programming languages have experimented with different manipulation mechanisms for block composition (affordance #1). For instance, the CUBE programming environment [77] has proposed a three-dimensional programming language to be embedded in a virtual reality environment. This would allow users to compose Prolog-style, Horn-clause based programs by grabbing, placing and moving components in three-dimensional worlds. AgentCubes, the 3D cousin of AgentSheets, in contrast, enables users to build 3D worlds but it only uses two-dimensional programming.

Domain-oriented programming and design environments [78] employ more abstract blocks including appropriate composition approaches (affordance #1). In these environments, blocks would not represent the traditional computer programming language components such as loops or conditional statements but rather components inspired by objects known to users in specific problem domains. Similarly, the process of composing would often not be perceived as or called programming but as a process of design. Construction kits, for instance, present users with design components that can be assembled at a problem domain abstraction level. The Pinball construction kit [78] provides users pinball components such as bumpers and flippers to design working pinball machines. Similarly, the Incredible Machine [79] provides users with design components that they can arrange into Rube-Goldberg-like puzzles. AgentCubes online provides an even wider range in the Consume ↔ Create spectrum [49] by integrating ideas of construction kits with blocks programming. AgentCubes online differentiates between play, design, and edit mode, providing not only components but also a mechanism to design and program these components. For instance, in edit mode users can create a SimCity-like world

consisting of components such as roads, buildings and cars. At the edit level, users would have to also provide the program to express the behaviors of these domain-oriented components, such as the cars following roads. At the design level, other users could clone a SimCity-like project to design their own city, similar to using existing SimCity like games. In contrast to the Pinball Construction Kit and the Incredible Machine, however, AgentCubes users would still be able to access the lower level programming if so desired. This may be useful to mod [80] the component's behavior.

Domain-orientation is not limited to construction kits but includes text as well as visual programming-based languages aimed at specific application domains. StarLogo TNG/StarLogo Nova is domain oriented towards the end-user programming of simulations [81]. Similar to AgentSheets and AgentCubes, this domain-orientation manifests itself in the support of simulations. Both systems, for instance, scaffold the typical simulation operations such as being able to count all instances of a class and plot these numbers or export them to spreadsheets (see later in Figure 26).

5. Shifting Focus to Semantic and Pragmatic Obstacles

Now, after 20 years of experience with designing drag-and-drop blocks programming languages and conducting large scale, national and international teacher training, I can reflect on the relevance of affordances and obstacles introduced in blocks programming. A key problem of the blocks programming community is its preoccupation with syntactic affordances and obstacles. The syntactic obstacles of programming are quite relevant to novice programmers. Typos involving missing or misplaced special characters such as semicolons can be the root of deep frustration and may be responsible for prematurely terminating the interest of novices in programming. However, the approaches that have emerged from blocks programming have largely addressed these syntactic obstacles. Even traditional text-based programming environments have benefited from approaches such as symbol completion to manage syntactic obstacles in ways that help novices and experts alike. There is a common perception among users that programming has not only become more accessible but actually is now accessible thanks to blocks programming. This is simply not true. An analogy may help. Blocks programming overcomes syntactic obstacles in programming languages in a way that is similar to how spelling and grammar checking overcomes syntactic obstacles in natural languages. But just because we have tools such as modern word processors, including powerful syntactic tools such as spell checking does not mean that we become enabled to write meaningful, interesting, and relevant text. In other words, if I would instruct a user to "go ahead and write a bestselling novel now that you have spell checking" most people would agree that spell checking, as a syntactic affordance, provides essentially no support towards this ambitious goal. The same holds true for blocks programming. With the syntactic challenge essentially being resolved, it is becoming urgent to dramatically shift research agendas to focus on the much harder semantic and pragmatic levels of programming languages.

The following sections describe some of my early explorations of *semantic and pragmatic affordances* that are relevant, but are not necessarily limited to, blocks programming languages. Importantly, these explorations should not be considered end points of investigation but more general research directions, including concrete starting points. In contrast to syntactic obstacles, some of the semantic and pragmatic obstacles are not just incrementally harder to overcome, but at some theoretical level may actually be impossible to get over in the most general case. For instance, the halting problem, which applies to semantic program analysis of Turing complete programming languages, suggests that there are semantic challenges that are simply undecidable in ways that would be impossible to overcome with any kind of computing. While this theoretical barrier exists, it does not imply the need to give up. Computers have become more powerful and more expressive. While faster computers alone cannot overcome theoretical barriers, they can enable new kinds of user–computer interfaces relevant to programming. Employing multiple cores, a computer can now efficiently run multiple threads to constantly analyze complete or partial programs. Fusing program analysis, program visualization, and real-time user interfaces, the powerful combination of computer affordances with human abilities can result in radically new support mechanisms to make blocks programming move beyond syntax.

My ultimate goal for blocks programming is to reach the level of pragmatics described by Webster as "The study of what words mean in particular situations." Blocks are just like words in natural languages. Pragmatic support suggests not only the notion of blocks executed in the context of other blocks, but also of blocks executed in specific situations defined by the aggregation of agents/objects comprising complex game and simulation worlds. When I program a Frogger-like game, what will my frog do when it is in this or that situation in the game? The game worlds need to be considered part of the programming environment to enable these kinds of explorations by the user supported by the computer. The following sections outline approaches that have been explored to move blocks programming beyond syntax in AgentSheets [18, 22, 40, 82-85] and AgentCubes [10, 32-35]. As the main tools of the Scalable Game Design curriculum [51, 86], AgentSheets and AgentCubes include the mechanisms described below. They are being used by students around the world [87, 88] and have been tested with respect to cognitive [47] and affective challenges [3].

5.1. Contextualized Explanations: Support Comprehension

To become more accessible, programs should be able to explain themselves. This is relevant to every kind of programming, but essential to blocks programming, which is

aimed at novice programmers with little or no programming experience.

One approach to increase the self-disclosure [89] of programs is to make programming languages more oriented towards natural languages. For instance, AppleScript, a textual scripting language for MacOS, was intentionally designed to be more readable by avoiding special characters and through some degree of verbosity. The relatively high AppleScript readability is traded off by the obstacle of actually reduced writability. Figure 8 shows a sample AppleScript-generated dialog based on this script:

```
display dialog "Bad news!" with icon
stop buttons "Okey dokey"
```

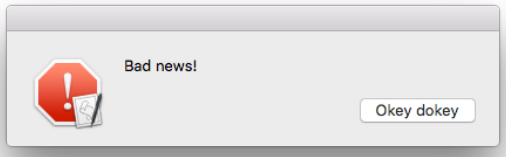
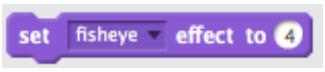


Figure 8. AppleScript generated dialog.

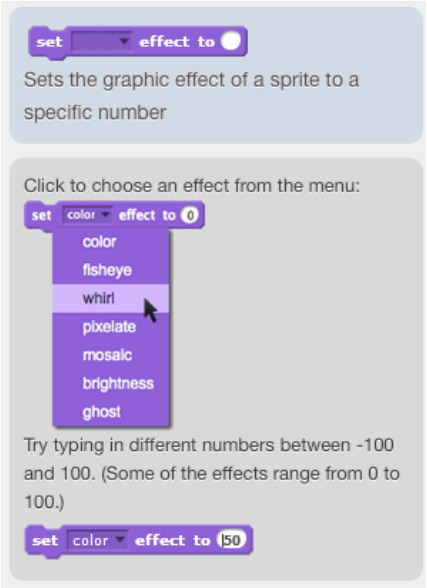
Blocks programming has additional options to make programs more self-disclosable without trading off writability for readability. Because blocks are objects on the screen, it is quite simple to add static or dynamic annotation features, such as tool tips, to programming primitives to explain them. Turkle has used the notion of "objects to think with" [90], talking about objects in the game world such as the Logo turtle. However, blocks programming can extend this notion to the programming language itself by making its objects, in other words the blocks, also objects to think with. This kind of thinking can be supported at three different levels:

Syntax: At the syntactic level, explanation is limited to language structure. For instance, an explanation could reveal that a condition is part of an IF statement and should not be confused with an action that can be executed in the THEN or ELSE part of a statement. However, this information would not include attempts to define the meaning of a specific condition. In most blocks programming languages, this information is captured statically through the visual representation of a primitive via a shape (e.g., puzzle piece approach [26]) or color and/or dynamically, such as through drag-and-drop feedback suggesting compatibility of blocks.

Semantics: At the level of semantics, explanations are often implemented through help functions describing the meaning of a block. For instance, when engaging Block Help in Scratch to explain the *set fisheye effect to 4* command (Figure 9), the user gets a semantic response in form of a generic help panel including a brief description of the meaning of the command and the listing of additional options. Importantly, the description is not about the specific form, i.e., the particular situation of the actual command in question. It does not explain what the "fisheye" effect is or the effect number 4 means in the context of actual situation, e.g., by applying it to an example shape created by the user.



(a) specific command



(b) generic explanation

Figure 9. A command and its explanation.

Pragmatics: At the level of pragmatics, explanations need to be constructed for the user from the specific context created by the user. That is, pragmatic explanations will have to interpret all the parameters of a block to dynamically generate an explanation about the settings used by the user. The pragmatic explanation is not about that type of block in general but about the specific block that was edited by the user. The benefit of this context information can be significant given that some parameters may be difficult to interpret. Users can experiment with parameters in support of comprehension. Pragmatic explanations allow blocks to be more compact, as they allow the use of compact representations, such as the arrow in Figure 10 indicating a direction to look for other agents. Experience with more verbose, AppleScript-like, representations of blocks in AgentSheets [91] suggested that they were appreciated by first time users but not liked by users with previous AgentSheets experience. The pragmatic explanation in Figure 10 is based on a dynamic tool tip-like annotation combined (Mac only) with a text-to-speech interface. An explanation produces a sentence based on the parameters of the block, annotates parts of the sentence in Karaoke sing-along style, and simultaneously makes the corresponding parameter blink (e.g., the arrow left corresponding to the "to my left" part of the sentence).

Syntactic, semantic and pragmatic explanations are not mutually exclusive. For instance, AgentSheets also has a traditional command help system providing generic

information including examples about blocks in addition to the pragmatic explanation. Blocks programming aimed at novices should provide all three levels of explanations.

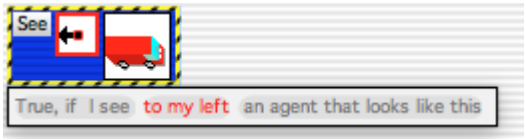


Figure 10. Pragmatic Explanation in AgentSheet.

Pragmatic explanations should include program context. For instance, to understand how a condition is used in context, one can select a rule, an IF/THEN statement containing the condition. Using the pragmatic explain function will produce an explanation for the entire rule including all of its conditions and actions but also including potentially implicit aggregation assumptions (Figure 11). For instance, by default in AgentSheets, all the conditions of a rule need to be true, i.e., they are linked by a Boolean AND. The implicit ANDing of conditions is made explicit in the explanation text, which is read out through text-to-speech interfaces producing sentences such as "If <condition 1> and <condition 2> then …". As each condition or action is explained, it is selected and animated in a Karaoke style highlighting each parameter and its corresponding text explanation. Rules are tested top to bottom and actions are executed top to bottom. The explanation can make these kinds of assumptions explicit to a novice user.

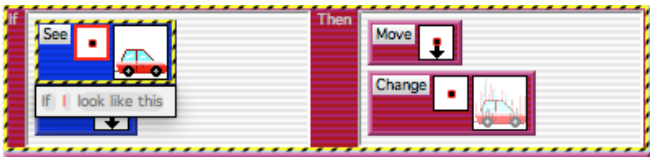


Figure 11. Explanation of an IF/THEN rule making a car fall down when there is nothing below the car.

Syntonic (the projection of oneself into something or someone else) explanations can help users to assume the perspective of the object to be programmed [92]. Body-syntonicity, a term suggested by Papert [93], describes experiences that are related to one's knowledge and sense about one's body. In the context of Logo turtle programming, Papert surmised that when students can project themselves into the turtle they would experience fewer problems with programming it. In my work with AgentSheets, I found confusion about perspective was often the source of programming problems. For instance, when programming a collision between a car and a frog in a Frogger-like game, students would often put code that was supposed to be in the frog into the car and vice versa. A syntonic approach tries to compel students to *become the frog* when they program it and to become the car when they program the car. For similar reasons, some science teachers introduced role-play games in their gym class for the student to experience being the car and the frog. I found that some degree of syntonicity could be induced by using explanatory language worded in ways suggesting to be the object to be programmed. A non-syntonic explanation of the condition in Figure 10 could be "This condition is true if the agent sees another agent looking like this to its left." The syntonic explanation, in contrast, suggests projection of the programmer into the object to be programmed by employing terms such as "I" and "my" resulting in "True, if I see to my left an agent that looks like this."

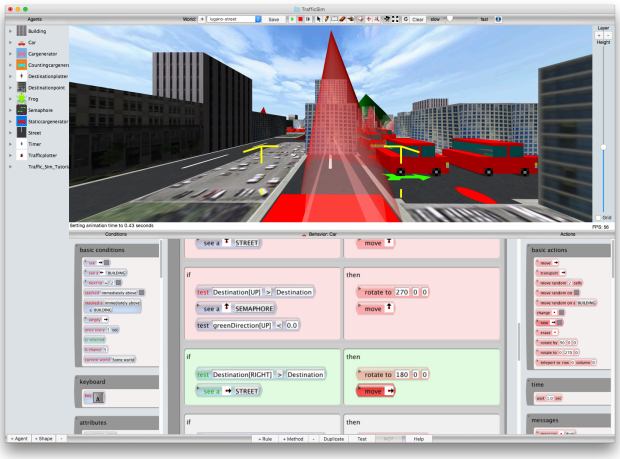


Figure 12. AgentCubes putting car into first person mode.

Programming environments can actively support body syntonicity through camera perspectives. Alice [30], for instance, does this by using coordinate systems that are object-relative. AgentCubes, as 3D Computational Thinking Tool, moves one step beyond the AgentSheets syntonic explanations by literally allowing the programmer to assume the perspective of agents to be programmed through camera operations. Every agent in AgentCubes can be selected and be set into first person camera mode. This can be done too in other 3D tools but typically requires more than a just selecting an agent and pressing the first person button. In Alice, a user has to write a simple program to set the so-called vehicle of the camera to be the object to be set into first person. For instance, in a Frogger-like game, the programmer can become one of the objects that move, such as the frog or the car (Figure 12), but also part of the scenery, such as agents representing the road or the river. The programmer will now see through the eyes of the agent. When the agent moves and turns then the camera will move and rotate with the agent. This can result in body syntonicity, helping programmers to negotiate intricacies of nested coordinate systems.

5.2. Conversational Programming: Help Predict the Future Proactively

Although computers have become incredibly powerful, debugging programs is still an arduous task. Imagine that a programmer is working on a game or simulation based on many objects, but the program is not behaving correctly and requires debugging. Pea [94] conceptualizes the process of debugging as "systematic efforts to eliminate discrepancies between the intended outcomes of a program [the program we want] and those brought through the current version of the program [the program we have]." There is a rich body of

research exploring debugging and developing highly sophisticated debugging tools. For instance, with the ZStep system, Lieberman has explored an approach to locate bugs in large code bases [95]. However, most of these tools are aimed at professional programmers and not at end-user programmers [7].

The computer, of course, cannot read the mind of users to access the programs they want. If it could there would be no need for the user to write a program to begin with. However, consistent with the notion of pragmatics, the computer can show what code means in particular situation. Visualizing the pragmatics of code, i.e., the program you have, users may be able to perceive discrepancies to the program they want.

Debugging tools for end-user programmers need to be simplified and should focus on strategies either preventing bugs or at least minimizing the time between creating a bug and being able to experience its consequences. The debugging of blocks programs can be supported at the syntactic, semantic and pragmatic levels.

Syntax: Fortunately, little work is required at the syntactic level because in most cases it can be reasonably safely assumed that programs are syntactically correct.

Semantics: Most blocks programming languages, including Scratch and AgentSheets, provide the affordance of testing blocks individually. Actions can be executed to explore their effects. Conditions can be tested to see if they are true or false. Live Programming [96-98], also found in most blocks programming languages, enables users to experience the outcome of a program by changing in real time – live – a running program.

Pragmatics: Applying the notion of pragmatics from natural languages, "The study of what words mean in particular situations," to programming languages results in the study of what programs, or fragments of programs, mean in particular situations. Pragmatics affordances such as Conversational Programming [99, 100] help programmers to explore the meaning of programs in the context of very specific situations. In order to establish the notion of a situation, a programming environment needs to be deeply connected to the representation of a simulation world. For instance, it must be possible for a user to arrange objects into a situation and define an operational perspective define by selecting objects. In a Pac-Man game, it must be possible for a user to select one of the ghosts in order to experience the meaning of its programming from a very specific context of being at a certain location in a maze with a Pac-Man and potentially many other ghosts.

My experience with semantic-level debugging tools is that they are best in the hands of experienced programmers who are typically not the prime audience of blocks programming. For instance, programmers used to programming environments providing Read Evaluate Print Loop (REPL) functionality found in languages such as Lisp, JavaScript and Python, understand the benefits of testing programs incrementally. Most blocks programming environments already do, or easily could, support this type incremental testing. These functions have existed in AgentSheets for over 20 years, but I have found that without highly explicit prompting, typical students and teachers, by and large, simply did not use them. The main problem is not that novices have a hard time to use debugging functions but that they do not anticipate the usefulness or even the presence of such functions. Instead, they are more likely to explore variations of their program in the hope to find a fix without planning to invoke some kind of debugging tools.

If users do not take the initiative for debugging, then computers should by becoming more proactive. After all, while users are contemplating options to remove discrepancies between the program they want and the one they have, computers, in spite of their multi-gigahertz, multi-core supercomputer capabilities, offer essentially no assistance. Conversational Programming [99, 100] is a proactive approach to harness this computational power to annotate programs with pragmatic information, i.e., the study of what the program means in a particular situation (Figure 13). The situation is described by an agent that is selected inside a complex simulation world. For instance, the user may have selected the frog inside a Frogger-like game. The situation combines all the state information, including the internal state of the frog and also the arrangement and states of all the other agents in the world. Conversational Programming is acting essentially like a proactive programming peer providing pragmatic information to the user. Even when the game is not running, Conversational Programming analyzes the program of the user-selected object in order to provide pragmatic feedback to the user by annotating that program (Figure 12 and 13).

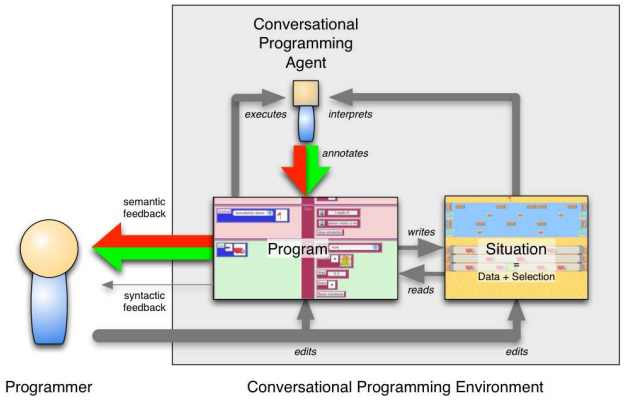


Figure 13. Conversational Programming. A Conversational Programming Agent (CPA) executes the program and provides rich, pragmatic feedback to the programmer relevant to objects of interest to the programmer.

Users can experience pragmatics by exploring various situations through the interaction with agents and observation how the program will respond differently. For instance, the user could drag the frog next to red truck (Figure 14) to observe which conditions will be true and which IF/THEN rules will fire. This helps users to understand why a certain rule does fire or why it does not. Rules that do not fire show why they do not fire, e.g., because one of their conditions is false.

The annotation includes detailed information of which condition was false resulting in the entire rule not being executable. Users can shift perspective by selecting different agents. How will the red truck react to the frog moving to its right? The Conversational Programming annotations are specific to an agent instance, not its class. If a game includes multiple frogs, then selecting different frogs will annotate the program of each frog program according to the specific situation that frog is in.

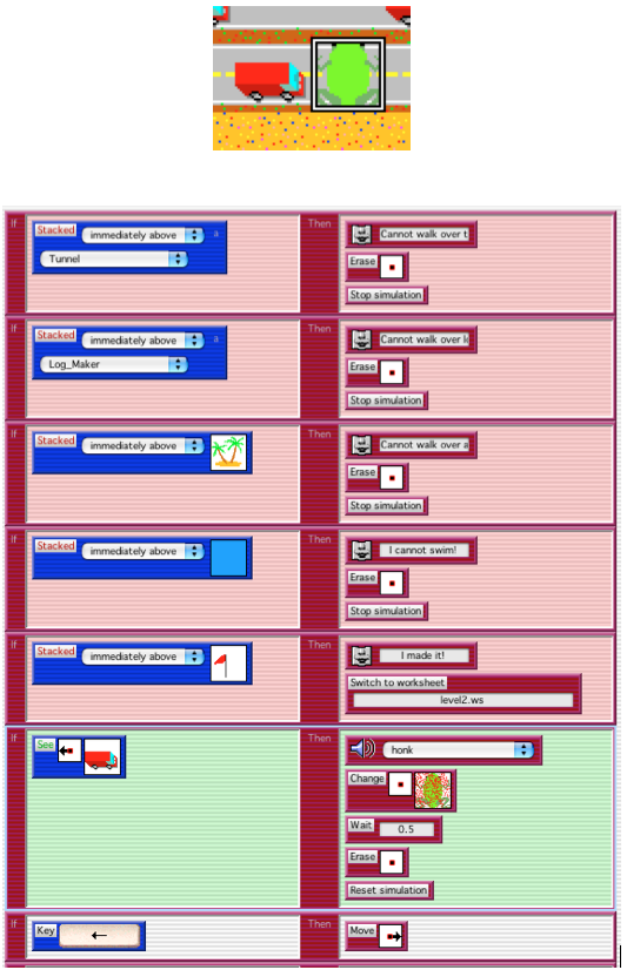


Figure 14. Conversational Programming annotates programs proactively to show the future of the simulation. In this example, the next to last rule is highlighted in green indicating that this rule will be executed. The frog is about to collide with the red truck approaching from the left. A sound will be played, the frog will turn into a bloody frog, and then the game is reset.

The proactive nature of Conversational Programming can answer questions that users have not asked yet or would not be likely to ask through more traditional, passive semantic debugging aids. Some consider this a type of pre-bugging [101] (proactive debugging tools). In essence, Conversational Programming interprets the current state of a simulation and computes the next step of the simulation from the viewpoint of an individual agent one step into the future.

Annotations may not be static because many agent behaviors include non-deterministic or time dependent code, e.g., code depending on AgentSheets/AgentCubes conditions such as *percentChance(<percentage>)* or *onceEvery (<time>)*. Employing these kinds of conditions results in animated annotations that show the frequency of a code execution path. For instance, in a complex IF/THEN/ELSE IF expression with a 10% and a 90% case, the 90% case would turn green more frequently than the 10% one. If this dynamic annotation becomes too much, users can simply deselect agents to turn annotations off.

Conversational Programming benefits from the simple rule structure of AgentSheets/AgentCubes. In contrast to the general Halting Problem for most AgenTalk programs, it can be assumed, but not determined, that the program will finish. That is, each agent evaluates a certain number of conditions resulting in the execution of a certain number of actions. For these cases, the visualization makes sense. However, even in AgenTalk, users can program recursive functions, making it impossible to determine if the program would ever halt. Nonetheless, even if it cannot be determined that a program would halt, Conversational Programming could be implemented in general purpose programming languages. This would make an interesting area for future research.

AgentCubes supports both Live Programming and Conversational Programming. When a simulation is running, because of Live Programming [96, 102], users can change the code to experience the consequence of these changes in real time. However, when a simulation is not running, because of Conversational Programming, users still see the consequences of their program changes. Conversational Programming is an extension to the Live Programming framework providing more control to users. In Live Programming, it can be difficult to navigate to a very specific program state to understand the precise effects of the program at that one state. Conversational Programming, in contrast, allows the experimentation with states by suggesting the future of the program without actually transforming the current state into the future one. In order to avoid tainting the future, or the present, this transformation needs to be done carefully, without creating any side effects.

5.3. Live Palettes: Make Programming More Serendipitous

Pragmatic support of programming should facilitate serendipitous discovery helping with the composition of blocks (affordance #1). The purpose of a programming block palette is to provide a menu of relevant language primitives to users. Syntactic, semantics, and pragmatics levels apply to suggest approaches that help users to locate relevant blocks. At the syntactic level, separate palettes, color-coding or tab based interfaces can be used to sort fundamental categories of blocks, e.g., conditions versus actions in AgentSheets/AgentCubes. At the semantics level, it typically makes sense to group blocks into commands with related meaning. At the pragmatics level, again, the main idea is to leverage the notion of context by facilitating the location of code relevant to specific situations.

Assuming that the world is a complex collection of agents, including one selected by the user, pragmatic programming block palettes transform from passive containers of blocks to live palettes serving as active exploration sandboxes. Identically to Conversational Programming, condition blocks, for instance, are annotated to show if they are true or false if tested by the currently selected agent in its particular situation. The element of serendipity comes into play through the proactive nature of Live Palettes. All conditions in the condition palette can be annotated efficiently by the computer. While some programming environments, including AgentSheets/AgentCubes and Scratch, support the evaluation of individual conditions, the reality is that few users use this feature to begin with, and out of the users employing the feature even fewer would regularly cycle through all conditions just to see which one may be true. This is also a good example of how the power of the computer can be harnessed to proactively support the programming process.

Figures 15–17 show how the conditions palette is reacting to the user's changes of the situation by moving the frog in the world. First, the frog is below the road, then the user drags it onto the first lane of the road and finally to the second lane of the road. While the user is dragging the frog around in the world the *See(left, "red truck")* and *Stacked ("immediately above")* conditions are updated by having their name turn green or red to reflect the truth value of the condition. This may provide users serendipitous information that could be relevant to design and implementation of programs based on situations that the user is exploring.

Pragmatics makes blocks in block palettes come alive in way that helps with composition of blocks (affordance #1). They are no longer just dead pieces of code but, instead, are dynamically explored as potential candidates for code that needs to be written. In other words, with Live Palettes the execution of blocks is already relevant to the decision process of the user before this user has even written any code. The annotation needs to be subtle to avoid overwhelming users with potentially irrelevant information. Simply using colors in the name of blocks has turned out to be sufficient to serve as serendipitous input without becoming intrusive.

An important concept to convey this type of pragmatic information is the responsiveness of the user interface. In his seminal work, Michotte [103] explored how people react to visual stimuli and noticed that people can actually perceive causality, even if connections between cause and effect are made up, as long as the manifestations of the effects satisfy narrow timing constraints. Similarly, we found that when blocks do react swiftly to situation changes, then humans are able to perceive a surprisingly large number of parallel changes that may result from this change. This is a good example of combining computer affordances (using parallel threads to bring block palettes to life) with human abilities (to perceive causal connections between manipulating a situation and perceiving changes) in order to move beyond syntactic support.

Figure 15. Frog is about to cross the street. Stacked (immediately above, ground) is true.; See (left, truck) is false.

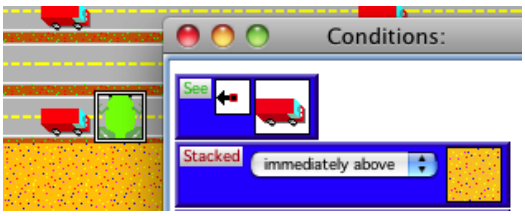


Figure 16. Frog is on street next to truck. Stacked (immediately above, ground) is false; See (left, truck) is true.

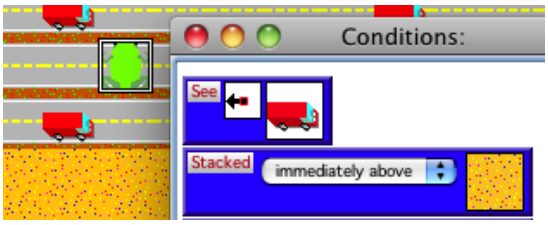


Figure 17. Frog is on street without a truck heading towards it. Stacked (immediately above, ground) is false; See (left, truck) is false

6. Computational Thinking Tools

Just as much as the research on blocks programming has not received enough attention at the language level for issues of semantics and pragmatics, there is an equally critical blind spot at the tool level. Going back to the Cognitive/Affective Challenges space (Figure 1), tools are essential to mitigate some of these challenges, but the very notion of programming tools may be too narrow, particularly in the context of Computer Science education. The goal of Computer Science education is not to write programs but to become Computational Thinkers [104]. It is gradually becoming more apparent that coding does not automatically lead to Computational Thinking. Duncan [105] summarized a pilot study with primary school students in New Zealand with

> "We had hoped that Computational Thinking skills would be taught indirectly by teaching programming and other topics in computing, but from our initial observations this may not be the case."

The Computational Thinking Process starts before writing the first line of code. Over many years, the Scalable Game Design project [51] has systematically trained teachers in Computational Thinking and evaluated the efficacy of these approaches. To adopt to the needs of Computer Science education, almost as a side effect, AgentSheets and AgentCubes have gradually shifted from being programming tools to becoming Computational Thinking Tools [37]. In contrast to traditional programming tools, Computational

Thinking Tools address a much wider spectrum of the cognitive challenges (Figure 18) and provide support for all three stages of the Computational Thinking Process (Figure 19).

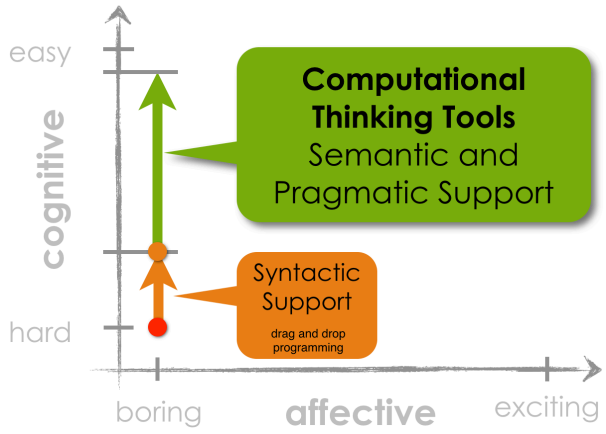


Figure 18. Computational Thinking Tools in the Cognitive/Affective Challenges space.

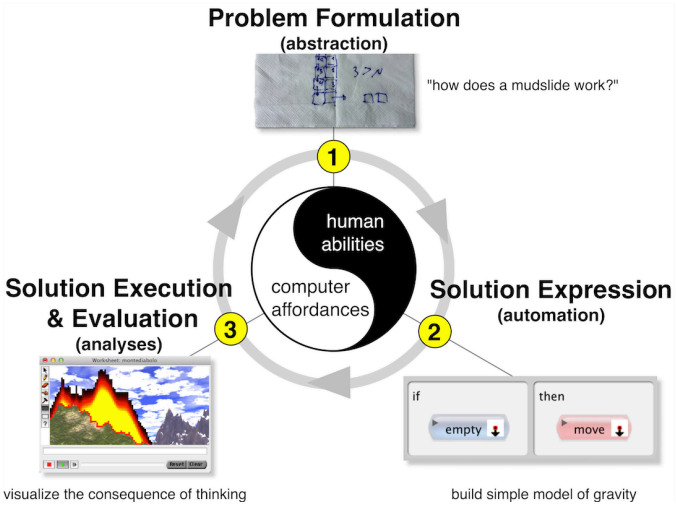


Figure 19. The Computational Thinking Process.

The term Computational Thinking (CT), popularized by Wing [104], had previously been employed by Papert in the inaugural issue of Mathematics Education [106]. Papert considered the goal of CT to *forge explicative ideas* through the use of computers. Employing computing, he argued, could result in ideas that are more accessible and powerful. Meanwhile, numerous papers [107] and reports have created many different definitions of CT. Recently, Wing followed up her seminal call for action paper with a concise operational definition of CT [108]:

> "Computational thinking is the thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer—human or machine—can effectively carry out."

Based on Wing's definition, the Computational Thinking Process can be segmented into three stages. The example in Figure 19 of a mudslide simulation is used to illustrate the three Computational Thinking Process stages.

1. ***Problem Formulation (Abstraction)***: Problem formulation attempts to conceptualize a problem verbally, e.g., by trying to formulate a question such as "How does a mudslide work?," or through visual thinking [109], e.g., by drawing a diagram identifying objects and relationships.
2. ***Solution Expression (Automation)***: The solution needs to be expressed in a non-ambiguous way so that the computer can carry it out. Computer programming enables this expression. A simple mudslide model can be expressed with just a handful of rules. The one rule in Figure 19 expresses a simple model of gravity: if there is nothing below a mud particle it will drop down.
3. ***Execution & Evaluation (Analysis)***. The solution gets executed by the computer in ways that show the direct consequences of one's own thinking. Visualizations, for instance the representation of pressure values in the mudslide as colors, support the evaluation of solutions.

The vision for Computational Thinking Tools [37] is to *support and integrate the three stages of the Computational Thinking Process*. Certainly, any kind of programming tool can be employed for Computational Thinking. End-user programming tools, for instance, are focused on the support of the solution expression by making programming more accessible. However, Computational Thinking Tools should go further by providing additional support for the problem formulation as well as the problem execution & evaluation stages of the Computational Thinking Process.

Of course, Computational Thinking can be stimulated by programming, but a trip from Chicago to Los Angeles can also be achieved by walking. Ultimately, one needs to better understand the precise goals and potential overhead of specific approaches. For instance, if the goal of programming is becoming a professional programmer versus a computational thinker, then different tools and different scaffolding [110] approaches may be necessary. When computing-skeptical STEM teachers see simple applications such as a two species ecosystem simulations turn into two hundred of lines of code, then one should not be too surprised that the adoption of programming in STEM courses is still abysmal. Blocks programming will not help either if the result is a similarly complex deeply nested Escher-esque color puzzle.

The different needs for programming in education pulls programming environments into two very different directions. *Programming tools* are general purpose programming environments that can be used for a large variety of projects, but most interesting programs quickly become elaborate because of accidental complexity [111]. Accidental complexity is complexity that cannot be traced back to the original problem. In contrast to intrinsic complexity, accidental complexity was added through a solution process involving certain tools or approaches. *Computational Thinking Tools*,

with their pronounced goal to support the Computational Thinking Process, have a more narrow range of projects, but they manage coding overhead in ways so that simple Computational Thinking can be expressed with little code. Of course, programming tools could be used for Computational Thinking or Computational Thinking Tools could be used to create general-purpose projects, but in either case the mismatch between tool and application is likely to cause excessive accidental complexity. This complexity, in turn, may simply be too much to justify educational uses.

Computational Thinking Tools and Programming Tools can be integrated technically or pedagogically. While most beginning mandatory courses with highly constrained time budgets may initially be best off to start with Computational Thinking Tools, it does often makes sense in later elective courses to switch to Programming Tools. There are many ways to technically integrate both kinds of tools. An early version of AgentSheets included an extremely powerful but also somewhat dangerous Lisp block allowing advanced users to enter arbitrary Common Lisp to be integrated into their Blocks program. With the GP system, Mönig et al. are going a different route by attempting to create a general purpose blocks programming language powerful enough to implement itself [112]. Alternatively, pedagogical integration would employ scaffolding approaches to transition from a Computational Thinking Tool to a Programming Tool without actually integrating tools technically. An example of a scaffolding approach is that AgentSheets/AgentCubes can convert blocks programs into Java and JavaScript sources respectfully. This can help students to understand how to make the transition.

AgentSheets and AgentCubes are Computational Thinking Tools. A first blocks programming prototype of AgentSheets implemented a large subset of Common Lisp concepts in order to become a programming tool. However, beyond the syntactic support of programming, which was important, it gradually became clear that, when focusing more on semantic and even pragmatic issues, it would be possible to create a conceptually different tool that could better support the problem analysis, solution formulation, and project expression stages of the Computational Thinking Process [104, 108]. A key principle of Computational Thinking Tools is that they should reduce the need for accidental complexity as much as possible. Guzdial reached a similar conclusion in the context of computing education [113] by suggesting that "If you want students to use programming to learn something else [e.g., how to author a simulation] then limit how much programming you use." The affordances related to the reduction of accidental complexity can be understood at three different levels:

Syntax: At the syntactic level, the form of a program can be controlled through disclosure mechanisms. For instance, just like the *&optional* directive in Common Lisp declares optional parameters, blocks in AgentSheets/AgentCubes can have optional parameters. The visibility of these optional parameters is controlled through disclosure mechanisms (Figure 20). Clicking a disclosure triangle will show/hide the optional parameters. Additionally, method blocks, containing rules, have disclosure triangles to show/hide their content. When the rules are hidden, a method will still show its documentation, turning the disclosure mechanism into a switch between viewing method implementation or only specification. While there are many textual programming languages that feature optional or named parameters, this concept appears not to have found widespread acceptance into other blocks programming languages, with the exceptions of blocks programming languages such as Alice [30] and Snap! The optional parameter mechanism is relevant to the notion of accidental complexity in the sense that optional parameters are typically chosen to capture less important or even qualitatively different parameters that may not be relevant to understand the main function of a program. For instance, in AgentSheets/AgentCubes, in contrast to regular parameters describing *what* should be done, optional parameters are used to describe *how* it should be done. For instance, the required direction parameter in the Move action describes which direction the agent will move, whereas the animation time and animator style parameters only describe animation details of the move transition.

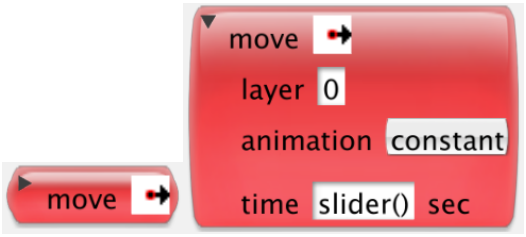


Figure 20. "move" action non disclosed (left). "move" action disclosed, showing information relevant for animation control (right).

Semantics: At the level of semantics, domain-orientation [114] is the provision of functions that reflect the needs of specific application domains. An Application Programming Interface (API) centered around related functions is an example. For instance, a set of functions to control a robot can be a domain-oriented API where the domain would be robotics. APIs are at the root of practically all domain-oriented languages, block-based or not. The main angle to reduce accidental complexity through domain-orientation is by eliminating the need build functions from the ground up. If a programming environment is frequently used to create scientific visualizations, then it should include domain-orientation offered through functions highly relevant and usable to create these visualizations.

Pragmatics: According to Webster, in the context of natural languages, pragmatics is about "the study of what words mean in particular situations." In programming, this could be modified to "the study of what code means in particular situations." For pragmatic support, Computational Thinking Tools are challenged to aid programmers to figure out what code does in specific situations. In a game context this means, for instance, that programmers should be able to manipulation the state of a game, i.e., the situation, and get tools that show potential impact on the execution of code. At the level of pragmatics, accidental complexity that gets in the way of

understanding the meaning of code in the context of specific situations should be reduced. To that end, it is important to understand the degree of *structure* of a situation. A situation in Scratch is the 2D stage containing sprites with certain locations and orientations. Similarly, a situation in Alice is a 3D world containing 3D objects. In both cases, however, the situations are essentially unstructured. The locations of 2D/3D objects have no intrinsic meaning. AgentCubes, in contrast, has a highly structured situation, i.e., the AgentCubes, which is a grid of rows, columns, and layers containing stacks of agents. The AgentCubes world provides a user interface empowering users to edit these 3D grids by placing agents, moving and copying agents similarly to how players edit Minecraft worlds. As one can witness with spreadsheets, structured situations can reduce accidental complexity dramatically because with spreadsheets no part of the user code is concerned with the maintenance of the cell structure. Spreadsheet formulas are merely capturing the functional dependence of values contained in cells without the need to understand how values are presented to users [115, 116].

The 15 squares puzzle, shown in Figure 21, is a classic children's toy that can be used to further illustrate the benefits of pragmatics. The game consists of sliding 15 numbered squares into a sorted arrangement, 1-15, in a 4 x 4 grid. Many computer program implementations of the game exist. From a Computational Thinking point of view, the core idea is simple: click a square next to the hole to make it slide into the hole.

Figure 21. 15 squares puzzle.

From a coding point of view, however, efforts can vary widely. A Python program to implement the "click to slide" functionality (e.g., [117]) quickly runs into hundreds of lines of code, not including the functionality to solve the puzzle. Similar programs, written in other programming languages such as Java and even in blocks programming languages, are of comparable size. Indeed, some blocks programming languages such as Scratch with missing class/instance object models often result in even more complex programs because of duplications [118]. The point here is not to be negative regarding programming tools, but to simply suggest that accidental complexity can be a huge overhead for Computational Thinking applications that is not automatically solved through blocks programming.

Employing AgentCubes as Computational Thinking Tool, the implementation of the 15 squares puzzle will include very little coding overhead. The "click to slide" functionality requires only four simple rules checking if there is an empty spot adjacent to the clicked square and, if so, move into that spot (Figure 22). Additionally, selecting squares activates Conversational Programming (square #11 was clicked) and highlights the fact that #11 can go left. Comparing Python to AgentCubes seems hardly fair. In AgentCubes, the notion of a grid, animations, and even numbered squares serve as situation structure dramatically reducing accidental complexity in a similar way that spreadsheets allow its users to focus on math. Additional affordances, such as the ability to access attributes of agents through spatial references, like in spreadsheets, and to express complex parallel animations, facilitate the creation of a wide range of projects from simple particle systems to games including sophisticated AI with very little code.

Figure 22. Four rules for 15 puzzle to make agent next to hole move into hole.

Each affordance has some limitations. Spreadsheets are the most frequently used end-user programming tools in the world, but they are not general purpose programming tools. Nobody would want to write a compiler with Microsoft Excel even though it may theoretically be possible. Looking at some of the incredibly elaborate designs that motivated users come up with, e.g., creating sophisticated machines by tediously arranging thousands of blocks in Minecraft, it is sometimes not clear what kinds of applications tools will afford. The 2D/3D grid structure in AgentSheets and AgentCubes is not well suited for applications requiring the computation of arbitrary trajectories. This would make it difficult to animate to trajectory of a cannonball. Even these limitations, however, have not stopped some AgentSheets users from implementing projects such as a three-body problem that would appear to be clear mismatches with the affordances of the tool.

A playable Pac-Man game (Figure 23), including endgame detection and collaborative AI [119] making ghosts collaborate with each other, can be created in just 10 rules (Figure 24). Due to collaborative diffusion, this game actually includes more sophisticated AI than the original game.

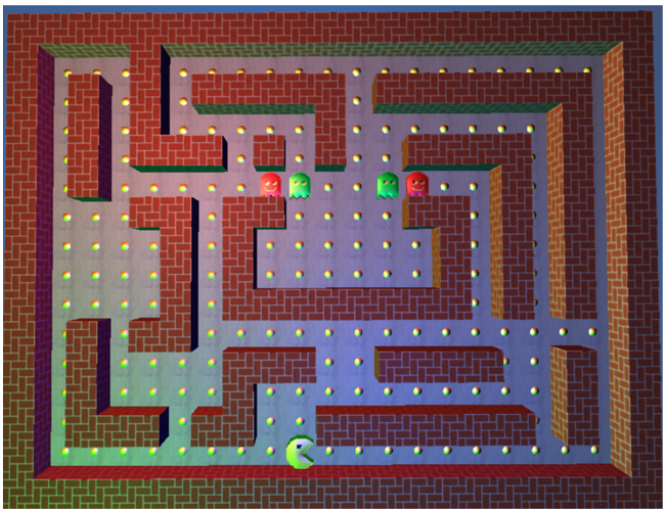


Figure 23. Pac-Man Game World.

These 10 rules implement:

- ***Collaborative Diffusion:*** [119, 120]: rule 1 of the background tile diffusing the scent of the Pac-Man as (variable `P`) and rule 2 of the pellet.
- ***Ghost hill climbing***: rule 1 of the ghost.
- ***Game won detection***: rule 1 of the Pac-Man.
- ***Game lost detection***: rule 2 of the Pac-Man.
- ***Pac-Man cursor key control***: rules 3-6 of Pac-Man.
- ***Pellets being eaten***: Pellet rule 2.

To start the diffusion the Pac-Man agent is given a `p` value of 1000. (Variables are case-insensitive, so `P` and `p` denote the same variable.) This is done through an agent attribute editor allowing users to edit arbitrary agent attributes. No programming is required to set agent attributes. They can be set and will be saved when the world containing the agent is saved.

The Flabby Bird 3D game (Figure 25) illustrates a volume scroller game (generalizing 2D side scroller games). A basic version of this game can also be created in 10 rules. This kind of game would be nearly impossible to create with 2D tools such as Scratch, but also would be difficult to create with 3D tools such as Alice.

Game design is highly motivational, but not the focus of Computational Thinking. AgentSheets and AgentCubes are not just about game design but about learning Computational Thinking patterns in ways the that they can be leveraged by students to build STEM simulations. The Predator/Prey project (Figure 26) can also built with just 10 rules to investigate the stability of ecosystems. AgentCubes includes plotting tools to visualize data and to export it to other tools such as Microsoft Excel or Google Sheets for further analysis.

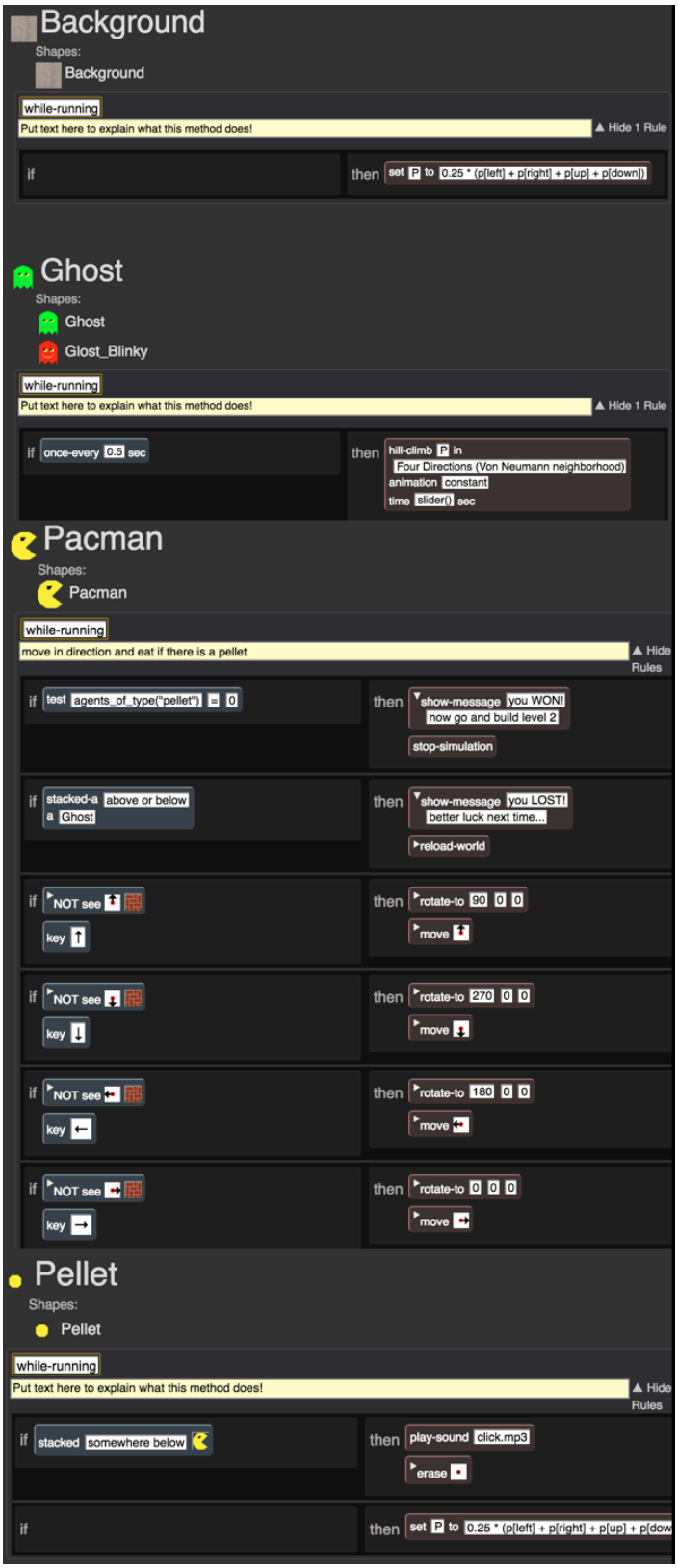


Figure 24. Complete Pac-Man game including collaborative AI in just 10 rules.

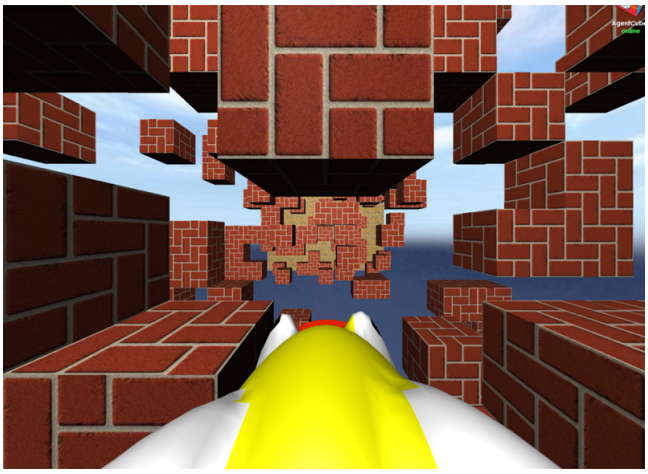


Figure 25. AgentCubes Flabby Bird 3D game.

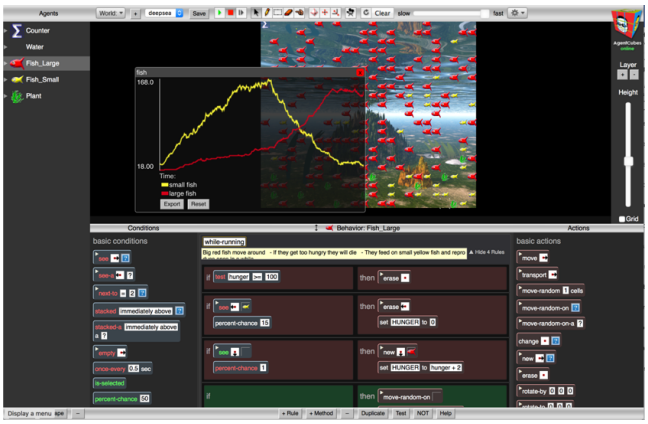


Figure 26. Predator Prey simulation including data visualization in AgentCubes.

There are downsides to Computational Thinking Tools. The scaffolding employed to make Computational Thinking Tools practical for classroom use may get in the way of general-purpose programming. This is a trade-off. Similarly, spreadsheets would not be well-suited for creating games such as billiards or Pong. And yet, spreadsheets are the number one end-user programming tool. At one point, AgentSheets did have graphs, but it felt like a confusing kitchen sink. Over time, these odd features got removed from AgentSheets. Other data structures are intrinsic to the AgentSheets/AgentCubes world. An array is a row, column, or set of layers of an AgentCubes world. In other words, the world and its structure *are* the data structure. There are 1D, 2D, 3D arrays that are similar to spreadsheets. Additionally, each cell can contain stacks of agents. As long as users can establish a conceptual match between the problem structure and the 3D row, column, layer, stacks metaphor, AgentCubes can serve as an efficient thinking and programming tool. But there are clear limits when additional functionality begins to erode affordances. Sometimes more is less.

The threshold between programming tool and a Computational Thinking Tool is not what can and cannot be done conceptually but what can be done practically from a classroom point of view. Advanced students have built sophisticated simulations of 3-body problems in AgentSheets. Using a fine grid with hundreds of thousands of agents, this can be done, but it goes against the grain of the solution structure implied by AgentSheets. Just as a scientific calculator can be built with millions of Minecraft blocks by a user with thousands of hours at his hands, these solutions could be built with Computational Thinking Tools, but they are not practical in a traditional educational context.

Just as the 3D cube with stack structures provides a spatial scaffold in the AgentSheets/AgentCubes programming language, AgenTalk is a language scaffold that removes many of the intricacies (but also affordances) of general-purpose programming languages. The rule-based nature of AgenTalk is surprisingly versatile. Rules can be grouped into methods that can be called through actions. Method calls can be recursive. Event-based programming (e.g., mouse clicks and timers) can be expressed. Cloud variables can be used to exchange values through the network to create distributed simulations. The combination of these features make it possible to cover the entire spectrum of Computational Thinking concepts, ranging from procedural abstractions over iterations through networking.

Computational Thinking Tools are specifically designed to support Computational Thinkers in schools. They scaffold the entire Computational Thinking Process. AgentSheets and AgentCubes are presented here as early examples of Computational Thinking Tools. The main point of this section is to suggest a new research direction and to illustrate the concept with a concrete starting point.

7. Conclusions

The blocks programming community, by and large, has been preoccupied with syntactic affordances of programming environments. It is time to shift research agendas towards the systematic exploration of semantic and pragmatic affordances of blocks programming. Syntactic affordances of programing languages can be compared to spell and grammar checking in word processing. This type of support is highly useful but, computationally speaking, trivial compared to the challenges ahead attempting to support users to produce meaningful programs. The most daunting challenge will be to support pragmatics, **that is the study of what code means in particular situations**. To overcome this challenge, new approaches require the combination of various promising approaches, including program analysis, program visualization, and real-time user interfaces.

A promising direction may be the exploration of what exactly situations really are in Computational Thinking Tools. In AgentCubes, situations are visible game or simulation states including complex 3D worlds that users can interact with. A situation should be a tight integration of game and program state allowing programmers to navigate fluidly in space and time from the code as well as from a world point of view. Select objects in scenes, change properties of objects, and observe the consequence on the program execution. Select

programming primitive and explore their consequent onto the world. New research will likely reconceptualize deep connections between the program state and the game world.

Twenty years ago, AgentSheets combined four key affordances to create an early form of blocks programming. After initially focusing on syntactic affordances, using AgentSheets in computer science education, I have experimented with approaches to move beyond syntax to address semantic and pragmatic obstacles. Three approaches are described: (1) Contextualized Explanations to support comprehension, (2) Conversational Programming to help predict the future proactively, and (3) Live Palettes to make programming more serendipitous. Additionally the vision of Computational Thinking Tools as a means to support Computational Thinking Processes while reducing accidental complexity emerging from coding has been outlined.

Acknowledgments

This research has been funded by the National Science Foundation (including projects EIA-0205625, DMI-0232669, DMI-0233028, DMI-0349663, SCI-0537341, IIP-0712571, DRL-0833612, IIP-0848962, IIP-1014249, IIP-112738, CNS-1138526, DMI-9761360, IIP-1345523, DMI-9901678, DRL-1312129), the National Institutes of Health, Apple, Google, the AMD Foundation, the Hasler Foundation, and the Swiss National Science Foundation. I wish to thank my advisor, my collaborators, my graduate students, and all the teachers and students for their amazing support in the last 20 years.

References

[1] "Women Who Choose Computer Science – What Really Matters, The Critical Role of Encouragement and Exposure," Google Report, May 26, 2014.

[2] A. Repenning, C. Smith, B. Owen, and N. Repenning, "AgentCubes: Enabling 3D Creativity by Addressing Cognitive and Affective Programming Challenges," presented at the *World Conference on Educational Media and Technology, EdMedia 2012*, Denver, Colorado, USA, 2012, pp. 2762-2771.

[3] A. Repenning, A. Basawapatna, D. Assaf, C. Maiello, and N. Escherle, "Retention of Flow: Evaluating a Computer Science Education Week Activity," *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*, Memphis, Tennessee, 2016, pp. 633-638.

[4] A. Repenning and A. Basawapatna, "Drops and Kinks: Modeling the Retention of Flow for Hour of Code Style Tutorials," *Proceedings of the 11th Workshop in Primary and Secondary Computing Education (WiPSCE '16)*, Münster, Germany, 2016, pp. 76-79.

[5] A. Basawapatna and A. Repenning, "Employing Retention of Flow to Improve Online Tutorials," *Proceedings of the 48th ACM Technical Symposium on Computing Science Education (SIGCSE '17)*, Seattle, Washington, USA, 2017, pp 63-68.

[6] D. Webb, A. Repenning, and K. Koh, "Toward an Emergent Theory of Broadening Participation in Computer Science Education," in *Proceedings of the 43rd ACM Technical Symposium on Computing Science Education (SIGCSE '12)*, Raleigh, North Carolina, USA, 2012, pp..173-178.

[7] H. Lieberman, F. Paternò, and V. Wulf, Eds., *End User Development*. Springer, 2006, 492 Pages

[8] T. Toffoli and N. Margolus, *Cellular Automata Machines*. Cambridge, MA: MIT Press, 1987.

[9] M. Resnick, "StarLogo: an environment for decentralized modeling and decentralized thinking," *Conference Companion on Human Factors in Computing Systems (CHI '96)*, Vancouver, British Columbia, Canada, 1996, pp. 11-12.

[10] A. Repenning, "Making Programming Accessible and Exciting," *IEEE Computer*, vol. 18, pp. 78-81, 2013.

[11] N. Shu, *Visual Programming*. New York: Van Nostrand Reinhold Company, 1988.

[12] B. Bell and C. Lewis, "ChemTrains: A Language for Creating Behaving Pictures," in *IEEE Workshop on Visual Languages*, Bergen, Norway, 1993, pp. 188-195.

[13] G. W. Furnas, "New Graphical Reasoning Models for Understanding Graphical Interfaces," *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '91)*, New Orleans, LA, 1991, pp. 71-78.

[14] R. Kirsch, A., "Computer Interpretation of English and Text and Picture Patterns," *IEEE Transactions on Electronic Computers*, vol. 13, pp. 363-376, 1964.

[15] A. Repenning, "Creating User Interfaces with Agentsheets," in *Proceedings of the 1991 Symposium on Applied Computing*, Kansas City, MO, 1991, pp. 190-196.

[16] A. Repenning, "Repräsentation von graphischen Objekten," Asea Brown Boveri Research Center, Artificial Intelligence group, Daetwill 5405, Switzerland, Research Report CRB 87-84 C, June, 1987.

[17] G. K. Zipf, *Human Behavior and the Principle of Least Effort*. New York: Hafner Publishing Company, 1972.

[18] A. Repenning, "Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments," Department of Computer Science, University of Colorado at Boulder, 1993.

[19] J. C. Spohrer, "ATG Education Research — The Authoring Tools Thread," Apple Computer, 1998.

[20] D. C. Smith, A. Cypher, and J. Spohrer, "KidSim: Programming Agents Without a Programming Language," *Communications of the ACM*, vol. 37, pp. 54-68, 1994.

[21] K. Schneider and A. Repenning, "Deceived by Ease of Use: Using Paradigmatic Applications to Build Visual Design," in *Proceedings of the 1995 Symposium on Designing Interactive Systems*, Ann Arbor, MI, 1995, pp. 177-188.

[22] J. Gindling, A. Ioannidou, J. Loh, O. Lokkebo, and A. Repenning, "LEGOsheets: A Rule-Based Programming, Simulation and Manipulation Environment for the LEGO Programmable Brick," *Proceedings of the 11th International IEEE Symposium on Visual Languages,* Darmstadt, Germany, 1995, 172-179.

[23] A. Repenning, "Bending Icons: Syntactic and Semantic Transformation of Icons," in *Proceedings of the 1994 IEEE Symposium on Visual Languages*, St. Louis, MO, 1994, pp. 296-303.

[24] A. Repenning and C. Perrone, "Programming by Analogous Examples," *Communications of the ACM*, vol. 43, pp. 90-97, 2000.

[25] B. Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," in *Human-Computer Interaction: A multidisciplinary approach*, R. M. Baecker and W. A. S. Buxton, eds., Toronto: Morgan Kaufmann Publishers, Inc., 1989, pp. 461-467.

[26] E. P. Glinert, "Towards "Second Generation" Interactive, Graphical Programming Environments," in *IEEE Computer Society Workshop on Visual Languages*, Dallas, 1986, pp. 61-70.

[27] A. Repenning and J. Ambach, "Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing," *Proceedings of the 1996 IEEE Symposium of Visual Languages*, Boulder, CO, 1996, 102-109.

[28] B. Freudenberg, Y. Ohshima, and S. Wallace, "Etoys for One Laptop Per Child," in *Proceedings of the 2009 Seventh international Conference on Creating, Connecting and Collaborating Through Computing*, Kyoto, Japan, 2009, pp. 57-64.

[29] A. Kay, "Squeak Etoys, Children & Learning," Viewpoints Research Institute, VPRI Research Note RN-2005-001, 2005.

[30] M. Conway, S. Audia, T. Burnette, D. Cosgrove, K. Christiansen, R. Deline, J. Durbin, R. Gossweiler, S. Koga, C. Long, B. Mallory, S. Miale, K. Monkaitis, J. Patten, J. Pierce, J. Shochet, D. Staack, B. Stearns, R. Stoakley, C. Sturgill, J. Viega, J. White, G. Williams, and R. Pausch, "Alice: Lessons Learned from Building a 3D System For

Novices," in *Proceedings of the CHI 2000 Conference on Human Factors in Computing Systems*, The Hague, Netherlands, 2000, 486-493.

[31] M. Resnick, J. Maloney, A. Monroy-Hernánde, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: Programming for All," *Communications of the ACM*, vol 52, no. 1, Nov, 2009, pp. 60-67

[32] A. Repenning, D. C. Webb, C. Brand, F. Gluck, R. Grover, S. Miller, H. Nickerson, and M. Song, "Beyond Minecraft: Facilitating Computational Thinking through Modeling and Programming in 3D," I*EEE Computer Graphics and Applications*, vol. 34, pp. 68-71, May-June 2014.

[33] A. Ioannidou, A. Repenning, and D. Webb, "AgentCubes: Incremental 3D End-User Development," *Journal of Visual Language and Computing*, vol. 20, no. 4, Aug., 2019, pp. 236-251.

[34] A. Ioannidou, A. Repenning, and D. Webb, "Using Scalable Game Design to Promote 3D Fluency: Assessing the AgentCubes Incremental 3D End-User Development Framework," in *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '08)*, Herrsching am Ammersee, Germany, 2008, 47-54.

[35] A. Repenning and A. Ioannidou, "AgentCubes: Raising the Ceiling of End-User Development in Education through Incremental 3D," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, Brighton, United Kingdom, 2006, pp. 27-34.

[36] A. Repenning, "Inflatable Icons: Diffusion-based Interactive Extrusion of 2D Images into 3D Models," *The Journal of Graphical Tools*, vol. 10, pp. 1-15, 2005.

[37] A. Repenning, A. Basawapatna, and N. Escherle, "Computational Thinking Tools," presented at the *IEEE Symposium on Visual Languages and Human-Centric Computing*, Cambridge, UK, 2016.

[38] W. F. Finzer and L. Gould, "Rehearsal world: programming by rehearsal," in [69], pp. 79-100.

[39] A. Repenning, A. Ioannidou, M. Rausch, and J. Phillips, "Using Agents as a Currency of Exchange between End-Users," in *Proceedings of]WebNET 98 World Conference of the WW, Internet, and Intranet*, Orlando, FL, 1998, pp. 762-767.

[40] A. Repenning and J. Ambach, "The Agentsheets Behavior Exchange: Supporting Social Behavior Processing," *CHI '97 Extended Abstracts on Human Factors in Computing Systems*, Atlanta, Georgia, 1997, 26-27.

[41] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys*, vol. 37, no. 2, Jun, 2005, pp. 83-137, 2005.

[42] J. Trower and J. Gray, "Blockly Language Creation and Applications: Visual Programming for Media Computation and Bluetooth Robotics Control," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, Kansas City, Missouri, USA, 2015, p. 5.

[43] H. Lieberman, "Dominoes and Storyboards: Beyond Icons on Strings," in *Proceedings of the 1992 IEEE Workshop on Visual Languages*, 1992, pp. 65-71.

[44] M. Bienkowski, E. Snow, D. Rutstein, and S. Grover, "Assessment Design Patterns for Computational Thinking Practices in Secondary Computer Science: A First Look," SRI International, 2015.

[45] K. Kyu Han, A. Basawapatna, H. Nickerson, and A. Repenning, "Real Time Assessment of Computational Thinking," presented at the *Visual Languages and Human-Centric Computing* (VL/HCC), Melbourne, 2014, pp. 49-52.

[46] K. H. Koh, H. Nickerson, A. Basawapatna, and A. Repenning, "Early validation of Computational Thinking Pattern Analysis," in *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITICSE)*, Uppsala, Sweden, 2014, pp. 213-218.

[47] K. H. K. Ashok Basawapatna, Alexander Repenning, David C. Webb, Krista Sekeres Marshall, "Recognizing Computational Thinking Patterns," in *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE)*, Dallas, Texas, USA, 2011, pp. 245-250.

[48] K. H. Koh, A. Basawapatna, V. Bennett, and A. Repenning, "Towards the Automatic Recognition of Computational Thinking for Adaptive Visual Language Learning," in *Conference on Visual Languages and Human Centric Computing (VL/HCC)*, Madrid, Spain, 2010, pp. 59-66.

[49] A. Basawapatna, A. Repenning, K. H. Koh, and M. Savignano, "The Consume-Create Spectrum: Balancing Convenience and Computational Thinking in STEM Learning," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*, Atlanta, GA, USA, 2014, pp. 659-664.

[50] K. Howland and J. Good, "Learning to Communicate Computationally with Flip: A Bi-modal Programming Language for Game Creation," *Computers & Education*, vol. 80, 2015, pp. 224-240.

[51] A. Repenning, D. C. Webb, K. H. Koh, H. Nickerson, S. B. Miller, C. Brand, I. H. M. Horses, A. Basawapatna, F. Gluck, R. Grover, K. Gutierrez, and N. Repenning, "Scalable Game Design: A Strategy to Bring Systemic Computer Science Education to Schools through Game Design and Simulation Creation," *Transactions on Computing Education (TOCE)*, vol. 15, no. 2, Apr. 2015, pp. 1-31.

[52] K. H. Koh, A. Repenning, H. Nickerson, Y. Endo, and P. Motter, "Will it Stick? Exploring the Sustainability of Computational Thinking Education Through Game Design," in *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13*), Denver, Colorado, USA, 2013, pp. 597-602.

[53] S.-K. Chang, *Principles of Visual Programming Systems*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[54] I. E. Sutherland, "Sketchpad: A Man-machine Graphical Communication system," in *Proceedings of the SHARE design automation workshop*, 1964, pp. 6.329-6.346.

[55] W. R. Sutherland, "The On-line Specification of Computer Procedures," MIT, Department of Electrical Engineering, Ph.D. Thesis, 1966.

[56] T. O. Ellis, J. F. Heafner, and W. L. Sibley, "The Grail Project: An Experiment in Man-Machine Communications," RAND Corporation, Memorandum RM-5999-ARPA, 1969.

[57] M. R. Minsky, "Manipulating Simulated Objects with Real-world Gestures Using a Force and Position Sensitive screen," *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, Jan., 1984. pp. 195-203.

[58] E. P. Glinert and S. L. Tanimoto, "Pict: An Interactive Graphical Programming Environment," *IEEE Computer*, vol 17. no. 11, , pp. 265-283, Nov. 1984.

[59] R. Lutze, "The Gestalt Analysis of Programs," in *Visualization in Programming* (Lecture Notes in Computer Science 282), P. Gorny and M. J. Tauber, Eds., ed. Berlin: Springer-Verlag, 1986, pp. 24-36.

[60] V. Koushik and C. Lewis, "A Nonvisual Interface for a Blocks Language," presented at the *Psychology of Programming Interest Group*, Cambridge, UK, 2016.

[61] S. Lerner, S. R. Foster, and W. G. Griswold, "Polymorphic Blocks: Formalism-Inspired UI for Structured Connectors," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, Seoul, Republic of Korea, 2015, pp. 3063-3072.

[62] M. Vasek, "Representing expressive types in blocks programming languages," B.S., Wellesley College, Honors Thesis, Wellesley College, 2012.

[63] R. Perlman, "Using Computer Technology to Provide a Creative Learning Environment for Preschool Children," MIT, Computer Science and Artificial Intelligence Lab (CSAIL), Artificial Intelligence Lab Publications, Boston, MA 1976.

[64] J. J. Anderson, "ChipWits: Bet you Can't Build Just One," *Creative Computing*, pp. 76-79, 1985.

[65] A. Begel, "LogoBlocks: A Graphical Programming Language for Interacting with the World," Electrical Engineering and Computer Science Department, MIT, Boston, MA, 1996.

[66] A. diSessa and H. Abelson, "Boxer: a Reconstructible Computational Medium," *Communications of the ACM*, vol. 29, no. 9, pp. 859 - 868. Sep. 1986.

[67] K. M. Kahn and V. A. Saraswat, "Complete Visualizations of Concurrent Programs and their Executions," in *Proceedings of the 1990 IEEE Workshop on Visual Languages*, 1990, pp. 7-15.

[68] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick, "Scratch: A Sneak Preview," in *Second International Conference on Creating, Connecting, and Collaborating through Computing*, Kyoto, Japan, 2004, pp. 104-109.

[69] A. Cypher (ed.), *Watch What I Do: Programming by Demonstration*. Cambridge, MA: MIT Press, 1993.

[70] D. C. Halbert, "Programming by Example," Xerox Office Systems Division, Technical Report OSD-T8402, 1984.

[71] D. C. Smith, "PYGMALION: A Creative Programming Environment," Thesis. Stanford Artificial Intelligence Laboratory Memo, AIM 260, Stanford University, 1975.

[72] P. T. Cox, F. R. Giles, and T. Pietrzykowski, "Prograph: a Step Towards Liberating Programming from Textual Conditioning," in *IEEE Workshop ib Visual Languages*, 1989, pp. 150-156.

[73] J. Poswig, K. Teves, G. Vrankar, and C. Moraga, "VisaVis – Contributions to Practice and Theory of Highly Interactive Visual Languages," in *Proceedings of the IEEE Workshop on Visual Languages*, 1992, pp. 155-161.

[74] B. Nardi, *A Small Matter of Programming*. Cambridge, MA: MIT Press, 1993.

[75] D. D. Hils, "Datavis: a Visual Programming Language for Scientific Visualization," presented at the *Proceedings of the 19th annual Conference on Computer Science (CSC '91)*, San Antonio, Texas, USA, 1991, pp. 439-448.

[76] D. D. Hils, "Visual Languages and Computing Survey: Data Flow Visual Programming Languages," *Journal of Visual Languages and Computing*, pp. 69-101, 1992.

[77] M. A. Najork and S. M. Kaplan, "The CUBE Languages," in *Proceedings of the IEEE workshop on Visual Languages*, 1991, pp. 218-224.

[78] G. Fischer and A. C. Lemke, "Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication," *Human-Computer Interaction*, vol. 3, no. 3, pp. 179-222, 1988.

[79] C. Lombardi and M. Weksler, "Duo Ex Machina: Tinkering with Sierra's The Incredible Machine," *Computer Gaming World*, vol. 105, pp. 52-52, 1993.

[80] M. S. El-Nasr and B. K. Smith. "Learning through game modding". *Computers in Entertainment* 7, 2006.

[81] E. Klopfer, H. Scheintaub, W. Huang, and D. Wendel, "StarLogo TNG," in *Artificial Life Models in Software*, M. Komosinski and A. Adamatzky, eds., London: Springer, 2009, pp. 151-182.

[82] A. Repenning, A. Ioannidou, and J. Zola, "AgentSheets: End-User Programmable Simulation," *Journal of Artificial Societies and Social Simulation*, vol. 3, 2000.

[83] A. Repenning and T. Sumner, "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages," *IEEE Computer*, vol. 28, no. 3, pp. 17-25, Mar. 1995.

[84] A. Repenning and W. Citrin, "Agentsheets: Applying Grid-Based Spatial Reasoning to Human-Computer Interaction," in *Proceedings of the IEEE Workshop on Visual Languages*, Bergen, Norway, 1993, pp. 77-82.

[85] A. Repenning, "Agentsheets: A Tool for Building Domain-Oriented Visual Programming Environments," in *INTERCHI '93, Conference on Human Factors in Computing Systems*, Amsterdam, NL, 1993, pp. 142-143.

[86] A. Repenning, D. Webb, and A. Ioannidou, "Scalable Game Design and the Development of a Checklist for Getting Computational Thinking into Public Schools," in *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)*, Milwaukee, WI, 2010, 265-269.

[87] N. Escherle, S. Ramirez-Ramirez, A. Basawapatna, D. Assaf, A. Repenning, C. Maiello, Y. Endo, and J. Nolazco-Florez, "Piloting Computer Science Education Week in Mexico," in *Proceedings of the 47th ACM Technical Symposium on Computer Science Education (SIGCSE '16)*, Memphis, Tennessee, 2016, pp. 431-436.

[88] N. Escherle, D. Assaf, A. Basawapatna, C. Maiello, and A. Repenning, "Launching Swiss Computer Science Education Week," in *Proceedings of the 10th Workshop in Primary and Secondary Computing Education (WIPSCE)*, London, U.K., 2015, pp. 11-16.

[89] C. DiGiano and M. Eisenberg, "Self-disclosing Design Tools: A Gentle Introduction to End-User Programming," in *Proceedings of the 1st Conference on Designing Interactive Systems: Processes, Practices, Methods, & Techniques (DIS '95)*, Ann Arbor, Michigan USA, 1995, pp. 189-197.

[90] S. Turkle, *Evocative Objects: Things We Think With*. Cambridge, USA: MIT Press, 2007.

[91] C. Rader, G. Cherry, C. Brand, A. Repenning, and C. Lewis, "Principles to Scaffold Mixed Textual and Iconic End-User Programming Languages," in *Proceedings of the 1998 IEEE Symposium of Visual Languages*, Nova Scotia, Canada, 1998, pp. 187-194.

[92] S. Watt, "Syntonicity and the Psychology of Programming," in *Proceedings of the Tenth Annual Meeting of the Psychology of Programming Interest Group, Milton Keenes*, UK, 1998, pp. 75-86.

[93] S. Papert, *Mindstorms: Children, Computers and Powerful Ideas*. New York: Basic Books, 1980.

[94] R. D. Pea, "Chameleon in the Classroom: Developing Roles for Computers, Logo Programming and Problem Solving," presented at the *American Educational Research Association Symposium*, Montreal, Canada, 1983.

[95] H. Lieberman, "Steps Toward Better Debugging Tools for LISP," presented at the *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, USA, 1984, pp. 247-255.

[96] S. McDirmid, "Usable Live Programming," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! '13)*, Indianapolis, Indiana, 2013, pp. 53-62.

[97] S. Burckhardt, M. Fahndrich, P. d. Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato, "It's Alive! Continuous Feedback in UI Programming," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*, Washington, USA, 2013, pp. 95-104.

[98] S. McDirmid, "Living it Up with a Live Programming Language," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPLSA '07)*, Montreal, Quebec, Canada, 2007, pp. 623-638.

[99] A. Repenning, "Conversational Programming: Exploring Interactive Program Analysis," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! '13)*, Indianapolis, Indiana, USA, 2013, pp. 63-74.

[100] A. Repenning, "Making Programming more Conversational," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Pittsburgh, PA, USA, 2011, pp. 191-194.

[101] M. Telles and Y. Hsieh, *The Science of Debugging*. Scottsdale: Coriolis Group Books, Scottsdale AZ, USA, 2001.

[102] S. H. Cameron, D. Ewing, and M. Liveright, "DIALOG: a Conversational Programming System with a Graphical Orientation," *Communications of the ACM*, vol. 10, pp. 349-357, 1967.

[103] A. Michotte, *The Perception of Causality*. Andover, MA: Methuen, 1962.

[104] J. M. Wing, "Computational Thinking," *Communications of the ACM*, vol. 49, no. 3, pp. 33-35, Mar. 2006.

[105] C. Duncan and T. Bell, "A Pilot Computer Science and Programming Course for Primary School Students," in *Proceedings of the Workshop in Primary and Secondary Computing Education (WiPCSE '15)*, London, United Kingdom, 2015, pp. 39-48.

[106] S. Papert, "An Exploration in the Space of Mathematics Educations," I*nternational Journal of Computers for Mathematical Learning*, vol. 1, pp. 95-123, 1996.

[107] S. Grover and R. Pea, "Computational Thinking in K–12: A Review of the State of the Field," *Educational Researcher*, vol. 42, pp. 38-43, 2013.

[108] J. M. Wing, "Computational Thinking Benefits Society," in *40th Anniversary Blog of Social Issues in Computing* vol. 2014, J. DiMarco, ed., University of Toronto, 2014 [Online.] http://socialissues.cs.toronto.edu/2013/01/40th-anniversary/

[109] R. Arnheim, *Visual Thinking*. Berkley: University of California Press, 1969.

[110] B. J. Reiser, “Scaffolding Complex Learning: The Mechanisms of Structuring and Problematizing Student Work,” *Journal of the Learning Sciences*, vol. 13, no. 3, pp. 273–304, 2004.

[111] F. P. Brooks Jr., “No Silver Bullet: Essence and Accidents of Software Engineering,” *IEEE Computer*, vol 20, no. 4. pp. 10-19, 1987.

[112] J. Monig, Y. Ohshima, and J. Maloney, “Blocks at Your Fingertips: Blurring the Line between Blocks and Text in GP,” in *IEEE Blocks and Beyond Workshop*, 2015, pp. 51-53.

[113] M. Guzdial, Learner-Centered “Design of Computing Education: Research on Computing for Everyone,” *Synthesis Lectures on Human-Centered Informatics*: Morgan & Claypool Publishers, 2015.

[114] G. Fischer, “Domain-Oriented Design Environments,” in *Automated Software* Engineering. vol. 1, ed Boston, MA: Kluwer Academic Publishers, 1994, pp. 177-203.

[115] C. Lewis and G. M. Olson, “Can Principles of Cognition Lower the Barriers to Programming?,” in *Empirical Studies of Programmers: Second Workshop*, Norwood, NJ, 1987, pp. 248-263.

[116] C. Lewis, “NoPumpG: Creating Interactive Graphics with Spreadsheet Machinery,” Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado, Technical Report CU-CS-372-87, August, 1987.

[117] A. Sweigart, *Invent Your Own Computer Games with Python: A Beginner's Guide to Computer Programming in Python*, 2010.

[118] F. Hermans and E. Aivaloglou, “Do Code Smells Hamper Novice Programming?,” Delft University of Technology, Software Engineering Research Group, Delft University of Technology Report TUD-SERG-2016-06, 2016.

[119] A. Repenning, “Collaborative Diffusion: Programming Antiobjects,” in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06*), Portland, Oregon, 2006, pp. 574-585.

[120] A. Repenning, “Excuse me, I need better AI!: employing collaborative diffusion to make game AI child's play,” in *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames (Sandbox '06)*, Boston, Massachusetts, 2006, pp. 169-178.

How Block-based Languages Support Novices

A Framework for Categorizing Block-based Affordances

David Weintrop
UChicago STEM Education
University of Chicago
Chicago, Illinois, USA
dweintrop@uchicago.edu

Uri Wilensky
Center for Connected Learning and
Computer-based Modeling
Learning Sciences and Computer Science
Northwestern University
Evanston, Illinois, USA
uri@northwestern.edu

Abstract *The ability to express ideas in a computationally meaningful way is becoming increasingly important in our technological world. In response to the growing importance of computational literacy skills, new intuitive and accessible programming environments are being designed. This paper presents a framework for classifying the ways that block-based introductory programming environments support novices. We identify four distinct roles that these graphical languages play in the activity of programming: (1) serving as a means for expressing ideas to the computer, (2) providing a record of previously articulated intentions, (3) acting as a source of ideas for construction, and (4) mediating the meaning-making process. Using data from a study of novices programming with a custom designed block-based language, we provide examples of each role along with a discussion of the design implications of these findings. In doing so, we contribute to our understanding of the relationship between the design of programming representations and their ability to support computational literacy. The paper concludes with a discussion of the potential for this framework beyond block-based environments to programming languages more broadly.*

Keywords *Block-based Programming, Cognition, Design, Learning*

1. Introduction

The skills and practices associated with computational thinking are critically important for learners in order to be full participants in our increasingly technological world [1–7]. Central to computational thinking is the ability to encode ideas into representations that can be executed by a computational device. Through mastering these skills, computational thinking can be infrastructural to learning across diverse domains and open pathways to new forms of expression. In this way, we align computational thinking with diSessa's [1] notion of computational literacy, which envisions a citizenry that are both consumers and producers of computational artifacts.

A key component for both comprehension and generation of computational artifacts is the representational infrastructure that mediates these processes. This historically has taken the form of text-based programming languages, but can also include visual programming languages or graphical interfaces that support the assembly of instructions [8], or applications designed to interpret drawings or glyphs created by the user [9]. Block-based programming languages in particular are becoming increasingly common in introductory programming contexts [10].

Each of these representational systems achieves the same ends (defining instructions for a computer to follow), but does so through very different means that directly influence the process. The characteristics of a representational system, including the visual presentation, syntax, relation to other representational systems, and expressive power, have a direct influence on how one goes about accomplishing a task and the resulting understanding that develops from that experience [1, 11, 12]. With the emergence of new forms of end-user programming languages and human-centered interfaces, providing a framework for categorizing the ways that representational tools facilitate these ends is important as it provides structure to understand the various roles that features of introductory programming languages play. Further, it can be used to improve the current generation of programming tools and inform the design of the next generation of expressive computational media.

In this paper, we present a framework for categorizing the various ways novices use block-based programming languages to express their ideas in a computational medium. Through analyzing novices playing a program-to-play constructionist video game, we identify four distinct usages of the programming language: (1) serving as a means for expressing ideas to the computer, (2) providing a record of previously articulated intentions, (3) acting as a source of ideas for construction, and (4) mediating the meaning-making process. This paper situates these roles in a larger framework and presents vignettes from a study to demonstrate what each use looks like when enacted. The contribution of this work is the development an empirically grounded framework that can be used to structure the study of block-based programing languages, advance our understanding of the learning that takes place through their use, and inform the design of future programming tools and expressive computational technologies. In the conclusion of the paper, we expand our focus to include non-block-based programming languages and discuss the potential broader applicability of the presented framework.

DOI reference number: 10.18293/VLSS2017-006

2. Orienting Framework

The constitutive role of language and tools on cognition has long been a topic of research. A central theme of Vygotsky's sociocultural theory of mind was the claim that mental functioning is mediated by tools and signs. "The sign acts as an instrument of psychological activity in a manner analogous to the role of tool in labor" [13, p. 52]. diSessa [1] calls this *Material Intelligence*, saying "we can instill some aspects of our thinking in stable, reproducible, manipulable, and transportable physical form" (p. 6). Work looking at the relationship between signs (or more broadly representations) and cognition has delineated the particularities of how representations are bound up with knowledge, learning, tasks and uses [1, 11, 12, 14-16]. Similar work focusing on the design of programming languages has shown how various features of the representation, be they visual [17, 18], semantic [19], or syntactic [20], all influence the ease of use of the resulting language.

In their work on the development of mathematical meaning in computational settings, Noss & Hoyles [21] developed the theoretical construct of *webbing* to capture the nature of the learning process in rich computational settings. Webbing describes a "structure that learners can draw upon *and reconstruct* for support – in ways that they choose as appropriate for their struggle to construct meaning" [21, p. 108]. The construct is intended to capture the full, interconnected set of resources available to the learner as they progress through their meaning-making endeavor and respects the fact that each learner is unique and will leverage different features of the language in different ways. Webbing was proposed as a way to describe how understanding emerges that is consistent with the situated nature of the learning task and acknowledges the central role of the tools used in the process. This construct is particularly valuable when analyzing the role of block-based programming languages in introductory learning environments as it provides a way to makes sense of the full set of features of the language (semantics of keywords, visual display, syntax constraints, etc.) and identify the differing roles they play during the learning process and across learners [22]. Likewise, it does not demand that each component of an environment be considered in isolation, a challenge often encountered when trying to study block-based programming environments [23]. Bringing this analytic lens to the study of block-based programming environments reveals that the language primitives and their presentation play a variety of roles in helping novices achieve their goals.

In this work, we bring a representation-as-mediational-means lens to block-based programming languages. As such, the unit of analysis for this work is not the individual blocks, nor the full library of blocks provided by a block-based environment, but instead, the unit of analysis is the block-based environment in conjunction with the user interacting with it. This is consistent with the theoretical construct of webbing and recognizes the central role of the learner in the learning experience. Thus, the framework and the examples provided, treat user and tool as co-constituents in the ongoing learning process. This lens brings specific features of the language (sematic and syntactic) into focus alongside the environment in which it is situated (programming activity and interface) and the unique experiences and prior knowledge of the learner.

Block-based programming environments leverage a programming-primitive-as-puzzle-piece metaphor that provides visual cues to the user about how and where commands can be used as their means of constraining program composition. Programming in these environments takes the form of dragging blocks into a composition area and snapping them together to form scripts. If two blocks cannot be joined to form a valid syntactic statement, the environment prevents them from snapping together, thus helping to alleviate difficulties with syntax while retaining the practice of assembling programs instruction-by-instruction. Block-based languages, unlike more conventional text-based languages make the atomic unit of composition a node in the abstract syntax tree of the program, as opposed to a smaller element (i.e. a character) or a larger element (like a fully formed functional unit). In making the abstract syntax tree node the constructible unit, the building block of the representation shifts, giving the user a different set of objects-to-think-with [4], and thus providing a different set of supports and enabling different types of uses relative to text-based alternatives. Understanding and giving structure to the new roles and affordances of the block-based modality is the central objective of this paper.

In formulating our framework for categorizing the ways that novices use block-based languages, we looked to the literature and found two distinct dimensions along which mediational roles differ that could lead to a productive classification that fit our emerging findings. Kaput [24], in his work on the roles of symbols in mathematics, identifies two complementary uses for the material form of mathematical expressions: "the support of internal cognitive processing and communication between persons" (p. 160). We categorize this difference as internal (cognitive) vs. external (communicative); these categories provide the first dimension of our framework. The second dimension along which programming representations can differ comes from the computer science education literature, where a distinction is made between the act of generating a program and that of comprehending one [25]. This difference in purpose (generative vs. interpretive) forms the second dimension of our framework, producing a 2x2 matrix (Table 1).

Table 1. The 2x2 matrix situating the four roles Block-based programming language primitives play in supporting novices.

	Generative	**Interpretive**
External (Communicative)	Means for expression	Record of previously expressed intentions
Internal (Cognitive)	Source of Ideas	Resource used in meaning-making

The four quadrants of this framework delineate the four roles we identify in our analysis. The External-Generative role is the one most closely aligned with the conventional view of the

purpose of programming languages: that of an expressive medium with which to encode ideas in a computationally executable form. In this role, the user conceives of a general idea or specific intention, and then uses the programming language to mediate the expression of that idea into a form the computer can carry out. The second identified use of the block-based representation is serving in an External-Interpretive role. In this capacity, the modality acts as an external record that preserves previous intentions, serving as the *memory* in the distributed cognitive system of the programming environment [26]. Unlike the first role, which defines the human-to-computer interaction, this role captures asynchronous human-to-human communication in the form of one user reading the instructions previous written by others. A third role that language primitives can play is acting as a source of ideas for constructions, which defines the Internal-Generative quadrant of our classification. In this role, the representational system is not mediating the expression of an idea, but instead, the language itself acts as a resource the user can leverage to form new ideas. Block-based languages are particularly well suited for this role given the way they are presented, as will be shown later in the paper. The final role of this orienting framework is Internal-Interpretive, which manifests itself as novices using the language as a cognitive resource to make sense of observed behaviors. In this role, the author uses the programming commands as a mechanism to help decipher and interpret observed behaviors of the program, serving as objects-to-think-with [4] in facilitating the meaning-making process.

While we see these four roles as distinct, in practice, they are often used in conjunction or quick succession as part of a single effort. We see this ontology as productive in that each dimension suggests a pattern of use for novices and provides a lens for studying the ways the representational system is being appropriated by the learner. Further, the application of this framework can be used to inform the evaluation and design of programming languages. This framework is not meant to be definitive, but instead is one possible way to categorize novice interactions with programming environments.

Finally, the framework was derived with block-based programming environments in mind, but may provide insights beyond block-based contexts. This aspect of the framework will be revisited at the conclusion of the paper.

3. Methods

To develop and validate this framework, we conducted a study asking programming novices to play RoboBuilder [27], a constructionist, program-to-play game [28] in which writing programs is the main mechanism of gameplay (Fig. 1). The central challenge of RoboBuilder is to design and implement strategies to make an on-screen robot defeat a series of progressively more challenging opponents. A player's on-screen robot takes the form of a small tank, which competes in one-on-one battles against opponent robots equipped with the same set of capabilities. Unlike a conventional video game where players control their avatars in real time, in RoboBuilder, players must program their robots before the battle begins. To facilitate this interaction, RoboBuilder has two distinct components: a graphical programming environment where players define their robots' strategy, and an animated battleground where their robots compete (Fig. 1). To implement their strategy, players use a domain specific, block-based programming language. The language includes movement blocks (ex: `forward, turn gun right, fire`) to control the robot's motion, event blocks (ex: `When I See a Robot, When I Get Hit`) to control when instructions will execute, and control blocks (ex: `Repeat, If/Then`) that can be used to introduce logic into the robot's strategy. RoboBuilder uses an event-based programming model where in-game events are linked to the language's event blocks, so that when a certain action occurs (like the robot hitting the wall), flow of control of the program is passed to the associated event (`When I hit a wall`).

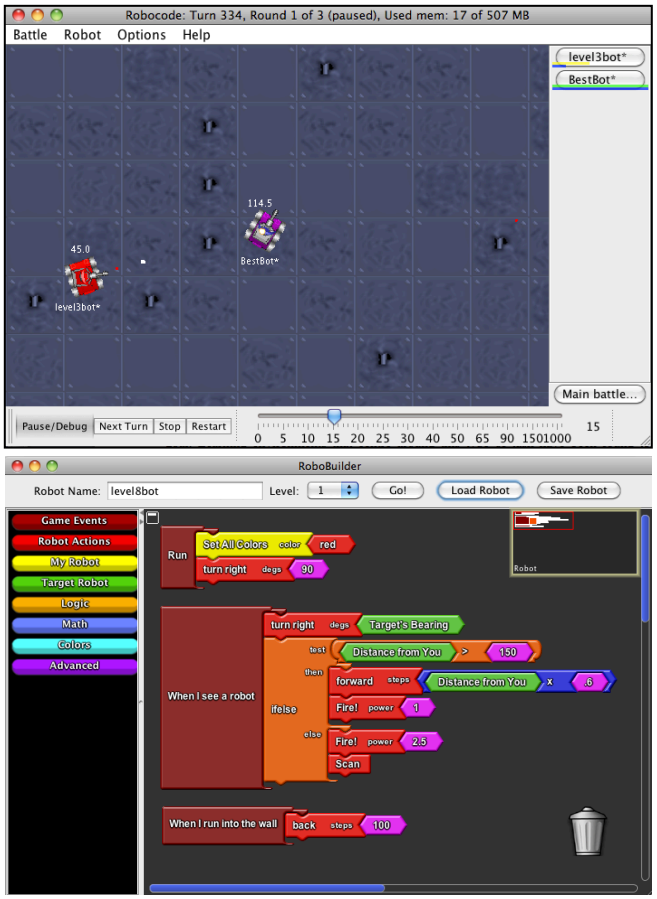


Figure 1. RoboBuilder's two screens. The battle screen (top) where players watch their robots compete and the construction space (bottom) where players implement their robot strategies.

The data presented in this paper are from 16 RoboBuilder play sessions conducted with programming novices ranging from middle school to graduate school. The university-aged participants were students at a Midwestern American university. Two of the younger participants were recruited through university connections, while the remainder of the participants were recruited through a community center in a Midwestern city that serves a predominantly African-

American, low SES population. Each participant played RoboBuilder for at least 40 minutes, resulting in a total of roughly 18 hours of interview and gameplay footage and over 200 robot strategies created.

The data were collected through one-on-one interviews in which a researcher sat alongside the participant as he or she played the game. At the outset of a session, the interviewer introduced the participant to RoboBuilder, explaining the game objective and the components of the game environment. The participant was then given a chance to ask questions before the actual game play procedure began. The gameplay portion of the session proceeded in an iterative, three-phase protocol. First, players are asked to verbally explain their intended strategies to the interviewer in conversation. Next, they are given the chance to implement their proposed strategy using the block-based language. Finally, participants click the 'Go' button, and then watch their robot compete, with the interviewer asking them to describe what they observe and whether or not it matches their expectations. At the conclusion of the battle, the next iteration of the protocol would begin with the interviewer asking participants what alterations they plan on making to their strategy to progress in the game. This three-phase cycle was repeated for the duration of the session. Throughout the session, the researcher's role was mainly to move the iterations forward by using various prompts to get participants to verbalize their thought process. The researcher also answered clarifying and technical questions when they arose. Each RoboBuilder session was recorded using both screen-capture and video-capture software.

4. Four Roles of Block-based Primitives

In this section, we provide vignettes and a discussion for each of the four roles of the framework. These vignettes are intended to demonstrate interactions for each quadrant of the framework and act as illuminating examples that can be drawn on to inform our thinking about how block-based languages support novices.

4.1. External-Generative: Primitives as an Means for Expression

In RoboBuilder, language primitives serving as a means for expression can be seen when a participant uses the language to implement an idea that he or she has conceived of, but not yet expressed in code. In other words, they are using the language to encode their intention so that the computer can execute them. An example of the block-based programming language playing this role involves Morris[1], a university student with no prior programming experience. At the outset of his interview, when asked what his strategy would be, Morris responded:

> *So, my master plan is to, like, be continuously moving, so it's harder to hit. If I get hit, kind of change the path so it's different than what you might be expecting however the sequence is running, and then, during that path, adjust to what the opponent is doing to hit them.*

He then proceeded with the construction of his robot strategy. After six minutes of working, he had produced his first program; the first three events of which are displayed in Fig. 2. Comparing the strategies Morris articulated in his initial remarks to the program he constructed, we can see the blocks taking on an expressive role, mediating and enabling the computational implementation of his ideas. His "master plan" included three distinct ideas, each of which can be seen in his resulting program. His first strategy: *"be continuously moving, so it's harder to hit"* is achieved with the `Run` method of his program (left side of Fig. 2). This series of instructions will result in his robot remaining in constant motion. Morris' second verbalized tactic: *"if I get hit, kind of change the path so it's different"*, can be found encoded in his `When I get Hit` event block (top right of Fig. 2). These two instructions will execute when his robot gets hit and will cause it to change its heading and move forward out of the current line of fire. His final idea: *"adjust to what the opponent is doing to hit them"* is captured by his `When I See a Robot` command (bottom right of Fig. 2), which makes his robot adjust its gun towards the location of his enemy and fire at it.

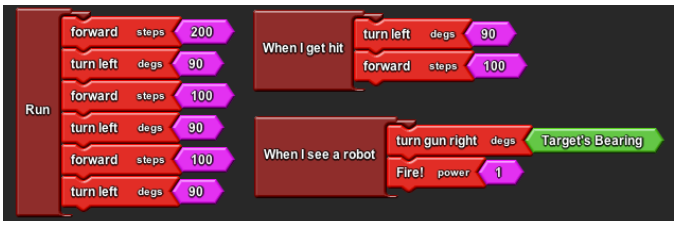


Figure 2. The first three events of Morris' initial RoboBuilder program.

From the first five minutes of Morris' RoboBuilder session we can see how the language primitives can serve as a means for expression. A second demonstration of the language serving in this capacity occurs roughly twenty minutes into Daniel's RoboBuilder session. Daniel is a tenth-grade student with no prior programming experience. After seeing his first two robot strategies struggle against the level-one opponent, Daniel decided he needed a new approach. He realized he was having difficulty locating his opponent; this prompted him to propose the following strategy: *"since* they *change the position of the robot every time, I won't know where it's at. So, I just want to make* [my robot]*, like, spin in a circle and shoot."* Having verbalized this new idea, Daniel proceeded to construct the strategy shown in Fig. 3.

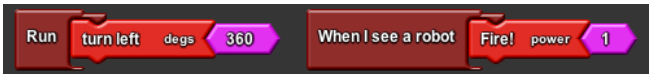


Figure 3. Daniel's implementation of his "*spin in a circle and shoot"* strategy.

The result of these commands is that his robot continuously rotates in a circle, shooting whenever the opponent comes into view. After trying out his new strategy, the interviewer asked

[1] All names are pseudonyms.

Daniel to describe what his robot was doing, Daniel responded: *"it's spinning in a full circle, and when he sees the robot he's shooting."* In other words, the robot is carrying out the strategy that Daniel had just vocalized. Here again we see the language primitives serving as a means of expression enabling the computer to carry out the intentions of the user.

These two vignettes were chosen because they provide clear demonstrations of the language primitives being used in the expressive capacity and serve as examples of the first identified role that language primitives can play in a programming activity: that of a mediating role between an idea generated by a user and a computationally executable reification of that same idea. This is a demonstration of language primitives being used in an External-Generative role, where the end result is a computationally executable form of the idea. It is important to mention that this idea-to-implementation process was not always so direct or easy. Often, over the course of our interviews, players either struggled to encode their stated intentions, or composed strategies that did not match their expressed intentions, at times relying on other features of block-based programming languages that will be discussed later in this analysis.

External-Generative: Discussion

The ability for a programming language to enable users to express ideas in such a way as to be executable by a computer is an essential feature of the representation, as, by definition, if it is not possible to write a program using the representational system, it can hardly be considered a programming language. That being said, it is certainly not the sole feature, and, arguably, not even the most important, as [29] famously says, "programs must be written for people to read and only incidentally for machines to execute". Programs, and programming languages, serving as a means of expression has long been argued as a pedagogical strength of the form [29]. This role is akin to the ability for the alphabet to be used to express ideas in the written form, the difference being in the case of programming languages, the audience is not solely another human, but also a computer.

In this way, programming languages serve as a bridge across what Hutchins et al. [30] call the gulf of execution, which describes the distance between a user's goals and the expression of those goals using the representations understood (and often defined by) the system. The design of the representational system can facilitate this bridging role "by making the commands and mechanisms of the system match the thoughts and goals of the user" (p. 318). In the case of RoboBuilder, to support programming novices in expressing their ideas with the provided representational system, the language primitives were designed to carry semantic meaning within the context of the game in such a way as to enable players to understand how they could be used. This can be seen in the close mapping between the verbal language of the player and the labels on the blocks, for example, Morris said: "*If I get hit"* and then used the `When I get hit` block.

In the first example, Morris relied on the natural language label on each block to select appropriate commands, the closeness of mapping to his intentions, and the shape of the blocks to facilitate his assembling them into a script. Daniel, along with these features, also used feedback from the environment in the form of seeing his opponent reposition itself, to inform the strategy he devised. All of these aspects have been identified as useful features of the block-based modality for learners [31]. These different supports designed into the language and environment contribute to the webbing upon which learners draw in order to support this first use of block-based languages. The two examples shown above highlight how not all users draw upon the resources available in a learning environment in the same way. In this way, block-based tools and their suite of scaffolds support an epistemological pluralism [32].

4.2. External-Interpretive: Primitives as a Record of Previously Expressed Intentions

The second role block-based languages can play is that of a record of previously expressed intentions, serving in an External-Interpretive capacity. After a user writes a program (i.e. uses the language in the previously discussed External-Expressive capacity), the language remains a visible, legible artifact that can later be referred back to and read either by the original author or other interested parties. Used in this capacity, the language serves as a record of previously expressed instructions, or as a resource to refer to for mapping outcomes onto expressed instructions. An example of this usage can be seen toward the end of Anne's RoboBuilder interview. Anne, a third-year undergraduate student, had just finished implementing the seventh iteration of her robot, during which she introduced the `When I get Hit` event to her strategy in hopes of addressing a weakness she had identified: if her robot got hit, it did not move; instead it stayed in place, making it easy for her opponent to hit her again. To address this issue, Anne decided to have her robot move to a new location if it got hit. Fig. 4 shows the two events from Anne's program that are relevant for this episode.

Figure 4. The two events of interest from Anne's robot strategy.

After starting a battle with this new behavior in place, her robot was behaving as expected until it was hit a few times in succession and backed into a wall. Her robot then remained pinned to the wall, motionless, getting hit until the match ended. Upon seeing this, Anne got a confused look on her face and said aloud: "*Wait, what happened?*" Not being able to make sense of what she was seeing based on what she remembered programming, Anne, speaking to herself, asked: "*Wait, but when I run into a wall, what'd I put?"* She then brought the programming window to the forefront and read through her instructions, quickly realizing the bug she had introduced. When her robot backed into a wall, her `When I Run into the Wall` logic would instruct her robot to back up an additional 300 steps; in doing so it hit the wall again,

thus producing an endless loop. To debug her strategy, Anne used the programming language in an External-Interpretive capacity; she read through the instructions using them as a record of her previously articulated strategy to identify the bug in her program.

External-Interpretive: Discussion

This vignette provides an example of the second role that programming language primitives can play during a programming task — that of a preserved record of the instructions followed by the computer that can later be referred to and analyzed. This use falls in the external dimension of our ontology as it relies on the communicative aspect of the blocks, but unlike the previous vignette, where the language was used in a generative capacity, here, Anne used the language to accomplish an interpretive goal. With computational representational systems, the primary audience for a constructed artifact is usually the computer on which it is going to be run, but there is also a secondary audience: any human tasked with interpreting, modifying, or extended the program. Because programs exist as sets of instructions that produce dynamic outcomes, it is essential for the language to support being read at a later time, either by the initial author or by others. Here again it is appropriate to cite [29] and their claim that "programs must be written for people to read and only incidentally for machines to execute". While it is being run, the written program serves as a blueprint, containing an explanation for the resulting behavior.

In this vignette, without referring back to her program, Anne was unable to make sense of what her robot was doing. To help her interpret its behavior, she re-read the program she had authored; using the language in a mediating role to provide guidance on what was happening. In this case, it was the original author who was reading her own code, but it is very common for programs written by one person to be read by others so they can understand, and ultimately use, or extend the program. In this way, programming languages serve as a means to mediate the expression of ideas as well as serve as a record of the ideas already expressed. Through the lens of webbing, the permanence of the constructed artifact, the previously mentioned closeness-of-mapping of the commands, and the visual execution of the program were all designed aspects of the environment that helped Anne debug her program. One goal for this framework is that it be useful for evaluating and improving programming environments.

In evaluating block-based programming's ability to be used in an External-Interpretive capacity, we see one potential direction for future improvement. Prior work has found that the block-based representation poorly supports longer programs [31]; as program length and complexity grow, the block-based modality can make the program more difficult to follow. In other words, block-based languages may struggle to support the External-Interpretative aspect of programming languages. In response to this drawback, new block-based tools are being designed to address this shortcoming by blending features of block-based and text-based modalities [33, 34] or by allowing users to move back-and-forth between modalities [35, 36].

4.3. Internal-Generative: Primitives as a Source of Ideas

When trying to develop an approach for accomplishing a desired computational goal, the language itself can be used as a resource. By internalizing the possibilities provided by the language, the author can use the language itself to bootstrap idea generation for potential solutions. This is one possible use of a programming language that falls in the Internal-Generative dimension of our framework. Block-based languages are especially well suited for this use as the visual arrangement and pre-defined categorization of the blocks make browsing and finding blocks easy. Our example of this usage comes from the start of the RoboBuilder interview conducted with Beth, an undergraduate student studying vocal performance. This was Beth's response to the initial question of how she was going to defeat her opponent:

> *Well, I...I don't know, it seems to make sense to have, to determine what would happen in every case, so I think I'll use these dark red buttons and try and figure out what I want to have happen.*

Beth then proceeded to go through each of the Game Events blocks (the "*dark red buttons*" she refers to in the quote), using them as a roadmap to develop her strategy. Fig. 5 shows Beth's first completed robot strategy alongside the Robot Events drawer that lists the available Game Event blocks.

Figure 5. On the left, is the Robot Events drawer; on the right is Beth's first implemented Robot.

What is especially interesting about Beth's first robot is that not only did she implement every event, but the order of the events in her program perfectly matches the presentation in the Robot Events drawer. The video from her interview shows Beth starting at the top of the events drawer and systematically working her way through the set of available blocks. This suggests that she did not have a clear, unified strategy when she began to program her robot. Instead, Beth built her program event-by-event, using the commands provided by the language to bootstrap the generation of a valid robot strategy. In this way, the language primitives supported Beth in conceptualizing possible actions that her robot could carry out.

Internal-Generative: Discussion

In this vignette, we see RoboBuilder's language primitives playing a distinctly different generative role than we saw in the vignettes in the External-Generative section. Whereas with Morris and Anne, the emphasis was on the language serving in an external and expressive capacity, with Beth, the primitives facilitate an internal, cognitive outcome; serving as a source of inspiration for generating ideas for her robot strategy. She even states her intention to use the language commands in this capacity, saying: "*I think I'll use these dark red buttons…and try and figure out what I want to have happen.*" Consistent with diSessa's [1] idea of "materially-mediated-thinking", in this episode we see Beth having ideas *with* the medium, as opposed encoding her preconceived ideas into the language. The language primitives are mediating her thinking about the challenge, seeding the ideation process for how to accomplish the in-game programming challenge. This use is further facilitated by the ease of testing and visualizing the behaviors of the blocks. The use of the language in this capacity also relates to Wilensky and Papert's [11] structuration theory linking representation and cognition, as the representation itself is making certain ideas more accessible. You can imagine that if instead of the descriptive blocks the game provides, the language was an abstract set of operations with labels like `operation1` and `state2`, then Beth would not have been able to use it in the way shown above, even if the language had the same computational capabilities. Here, the language serves in an Internal-Generative role, facilitating the generation of a new idea. When designing programming languages for novices, recognizing that primitives serve this role is important, as this use can help a novice achieve early programming successes. To the growing list of features that block-based languages include that support learners, we now add the organization and visual arrangement of the full set of blocks as another element of the webbing learners can draw on.

4.4. Internal-Interpretive: Primitives as a Resource Used in Meaning Making

The final quadrant of the framework describes programming languages serving in Internal-Interpretive roles. In this capacity, the language is used as a cognitive tool with which to interpret and make sense of the computational task at hand. Used in this way, the language need not be visible or even present, but instead is employed as a cognitive resource through which observed behavior can be understood. This vignette, also taken from Beth's RoboBuilder session, occurred during her second battle against the level-one opponent. The level-one robot's strategy is to remain motionless until its energy drops below 50, at which point it begins to move. At the start of the second battle, as Beth was watching the battle, she asked the interviewer when the opponent was going to start moving. The interviewer responded "*It happens at 50*", which prompted Beth to say:

> *It happens when it reaches 50? OK, so that robot must have something built into it when it reaches 50. OH! There we go, so that's what the, that's what the other boxes are for, so like if you reach a certain health level you can change the actions, oh, ok.*

This brief excerpt shows Beth using the language as a tool to mediate her understanding of the opponent's behavior without ever seeing the instructions externally represented. Her exclamation "*OH! There we go,*" suggests a moment of revelation, when some piece of the puzzle of how her opponent was behaving fell into place. She then explains that the "*other boxes*" (referring specifically to the conditional and robot state blocks, a fact that became clear later in the interview) can be used to create the behavior her opponent is carrying out. The key piece of this excerpt is her stating: "*if you reach a certain health level you can change the actions.*" This description maps perfectly onto the program that is controlling her opponent (shown in Fig. 6), but, importantly, these blocks are not visible to Beth, so she was unable to read the instructions, like we saw Anne do in the External-Interpretive vignette. Instead, she used the blocks as cognitive tools with which to interpret the opponent's behavior and devise a possible explanation for how its stationary-then-active strategy was achieved.

Figure 6. The hidden conditional logic inside the level-one opponent.

Internal-Interpretive: Discussion

In this fourth role, we see again how the language primitives can be used as objects-to-think-with [4] to support the meaning making process. This use fits with the diSessa's [1] Material Intelligence, where symbols serve as cognitive tools with which to make sense of the world. Likewise, it matches Kaput's [24] discussion of mathematical symbols and their role in delineating and providing structure for the mathematical activity at hand. What makes computational representational systems, and in particular block-based languages, especially capable for being used in the Internal-Interpretive capacity is their ability to offer a suite of resources, i.e. the webbing of the environment, to facilitate meaning making. This includes the ability to incorporate visual cues like color and shape that can make it easier to categorize how specific primitives can be used, and the embedding of existing, familiar symbol systems and representational conventions into the language's design, including natural language labels and mathematical symbols. This enables the set of primitives to include semantic hints in the form of meaning-carrying labels (such as `move forward` and `When I hit a wall`) that can bootstrap the cognitive process of interpreting observed behavior through the language itself.

5. The Challenge of Designing for All Four Roles

Recognizing the various roles programming primitives play has implications for designers of novice programming

environments and introductory programming languages. Attempting to design for all four quadrants of the Internal/External, Generative/Interpretive framework presents a challenge to the designer, as some design decisions made to support one usage may be at the expense of another. Each role suggests a different set of priorities and considerations for how the language should be designed and presented. An example from RoboBuilder's language makes this tension more concrete. The set of game events provided in RoboBuilder (`When I See a Robot, When I get hit,` etc.) were designed to provide conceptual hooks for players to introduce behavioral logic and enable them to use the blocks to guide the creation of strategies, as we saw in Beth's first vignette. However, by providing a fixed set of events, the language constrains how and when logic can be introduced in the game, limiting its expressive capabilities in the External-Generative capacity. This type of design decision comes down to a question of finding the right grain size for the language primitives. This challenge was encountered in the design of low-threshold computational modeling tools: "It is critical to design primitives not so large-scale and inflexible that they can only be put together in a few possible ways…On the other hand, we must design our primitives so that they are not so 'small' that they are perceived by learners as far removed from the objects they want to model" [37, p. 168]. Finding the right size primitives is one of the central challenges for designers when creating languages for novice programmers. Our decision to provide a standard set of events, as opposed to a customizable set, is an example of the design trade-offs one encounters when designing a representational system that can support all of the roles specified by this framework.

While the analytic framework we put forth in this paper was introduced and discussed as a means of understanding block-based languages, it need not be tied to that modality, as text-based or other graphical representations share these four distinct uses. While we expect the manifestations of the four quadrants would differ with other representational systems, we expect the framework would still be illuminating and fruitful.

6. Conclusion

When creating a new computational language for novices, a diverse set of uses should be considered. By providing a classification system for the roles block-based programming languages take in for novices, and providing examples of each, we seek to provide a set of aspects designers should consider when creating new computational tools. We also see this framework as a useful lens with which to analyze existing computational representational systems. Understanding how they are used is an important first step in refining existing and designing new tools.

In our use of webbing as a theoretical construct to ground the analysis, the findings were necessarily coupled with the block-based language under investigation, but it is easy to draw connections from this work to conventional text-based languages. Text-based programming languages provide the same fundamental capabilities as block-based tools, although at times the specifics may differ. As such, we believe this framework can be useful when applied to conventional text-based programming languages, but for now, this remains future work.

The creation of accessible, yet powerful, languages is a critical challenge we face in laying the infrastructure for the computationally literate society championed at the outset of this paper. By recognizing the various roles primitives can play in supporting novices in computationally expressing ideas, we as designers and educators can begin to develop new languages and environments that support these different usages to scaffold learners. In doing so, we can make progress toward this vision of a computationally literate 21st century.

References

[1] A. A. diSessa, *Changing Minds: Computers, Learning, and Literacy*. Cambridge, MA: MIT Press, 2000.

[2] M. Guzdial and E. Soloway, "Computer science is more important than calculus: The challenge of living up to our potential," *SIGCSE Bulletin*, vol. 35, no. 2, pp. 5–8, 2003.

[3] National Research Council, *Report of a Workshop on The Scope and Nature of Computational Thinking*. Washington, D.C.: The National Academies Press, 2010.

[4] S. Papert, *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic books, 1980.

[5] D. Weintrop, E. Beheshti, M. Horn, K. Orton, K. Jona, L. Trouille, and U. Wilensky, "Defining Computational Thinking for Mathematics and Science Classrooms," *Journal of Science Education and Technology*, vol. 25, no. 1, pp. 127–147, 2016.

[6] U. Wilensky, "Modeling nature's emergent patterns with multi-agent languages," in *Proceedings of EuroLogo*, Linz, Austria, 2001, pp. 1–6.

[7] J. M. Wing, "Computational thinking," *Communications of the ACM*, vol. 49, no. 3, pp. 33–35, 2006.

[8] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.

[9] K. D. Forbus, R. W. Ferguson, and J. M. Usher, "Towards a computational model of sketching," in *Proceedings of the 6th International Conference on Intelligent User Interfaces*, 2001, pp. 77–83.

[10] C. Duncan, T. Bell, and S. Tanimoto, "Should Your 8-year-old Learn Coding?," in *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*, New York, NY, USA, 2014, pp. 60–69.

[11] U. Wilensky and S. Papert, "Restructurations: Reformulating knowledge disciplines through new representational forms," in *Proceedings of the Constructionism 2010 conference*, Paris, France, 2010.

[12] J. Kaput, R. Noss, and C. Hoyles, "Developing new notations for a learnable mathematics in the computational era," in *Handbook of International Research in Mathematics Education*, 2002, pp. 51–75.

[13] L. Vygotsky, *Mind in Society: The Development of Higher Psychological Processes*. Cambridge, MA: Harvard University Press, 1978.

[14] B. L. Sherin, "A comparison of programming languages and algebraic notation as expressive languages for physics," *International Journal of Computers for Mathematical Learning*, vol. 6, no. 1, pp. 1–61, 2001.

[15] U. Wilensky and S. Papert, "Restructurations: Reformulations of knowledge disciplines through new representational forms," Manuscript in Preparation.

[16] J. Zhang and D. A. Norman, "Representations in distributed cognitive tasks," *Cognitive Science*, vol. 18, no. 1, pp. 87–122, 1994.

[17] D. Weintrop and U. Wilensky, "Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research (ICER '15)*, New York, NY, USA, 2015, pp. 101–110.

[18] C. D. Hundhausen, S. F. Farley, and J. L. Brown, "Can direct manipulation lower the barriers to computer programming and promote

transfer of training?," *ACM Transactions on Computer-Human Interaction*, vol. 16, no. 3, pp. 1–40, Sep. 2009.

[19] J. F. Pane, B. A. Myers, and L. B. Miller, "Using HCI techniques to design a more usable programming system," in *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, Los Alamitos, 2002, pp. 198–206.

[20] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," *ACM Transactions on Computing Education*, vol. 13, no. 4, pp. 1–40, Nov. 2013.

[21] R. Noss and C. Hoyles, *Windows on Mathematical Meanings: Learning Cultures and Computers*. Dordrecht: Kluwer, 1996.

[22] D. Weintrop and U. Wilensky, "Situating programming abstractions in a constructionist video game," *Informatics in Education*, vol. 13, no. 2, pp. 307–321, 2014.

[23] D. Weintrop and U. Wilensky, "The challenges of studying blocks-based programming environments," in *IEEE Blocks and Beyond Workshop*, 2015, pp. 5–7.

[24] J. J. Kaput, "Towards a theory of symbol," in *Problems of Representation in the Teaching and Learning of Mathematics*, C. Janvier, Ed. Hillsdale, NJ: Lawrence Erlbaum Associates, 1987, p. 159.

[25] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion," *Computer Science Education*, vol. 13, no. 2, pp. 137–172, 2003.

[26] J. Hollan, E. Hutchins, and D. Kirsh, "Distributed cognition: toward a new foundation for human-computer interaction research," *ACM Transactions on Computer-Human Interaction*, vol. 7, no. 2, pp. 174–196, 2000.

[27] D. Weintrop and U. Wilensky, "RoboBuilder: A program-to-play constructionist video game," in *Proceedings of the Constructionism 2012 Conference*, Athens, Greece, 2012.

[28] D. Weintrop and U. Wilensky, "Program-to-play videogames: Developing computational literacy through gameplay," in *Proceedings of the 10th Games, Learning, & Society Conference*, Madison, WI, 2014, pp. 264–271.

[29] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*. MIT Press 2nd ed, 1996.

[30] E. L. Hutchins, J. D. Hollan, and D. A. Norman, "Direct manipulation interfaces," *Human-Computer Interaction*, vol. 1, no. 4, pp. 311–338, Dec. 1985.

[31] D. Weintrop and U. Wilensky, "To block or not to block, that is the question: Students' perceptions of blocks-based programming," in *Proceedings of the 14th International Conference on Interaction Design and Children (IDC '15)*, New York, NY, USA, 2015, pp. 199–208.

[32] S. Turkle and S. Papert, "Epistemological pluralism: Styles and voices within the computer culture," *SIGNS: Journal of Women in Culture and Society*, vol. 16, no. 1, pp. 128–157, 1990.

[33] M. Kölling, N. C. C. Brown, and A. Altadmri, "Frame-based editing," *Journal of Visual Languages and Sentient Systems*, vol. 3, no. 1, Jul. 2017.

[34] J. Mönig, Y. Ohshima, and J. Maloney, "Blocks at your fingertips: Blurring the line between blocks and text in GP," in *IEEE Blocks and Beyond Workshop*, 2015, pp. 51–53.

[35] D. Bau, D. A. Bau, M. Dawson, & C. S. Pickens, "Pencil Code: Block Code for a Text World," in *Proceedings of the 14th International Conference on Interaction Design and Children (IDC '15)*, New York, NY, USA, 2015, pp. 445–448.

[36] M. Homer and J. Noble, "Lessons in combining block-based and textual Programming", *Journal of Visual Languages and Sentient Systems*, vol. 3, no. 1, Jul. 2017.

[37] U. Wilensky, "GasLab: An extensible modeling toolkit for connecting micro-and macro-properties of gases," in *Modeling and Simulation in Science and Mathematics Education*, N. Roberts, W. Feurzeig, and B. Hunter, Eds. Berlin: Springer-Verlag, 1999, pp. 151–178.

Towards Understanding Successful Novice Example Use in Blocks-Based Programming

Michelle Ichinco, Kyle J. Harms, Caitlin Kelleher
Dept. of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO, USA
{michelle.ichinco, harmsk, ckelleher}@wustl.edu

Abstract *Blocks-based environments are frequently used in settings where there is little or no access to teachers. Effective support for examples in blocks-based environments may help novices learn in the absence of human experts. However, existing research suggests that novices can struggle to use examples effectively. We conducted a study exploring the impacts of example-task similarity and annotation style on children's abilities to use examples in a blocks-based environment. To gain understanding of where and why children struggle, we examined the degree to which (1) children were able to map a task to its corresponding example, and (2) their programming behavior predicted task success. The results suggest that annotations improve task performance to an extent and that mappings and programming behavior can begin to explain the remaining problems novices have using examples.*

1. Introduction

Blocks-based programming environments, such as Scratch [1], App Inventor [2], and Looking Glass [3] have been growing in popularity. In this paper, we start to explore how novices use examples in a blocks-based programming environment. Understanding example use in blocks-based environments is important because effective example use can: (1) help to provide learning support for novices who lack access to formal classes and qualified teachers, and (2) be a useful skill as programmers transition to more independent projects.

Blocks-based environments are often used by children with very little programming experience. Due to a lack of qualified computer science teachers [4], children commonly use blocks-based environments for short classroom projects, extracurricular activities, or on their own. As a result, many blocks-based programmers rely on the programming environments and related online materials to provide learning opportunities. Current systems support learning through tutorials [5], games [6, 7], and intelligent tutoring [8]. However, in order to work toward a programming goal, novices likely need more context-specific information like they might find in example code that is similar to their goal.

Studies have found that programmers of all experience levels often use example code found online to accomplish particular tasks [9–13]. The ability to use examples effectively is an important skill for programmers to have, as it enables the programmers to continue to gain new skills as technologies change. Unfortunately, inexperienced programmers often struggle when attempting to reuse others' code [14]. Due to their difficulties understanding example code behavior, some novices report referencing example code as a model rather than adapting it and integrating it into their programs [12, 13]. Yet, research on programming and examples has primarily focused on selecting examples [15] or adapting them [16], rather than the process of example use.

In this work, we ran a study exploring how novice programmers use examples. Our first goal was to compare annotation styles and example-task similarities to understand how these commonly used styles of examples affect performance. We measured performance through task success and analogical mappings. The idea of analogical mappings comes from problem solving, where an analogical mapping refers to a relationship between corresponding parts of two problems. Often, that relationship can help a learner to solve a problem by figuring out how the solution of one problem relates to another problem they are trying to solve [17, 18]. Likely, using an example to solve a programming problem requires a mapping between the two snippets of code. If analogical mappings between example code and tasks indicate a likelihood to succeed, systems could use that information to determine when and how to provide support. In order to not affect the problem solving process, we collected analogical mappings after users completed tasks, which may not be able to indicate whether analogical mappings can predict success during a task. Thus, we hypothesized that novices' mappings between tasks and examples would correlate with task performance, which can begin to indicate how analogical mappings in programming relate to task success.

The example styles only had a small impact on both task performance and analogical mappings, so we performed a post-hoc analysis that looked at performance in terms of four "stages" of programming task completion. In blocks-based programming, programmers often need to (1) manipulate the GUI, and then (2) locate, (3) insert, and (4) correctly apply blocks in order to solve programming tasks. We used these

DOI reference number: 10.18293/VLSS2017-009

four stages to investigate two ways of understanding when novices are having trouble using examples: correctness of analogical mappings and programming behaviors. We hypothesized that participants' programming behaviors (i.e. manipulating the interface, editing the code, or executing the program) will have some predictive power, which we explore using decision trees.

To investigate these hypotheses, we ran and analyzed a study looking at novice programmers using examples in a blocks-based programming environment. Researchers can use the results of this study to inform the design of ways to help novice programmers use examples in blocks-based programming environments and educational systems for computer science using examples. The goal of this study is to better understand novices' example use in programming by answering 3 questions:

1. How does the interaction of annotations and example similarity affect novice programmers' performance on tasks using examples?
2. To what degree does the ability to map an example and target problem correlate with task success?
3. To what degree do programming behaviors predict task success?

2. Related Work

We first discuss related work surrounding examples in programming and in education. We then discuss how this work relates to other ways for novice programmers to learn independently in blocks-based programming environments.

2.1. Support for Programming with Examples

There is a broad range of work on programming with examples. However, most of it focuses on support for selecting appropriate examples or adapting an example for a new context, but does not look at the problems novices have in trying to use examples.

Systems focused on assisting in example selection either help programmers to find better examples or provide them directly. Specialized search tools for programming enable users to quickly perform keyword searches over API documentation [19], open-source projects [20–22], and code snippets presented on web pages [15, 21, 23]. Since keyword searches can return a large number of irrelevant results, some systems support specifying more constrained searches [24–28] or provide suggestions based on the programmer's current context [29]. Some programming environments provide access to examples by including a small set of pre-created examples [30–37], integrating example search directly into the environment [19, 20], enabling access to programs created by other users [1, 38, 39], or using direct manipulation to generate example code [29, 30, 33]. All of these systems aim to either make it easier to select an analogous example for a certain problem or attempt to suggest a useful example. Only one system that we know of, the Idea Garden, actually frames example use as analogical reasoning [40]. In the Idea Garden, analogy is introduced as a strategy for overcoming barriers, but the analogy was used mainly for selecting an example to use.

Programming systems currently support using examples, as well as integrating example code. Systems have added annotations as a way of supporting example use. Generally, these annotations either provide general descriptions of the code as a whole [15, 40] or provide specific information about certain parts of the code [16, 30, 41]. Other systems support programmers in integrating example code into programs, essentially removing the need for programmers to create mappings in order to use an example to solve their problem. For instance, Codelets provides a widget to allow easy modification of example code [42], while WebCrystal allows users to select which combination of features to integrate from an example [16].

Work on programming with examples is mainly in textual programming languages and focuses on selecting an example or supporting example integration, rather than understanding the issues novice programmers in blocks-based environments have while using examples.

2.2. Learning From Examples

The idea of supporting example use in order to improve independent learning opportunities is supported by two areas of research: (1) educational systems that introduce examples, and (2) theories of learning from examples. This study used examples more like programmers would find on the web because we wanted to simulate the experience of a novice programmer learning programming outside of a classroom, but the research on examples in education has inspired design choices in this study.

A number of educational systems provide support for example use [30, 43–45]. There are also systems [46, 47] that integrate examples into tutorial systems for programmers based on Caroll's theory on minimalist documentation [48, 49]. Similarly, other systems also provide sets of annotated examples to support learners, called 'case libraries' [50]. Case libraries provide sets of examples that relate to a problem learners are trying to solve [51]. This idea is supported by case-based reasoning theory, which focuses on having students learn through experiences and reflection [52]. The work on case libraries for case-based reasoning is limited and does not address programming examples. Furthermore, all of the examples in these systems are designed to fit within educational systems, rather than considering how novices can use examples to learn independently. They also do not address different types of example styles and how they affect novice programmer use. One study [53] looked at novice programmer example use, but only for one type of annotation style and mainly focused on the participants' descriptions of their difficulties, rather than on data that could possibly predict success or failure. In the discussion, we explore the relationship between the results in the two studies.

Research on learning from examples, such as worked examples and cognitive load theory, are important to consider

in thinking about novice programmer example use. Worked examples are a popular and well studied learning method found to be more effective than problem solving in some cases [54]. Worked examples are grounded in cognitive load theory, which is the mental effort for a novice to learn something new. Cognitive load theory can be reduced by improving instructional material, such as worked examples, to focus a learner's attention on the steps needed to solve a problem [55]. Worked examples have been used to teach a variety of topics, including mathematics [54] as well as programming [44, 56]. One recent study uses programming worked examples to study the effect of labels on learning [57]. Worked example research has also investigated differing degrees of example similarity, finding that both similar and dissimilar examples can be useful for learning [58, 59]. The research on worked examples supports the idea that the use of examples in learning can be highly effective, but worked examples have been primarily studied in classroom contexts. More work needs to be done to determine how the ideas from cognitive load theory and worked examples can apply to the types of examples programmers would find on the web, when programmers are more focused on completing a task than working through educational material.

2.3. Independent Learning for Novice Programmers

There are a variety of blocks-based programming environments that are used outside of classrooms and often provide some support for learning without a structured class, such as games, tutorials, and reuse.

Many blocks-based environments provide tutorials on their websites, such as Scratch [60] and App Inventor [61], which both have web pages providing video tutorials to get users started. However, video tutorials can be hard to use because it is often difficult to step forward or backward [62]. Furthermore, one study found that tutorials were not as effective as puzzles for novices for learning [63]. Some blocks-based environments are games-based, such as Blockly Games [64] and Code.org [6]. Games have been shown to be effective learning mechanisms for novice programmers [65], but often do not allow users to create their own project. Furthermore, learning through games does not apply to learning more advanced programming, where example use is critical.

Many blocks-based environments now provide the ability to share and reuse others' code, similar to the way experienced programmers sometimes use example code. Scratch [60] and Looking Glass [3] provide explicit "remixing" features. App Inventor [2], Kodu [66] and Touch Develop [67] also allow users to use other programmers' projects. However, research has found that novice programmers often have trouble selecting the code they need, which likely makes adapting others' code difficult for novices [14]. Research has been able to cluster behaviorally similar visual code for Scratch [68], which could help novices to find appropriate code more easily in blocks-based environments, but it is still unclear how to help them use example code more effectively.

Overall, prior work on examples in programming has focused on more experienced programmers in text languages. The research on examples in education supports the idea that examples can be an effective way for novices to learn new concepts. Novice programming environments provide a variety of learning supports, but none addresses understanding novices' behavior when using examples as models for learning.

3. Study

The goal of this study was to better understand novices' example use by exploring (1) how example similarity and annotations impact example use, (2) whether successful analogical mappings correlate with task success, and (3) whether programming behavior can be used to predict success.

For the study, we asked participants to complete twelve experimental programming and mapping tasks, equally divided between similar and different examples. Participants were randomly assigned to use examples with one of our three annotation styles or no annotations throughout all tasks. Participants completed the tasks in Looking Glass, a blocks-based programming environment for creating 3D animations, designed for middle school aged children (see Figure 4). In this section, we discuss the reasons for our study design and the details of how we ran the study.

3.1. Study Design Rationale

In this section, we explain how we chose our experimental materials (example similarity and annotations) and why we decided to look at how analogical mappings and behaviors relate to task success or failure.

3.1.1. Example Task Design

We decided to vary the similarity between examples and task code to simulate the natural variety that would occur in found examples online. However, analogical reasoning research suggests that learners are more successful at completing problem solving tasks when they have an example that is similar to the target problem [17]. This work demonstrates the utility of two kinds of similarity: surface similarity refers to the correspondence between superficial features of the problem and example; structural similarity refers to the correspondence between the operations necessary to solve the problem and example [17]. To explore the impact of similarity, we created two kinds of tasks: similar and dissimilar example tasks.

The similar examples and programs have both structural and surface similarity because they share a similar code structure. For instance, in the similar task and example in Figure 1, the example and the solution use the exact same structure, a *Do together* block with two nested statements. For the dissimilar example tasks, the structures in the examples differ greatly from the solutions. As shown in Figure 1, the example has a *Do together* block with three statements, however the solution requires two *Do together* blocks each with two statements. Additionally, the dissimilar task and example differ in the types and number of objects used in the *Do together* blocks. We hypothesized that participants would be more successful at completing tasks with similar examples than tasks with dissimilar examples.

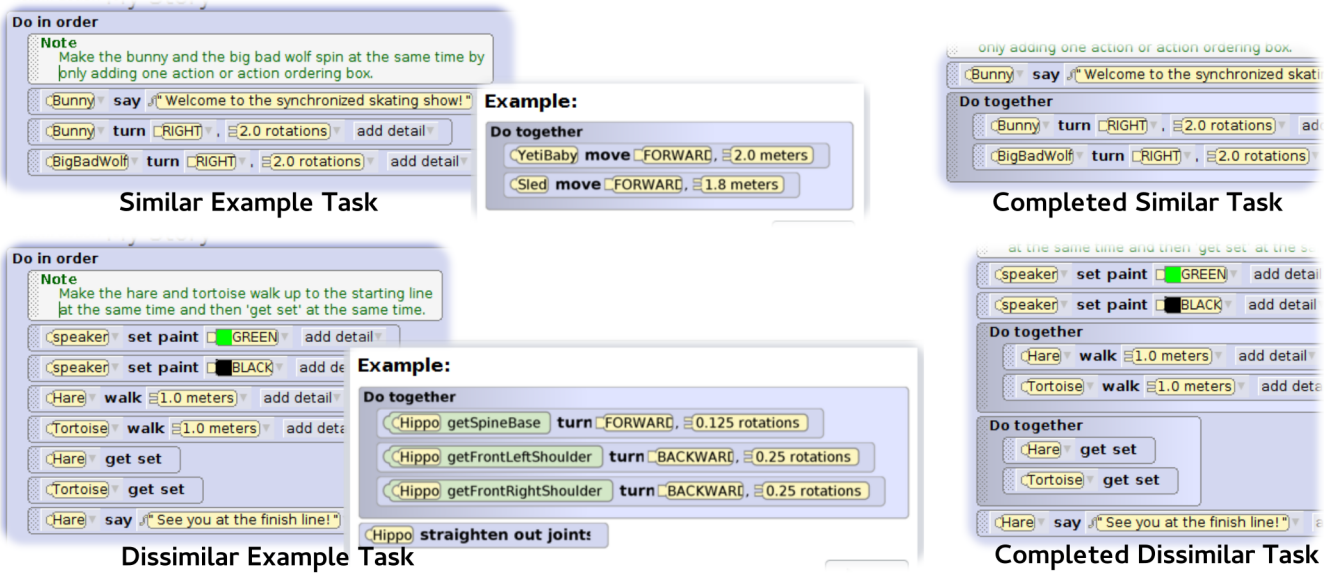


Figure 1: The similar and dissimilar example tasks for the simple parallel execution (*Do together*) task.

3.1.2. Example Annotations

We decided to compare example annotations because they are a common feature of found examples online, as well as in examples provided by support systems. Figure 2 shows the three annotation styles we selected and an example with no annotation (the control condition).

Below we describe the four annotation styles used in this study:

Brief Summary: provides a high-level description of code behavior, but it does not link explanations to individual lines of code. We selected this type of annotation because it is used in other systems and example work [15, 40]. We believe brief summaries could be useful in solving programming tasks because they can help a user to understand the overall behavior of the code, which could help with forming mappings and solving tasks.

Line-Specific Notes: provides descriptions of each salient line, shown near the associated code. We also selected this style as a comparison because it is commonly used for programming examples [16, 30, 41, 57]. Furthermore, line-specific notes could help a user who is confused about how a certain construct works to view information specifically about that part of the code.

Visual Emphasis: provides a red highlighted outline around the critical element or elements of the example. We designed this annotation based on formative testing, in which we found that simply circling the important part of an example in red helped novices to solve programming tasks. This could be because programmers can test the correctness of their code, so this annotation could provide them with enough information to identify the code elements involved in the solution so that they could then use that to try out possible solutions. A benefit of this annotation style is that it is quick to create and is not based on writing style, reading comprehension, or language.

No annotation (Control): the code example is shown without any textual information or highlighting.

3.1.3. Example-Task Analogical Mapping

Based on the idea that analogical problem solving is similar to completing a programming task using an example, we hypothesized that having novice programmers complete an analogical mapping task might provide insight into whether they correctly completed the task.

Cognitive psychologists define analogical problem solving as using one provided problem and solution (the base) to solve another problem (the target) [17, 18]. We believe that novice programming with examples is most closely related to analogical reasoning in mathematics [69]. To illustrate the model of analogical problem solving using mathematics, imagine a student solving problem 3x+2 = 11 using an example 2x-4 = 6, as shown in Figure 3. The student must first map the parts of the task and example that are related. In this case, the 3x and 2x are related, the + 2 and - 4 are related, and the =11 and =5 are related. The example solution might begin with moving 4 to the opposite side of the equation. Using the mapping, the learner would similarly subtract the 2 from both sides of the target equation. The mappings allow learners to adapt the sequence of steps necessary to solve the problem to the context of the target problem. In this case, the base and target problems have high surface similarity, meaning that the words are similar, making it easy to map the two [70].

Prior work is divided on whether the ability to generate correct mappings can predict task success. Gentner [18] describes the structure-mapping theory, which argues that there is a set of relations in the base problem that is also true for the target. The analogy is a mapping between the set of relations for the base and target problems. Gentner's work suggests that the primary difficulty associated with solving a problem using an analogy comes from mapping the example and the target. If participants

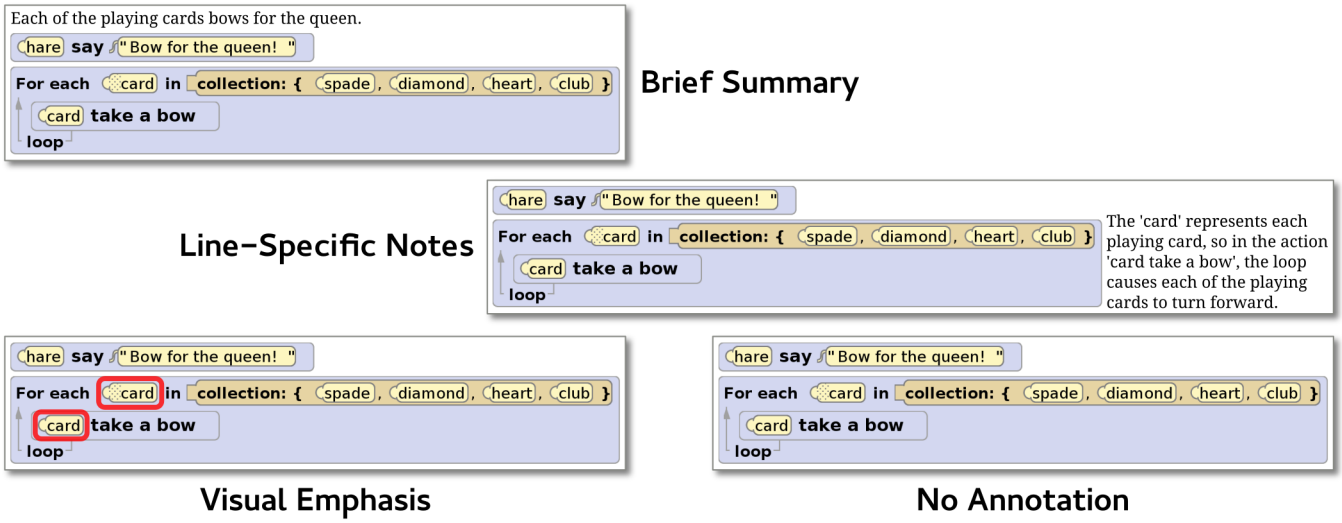


Figure 2: An example shown with three different styles of annotation and with no annotation.

can correctly map the example and target, they are highly likely to correctly solve the problem. In contrast, Novick and Holyoak's research [71] in the context of mathematical problem solving suggests that while mapping the example and target problems is necessary, it may not be sufficient to enable a learner to solve a problem using an analogy. In particular, when learners need to adapt the example to fit their target problem, some learners may succeed at mapping but struggle to construct a full solution [71].

Programming using an example shares some similarities with analogical reasoning in mathematics: novice programmers attempt to use a completed solution, in the form of an example to develop a related solution. However, there are also two important differences between analogies in mathematics and programming. First, novice programmers can incorporate testing into their problem solving process. Second, programming examples are not analogies in the traditional sense. Generally, an analogy provides both an analogous problem and the sequence of steps necessary to arrive at the solution. However, example code is essentially the completed solution. Because of these differences, it is important to evaluate how analogical mappings in programming relate to task success.

In order to understand whether correct mappings correlate

Problem

Solve for x? $3x + 2 = 11$

Example

Solve for x? $2x - 4 = 6$

$2x = 10$

$x = 5$

Figure 3: An example of analogical problem solving.

with success for novice programmers, we needed to collect mapping and task performance. Mapping can be operationalized as the ability to define relationships between elements in the example and target [17]. If a mapping is achieved, it occurs at some point during the task, possibly before the task is completed, making it difficult to collect mapping information without interrupting the task. We chose to collect the data after the task to prevent the data collection from changing the problem solving strategy.

3.1.4. Example-Based Problem Solving Process

While previous work suggests that inexperienced programmers often struggle to make effective use of example code, relatively little is known about how novices attempt to solve problems using example code and what kinds of behaviors predict success or failure. Characterizing predictive behaviors may help to identify opportunities for future systems to better support example use. For instance, knowing whether difficulty finding specific code blocks is linked to success or failure can indicate how big of a hurdle the programming environment is in solving a task. This can be accomplished using log data from the programming tasks and using decision trees to understand which features predict success and failure. Decision trees are often used for prediction across many domains, and have been successfully used in human-computer interaction, such as in predicting interruptability [72]. We selected this analysis method after the study was complete to answer questions about programming behavior generated by the example-task mapping analysis.

3.2. Study Methods

We gave participants 90 minutes to complete a computing history survey, a training task, and 12 programming tasks with examples. The computing history survey asked participants about their experiences using computers and with programming in the past to confirm that they were eligible for the

study. Each participant was assigned to one of the four annotation conditions. For this evaluation, we used the novice programming environment, Looking Glass [3]. Looking Glass has similar complexity to other blocks-based programming environments in many respects, like having blocks organized in palettes.

3.2.1. Participants

We recruited 99 participants between the ages of 10 and 15 for our study through the Academy of Science of St. Louis mailing list. The Academy of Science of St. Louis is an organization that provides opportunities for members to participate in science and technology programs city-wide. Community members also forwarded our email to a newspaper and a homeschool message board on their own. Since this work aims to address the problems of novice programmers who do not have access to formal computer science education in schools, we asked that participants have "minimal" programming experience, which we defined as 3 or fewer hours. This limit on the number of hours participants had coding also ensures that the participants did not have prior experience using the programming concepts in the tasks.

We analyzed the data for 80 participants (33 female, 47 male, age: M = 11.8, SD = 1.3). Each participant received a $10 gift card for Amazon.com in recognition of his or her participation. We excluded 19 participants: 13 had more than minimal programming experience (i.e. had programmed for more than three hours); 4 did not complete the study within the allotted time; and we had 2 study administration mistakes.

3.2.2. Training Task

We asked participants to complete a training task designed to help familiarize them with the study format and the basic mechanics of the Looking Glass programming environment. The training task had two parts: (1) participants completed a simple program using an example, (2) participants mapped the example and training program on paper.

To complete the training program, participants assembled a simple three-line program in the correct order using an onscreen example. While completing the training program, participants could reference a mechanics help sheet that provided an overview of creating a simple program within the programming environment. This task was designed to introduce participants to the format of the programming tasks and basic interface mechanics. Through pilot tests, we found that this introduction helped to reduce the number of interface and task-related problems during the experimental tasks. Participants were free to reference the mechanics help sheet throughout all study tasks. There was no time limit for this task, but if participants seemed stuck, a researcher helped them to complete the task. Participants were also allowed to ask questions during this task.

After completing the training program, participants also completed a paper-based analogical mapping task. This mapping task asked participants to draw lines connecting elements in the example code with the introductory program (see Figure 5). As with the program training task, the mapping training task familiarized participants with the task instructions.

3.2.3. Programming Tasks with Examples

We next asked participants to complete twelve programming tasks using examples. The tasks covered six programming concepts, listed here from easy to difficult: simple parallel execution, using a *for loop*, using an iterator within a *for each loop*, using an advanced API method unique to Looking Glass, setting a conditional for a *while loop*, and using a function's return value as an argument to a method call. For each programming concept, we developed similar and dissimilar example tasks. See Figure 1 for the similar and dissimilar example programming tasks for the simple parallel execution concept.

We chose programming concepts that greatly varied in difficulty to help provide insight into how concept difficulty affects novices' abilities to complete the tasks. We designed the tasks with the understanding that many would be challenging, especially for novice programmers with minimal programming experience, leading to an expected low overall task performance. This was purposeful because we wanted to explore both the times when novices succeeded using examples as well as when they had difficulties. As prior research suggests that novices are often unsuccessful in using examples, we included a significant proportion of tasks that our pilot tests indicated would have a low success rate in order to capture the challenges novices face when using examples.

As in the training task, each programming task consisted of: (1) modifying a program using an on-screen example (see Figure 4), and (2) completing a paper-based analogical mapping task (see Figure 5).

To complete each programming task, participants needed to modify an existing program to achieve a stated goal while meeting task constraints intended to ensure use of the targeted programming concept. For example, in looping tasks, we required that participants only add a certain number of code blocks in order to force them to use a loop in order to correctly complete the task. Figure 4 shows one programming task used in the study. Participants had at most five minutes to complete each programming task. If participants stated that they finished the task early, a researcher asked them if they were sure that they fulfilled all of the criteria of the task, but did not tell them if it was correct or incorrect. We did this to encourage participants to check their own work and try to make sure they fulfilled the criteria on their own, as pilot studies showed that participants sometimes did not always read directions carefully. We created the tasks and selected time limits based on formative and pilot testing.

To complete the mapping task, participants drew lines connecting code elements in the example to related elements in the program as shown in Figure 5. There was no time limit on the mapping task, but participants generally completed mapping tasks very quickly. Participants completed paper mappings after each programming task to prevent the mapping task from

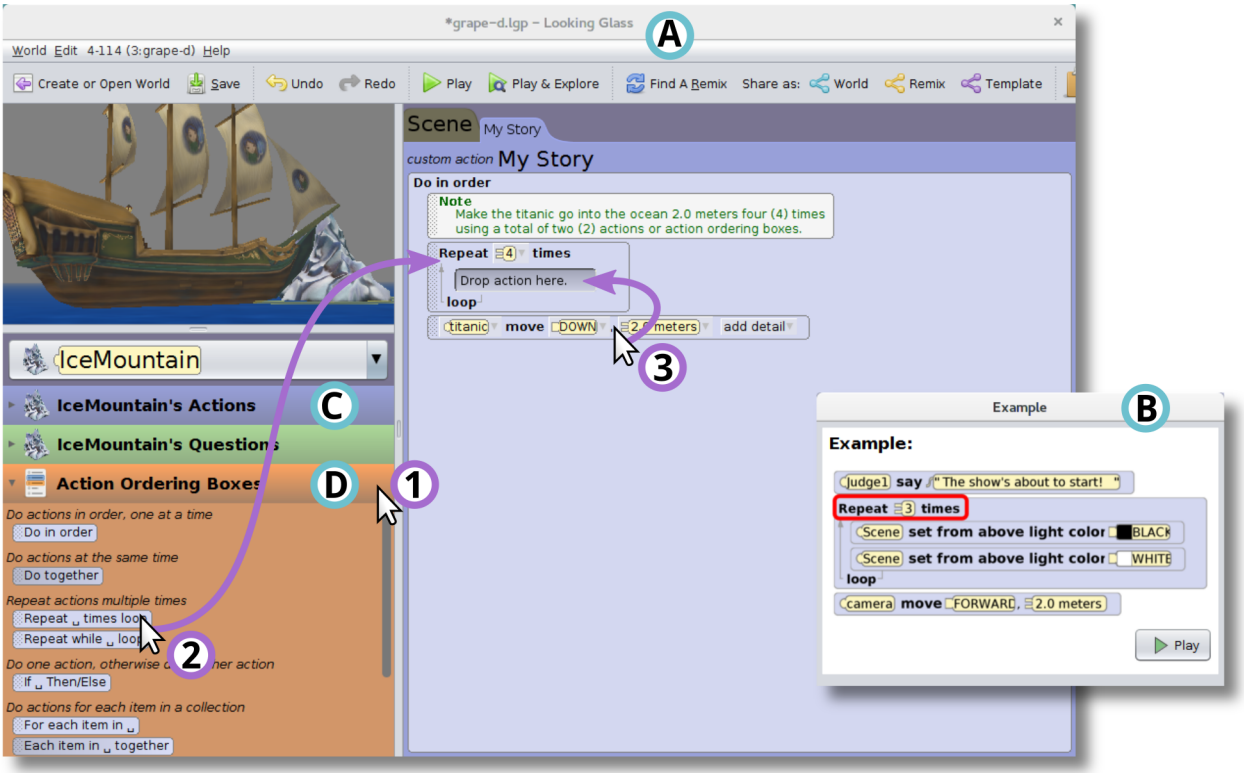


Figure 4: The dissimilar repeat task shown in the (A) Looking Glass programming environment with a (B) dissimilar example. Initially the (C) actions tab is selected. There are three steps to complete the task, shown with pointers in the figure. Step (1): the user clicks on the (D) constructs tab to transition from the *Exploring & Searching Stage* to the *Ready-to-Program Stage*. Step (2): the user drags the repeat construct into the code editor to transition from the *Ready-to-Program* to *Assembling Stage*. Step (3): the user drags the move statement into the repeat construct to correctly complete the task.

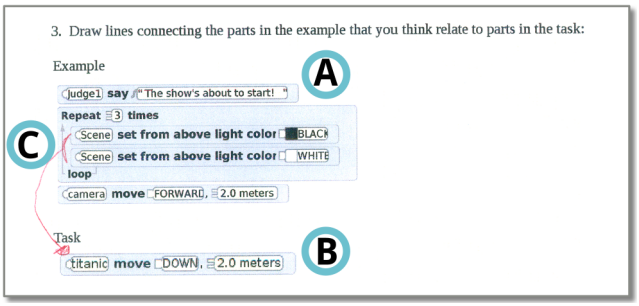


Figure 5: The dissimilar repeat mapping task completed by a study participant. We asked participants to connect the (A) example shown in the task with (B) the initial state of the program by (C) drawing lines connecting the two.

influencing their problem solving process. We acknowledge, however, that collecting mappings at the end of the task could overestimate how many participants had correct mappings during the tasks. Furthermore, it is possible that not having a time limit on mapping tasks may have allowed participants to figure out the mapping at the end of the task.

To account for learning effects, we used a Latin squares design to assign task orderings across participants. We varied the order of the six programming concepts using a balanced 6 x 6 Latin square. For each programming concept, a participant first completed a similar or dissimilar example task for that concept, immediately followed by the other example similarity type. For the six concepts, each participant completed three with a similar example task followed by a dissimilar example task and three with a dissimilar example task followed by a similar example task. Each participant saw one of the four example annotation types for all 12 tasks.

4. Data and Analysis

We analyzed three types of data: program performance, example-target mappings, and programming behavior.

4.1. Program Performance

We saved participants' task programs when they finished the task, which was when they either stated that they were finished or timed out after five minutes. We graded these final state programs for correctness using criteria based on previous work scoring similar programming tasks [17]. We assigned points for (1) correct usage of the target programming construct, (2) correct placement of the provided code statements relative to the target programming concept, and (3) a lack of extraneous changes. We created rubrics in which participants earned one point for each correct attribute in these three categories. One researcher then scored each participant's programs using the rubric.

User Interface Behaviors
Number of UI actions: User interface actions such as exploring a drop down menu, changing a tab stage, etc.
Time performing UI actions: Continuous blocks of user interface actions spaced closer than one second apart.
Number of example actions: The number of times participants interacted with the example (i.e. by mousing over or playing it). It is important to note that this is, at best, an imperfect indicator of example engagement. While some users moused over or moved the example while studying it, we also observed participants who were clearly studying the example without use of the mouse.
Time interacting with example: Continuous blocks of interacting with the example using the mouse, spaced closer than one second apart.
Idle time: The amount of time the participant spent not doing anything. We computed this by looking for periods where the time between events was greater than fifteen seconds.
Number of scene edits: The number of times the participant changes the scene layout (e.g. moving the camera or changing the position of a 3D model). None of the tasks required that participants edit the scene, so this is always off-task behavior.
Code Editing Behaviors
Number of irrelevant edits : An irrelevant code edit does not make any progress towards a correct solution and does not touch any code elements used in achieving a correct solution.
Number of semi-relevant edits: A semi-relevant edit modifies an element of the code that is involved in a correct solution, but does so in a way that does not make progress towards a correct solution. For example, in the case where a participant needs to replace a numeric parameter with a call to a function, a semi-relevant edit might change that numeric parameter to a different numeric value.
Number of relevant edits: The number of relevant edits and the time at which the participant first made a relevant edit. We define a relevant edit as one that makes progress towards a correct solution.
Number of edits: This included all code edits, regardless of type.
Number of tinkering edits: We noticed that when participants gave up on solving tasks, they appeared to frequently transition to changing unrelated parameter values or experimenting with keyed parameters (e.g. animation styles, duration, etc). This often takes the form of "what does this do?" style experimentation. We refer to these types of changes as tinkering edits.
Number of executions: The number of times the participant executes their program.

Table 1: Programming behavior features

4.2. Example-Target Mappings

To analyze the mapping tasks, two authors independently transcribed which components the lines were drawn between for 14% of participants, reaching high agreement (Cohen's $\kappa > .61$) for the mappings ($\kappa = .794$, $p < .001$). The authors then worked independently to transcribe the remaining participants' mappings. This was necessary because in a small number of cases, it was slightly ambiguous which blocks the mappings were connecting. We developed a set of correct mappings consistent with the program solution and recorded whether the transcribed mappings contained one or more of these correct mappings, similar to the analysis from Spellman et al. [73].

4.3. Programming Behavior

In addition to mappings and program performance, we also recorded log files of the actions that participants took while working on each task. This includes data such as interface interactions like opening a palette of blocks, time spent with their mouse interacting with example, and program modifications, as shown in Table 1. We use this log information to compute program solution progress, which we will discuss throughout our results section to provide more granularity than just success or failure. Our log parser analyzed participants' code editing behavior when working towards the solution of a task. Because each task has a known solution and a known interface state to reach that solution, we were able to ascertain whether their actions were relevant to the solution or were in no way related to the task. We also investigate whether this log data can predict success or failure on tasks using decision trees, which we will go into greater detail in the results section.

4.3.1. Program Solution Progress

The majority of our tasks required that participants work through a series of stages in order to arrive at a solution, which we will discuss throughout the results section. These stages are four key parts of a task in a blocks-based environment that we hypothesized could provide us with more in-depth information about where novices struggled:

Exploring & Searching Stage: The user has made no progress towards a solution. Typically, the user is exploring and searching the interface in an effort to advance their task. This stage may include both code changes and user interface actions that do not advance the user towards a solution. This is an essential part of many new programmers' experiences in blocks-based environments, where rather than being able to type a command, they must explore the environment in order to find the blocks they need to program.

Ready-to-Program Stage: The user interface is in a state where the program elements necessary to solve the problem are accessible. For example, if the user needs to add a loop to solve a problem, the palette containing the loop code block is visible in the interface, as shown in Figure 4-(1). For a blocks-based environment, this demonstrates progress toward solving a task because it likely indicates that the programmer has correctly determined which block they need and have also found it in the interface.

Assembling Stage: Any needed program elements have been added to the program, but the code is incorrect. Continuing the loop example, the loop is in the user's program but is incorrectly placed or does not contain all of the required statements, as shown in Figure 4-(2). At this point, in a blocks-based environment, all the programmer must do is re-arrange the blocks to create the correct solution.

Completed Stage: The program is completed and correct, as shown in Figure 4 after the action in (3).

For each task, we record the time at which a user first reaches each stage. We note that it is possible that a user can regress to a previous stage. This commonly happens with the *Ready-to-Program* stage because the user may change the state of the interface away from being ready to solve the problem. Our stage-based model was a post-hoc analysis designed to more deeply understand the behaviors affecting task success. For our stage-based analysis we excluded two programming concepts: using an iterator within a *for each loop* and calling an API method (four of the twelve tasks in total). These four tasks initially begin in the *Ready-to-Program Stage*, instead of the first stage, *Exploring & Searching* because the participants did not need to change the state of the interface to find the component needed to complete the task. We believe that analyzing the remaining 8 tasks for this part makes sense because in order to best understand exactly where the problems are taking place, we want to look at tasks where all four stages are required for successful task completion. All results unrelated to our stage-based analysis are based on the data from all twelve tasks.

5. Results

We seek to answer three questions: (1) how does the interaction of annotations and example similarity affect novice programmers' performance on tasks using examples, (2) to what degree does the ability to map an example and target problem correlate with task success, and (3) to what degree do programming environment and coding behaviors predict task success?

5.1. How do novice programmers perform on tasks using examples and how do annotations and example similarity affect performance?

To answer this, we discuss the overall task success, the effects of annotations on task success, and the effects of example-task similarity on task success.

5.1.1. Overall Task Performance

Overall, participants completed 30.3% of tasks correctly. This result generally aligns with our expectation and the idea that novices often struggle completing tasks using examples. We will discuss this result further later in the paper. When completing most tasks, most participants did not move past the *Exploring & Searching Stage* (51.6%). Conversely, most participants failed to reach the correct solution (only 30.3% succeeded). Only 9.8% of tasks ended at the *Ready-To-Program Stage* and 8.2% of tasks ended at the *Assembling Stage* (as seen in Table 2). We also investigated whether gender may play a role in task performance by including it as a covariate in our analysis based on past research on gender in end-user programming [74]. However, we found no significant gender differences in any of the statistical tests we ran.

Stage	% Tasks	% Correct Mappings	% Tasks	% Correct Mappings
	Ended at Stage		Reached Stage	
Exploring & Searching	51.6%	45.6%	100%	55.2%
Ready-to-Program	9.8%	50.0%	48.4%	65.4%
Assembling	8.2%	56.8%	38.6%	69.3%
Completed	30.3%	72.7%	30.3%	72.7%

Table 2: This table shows the percentages of tasks and mappings that ended at each stage and reached each stage. The percentage that ended at each stage shows how many tasks were at a stage when the tasks were over. The percentage that reached each stage demonstrates the amount of tasks that that got to each of the stages.

5.1.2. Effects of Annotations

Using MANCOVA and Roy's largest root, there is a significant effect of annotations versus no annotation on program performance, Θ = .35, F(12, 66) = 1.93, $p < .05$. The three annotation conditions outperformed the no annotation condition. Separate univariate ANCOVAs revealed that there are no significant differences between the three annotation styles.

This effect was also present for the similar example tasks; participants who used similar examples with any annotation style significantly out-performed participants without any annotations, Θ = .21, F(6, 72) = 2.52, $p < .05$. However, we found no significant effect for dissimilar examples and annotation styles. We would have expected annotations to assist novice programmers in solving dissimilar example problems, since the annotations help novice programmers to map examples and problems. For dissimilar example tasks, participants may have needed more time to understand the mapping between the task and the example, which prevented them from having time to actually complete the task, though they may have realized how to complete it by the end of the task time.

We also wanted to know whether the annotations had any effect on which stage participants made it to for their tasks. We computed the percent of the tasks that finished in each stage, for each participant. Because the *Completed Stage* is functionally equivalent to program correctness results presented above, we report whether the annotations had any effect in the earlier stages. Using a MANCOVA and Roy's largest root, there is a significant effect of annotation versus no annotation on the stage participants reached, Θ = .10, F(3, 81) = 2.74, $p < .05$. Compared to the no annotation condition, participants with annotations were slightly more likely to finish their tasks at a higher stage than the no annotation condition. Separate univariate ANCOVAs revealed that there are no significant differences between the three annotation styles.

5.1.3. Effects of Example-Task Similarity

As predicted, participants correctly completed more tasks using similar examples (Mdn = 2.2 of 6 points) than dissimilar examples (Mdn = 1.83 of 6 points), $p < .01$, r = -.34. The low median task scores align with the low overall success rate. Using a MANCOVA and Roy's largest root, there is a significant effect of example-task similarity on the stage,

Θ = .06, F(2, 168) = 4.78, $p < .01$. We will discuss how annotations and example-task similarity affect mappings in the next section.

5.2. To what degree does the ability to map an example and target problem correlate with task success?

First we discuss the overall results for mapping and task success as well as example similarity, which both support the idea that mapping and task success are related. However, it turns out that mappings and performance actually have a relatively low correlation. Finally, we describe behaviors that may be influencing the low correlation.

5.2.1. Overall Mapping and Task Success

In the majority of tasks, participants either made no measurable progress towards a solution (51.6% ended the task in the *Exploring & Searching Stage*) or correctly completed them (30.3% ended the task in the *Completed Stage*). Table 2 shows the percentage of the total tasks that reached each stage and the proportions of tasks for the stages that had correct mappings. While there are some instances in which participants have identified the necessary code elements and failed to arrive at a fully correct solution (ending the task in the *Assembling Stage*), the ability to modify and test a program appears to enable those who successfully added the needed code element to complete the task. We found an increasing proportion of correct mappings for tasks ending in the later stages. For tasks ending in the *Exploring & Searching Stage*, 45.6% of tasks had correct mappings. By the *Completed Stage*, 72.6% of tasks had correct mappings. This trend is consistent with correct mappings contributing to task success.

5.2.2. Annotation and Mapping Results

Using MANCOVA and Roy's largest root, there is a significant effect of annotations versus no annotation on correct mapping of the example to the program, Θ = .38, F(12, 66) = 2.10, $p < .05$. Separate univariate ANCOVAs revealed that there are no significant differences between the three annotation styles. Compared to the no annotation condition, all participants whose examples had annotations constructed correct example-program mappings more often. There is also a significant effect of annotations versus no annotation on correct mappings when looking at both the similar example tasks, Θ = .23, F(6, 72) = 2.73, $p < .05$, and the dissimilar example tasks, Θ = .25, F(6,72) = 3.01, $p < .05$.

This suggests that even for similar examples, annotations are important in helping novices understand mappings. While this is what we expected for dissimilar tasks, as the annotations can help to fill in information missing when there is low surface similarity, this is unexpected for the similar example tasks. We would expect the surface similarity in the similar example tasks to make them doable without annotations. We believe one reason for this may be the difficulty level of some tasks. When a novice programmer is working on a task that is beyond their current level of understanding, surface similarity may not be enough to assist in understanding the correlation between an example and a problem, but an annotation can improve this.

5.2.3. Example Similarity and Mapping

A Wilcoxon Signed-Ranks test revealed that participants were more successful at similar example mapping (Mdn = 5 of 6 correct mappings) than dissimilar example mapping (Mdn = 3 of 6 correct mappings), $p < .001$, r = -.65.

5.2.4. Mapping and Task Success Connection to Analogical Reasoning

While the upward trend of mappings for each stage and the similar and dissimilar example results start to support a relationship between mapping and task success, we found only a weak correlation, rb = .21, $p < .001$. This low correlation provides some support for Novick and Holyoak's findings that mappings are necessary but not sufficient for problem solving [71]. However, it is worth noting that nearly 28% of fully correct tasks did not have correct mappings, which suggests that some participants may be solving tasks using a strategy outside of analogical reasoning.

If Gentner's structure-mapping theory holds for programming, we should have seen a strong correlation between mapping success and program success. Structure-mapping theory states that the primary difficulty problem solving using analogies comes through mapping the problems. If learners can successfully map the problems, they should be able to correctly solve the target problem. In that scenario, we would only observe problems with executing that plan such as difficulty finding a needed program element. Instead, we saw a weak correlation. We explore possible reasons behind the weak correlation using the stages of task completion:

1. Correct mappings and incorrect tasks:
   - Participants may have developed their mappings too late in the task to use them. This could happen as a result of attempting to solve the problem without using the example initially, either through a desire to complete the task independently or because of difficulties understanding the example (discussed in Sec. 5.2.5 **Correct Mappings and Incomplete Solution Plans**). This is suggested by a large number of irrelevant edits in the early stages.
   - Participants may have generated full plans based on the mappings between the example and target problems but struggled to execute those plans within the programming environment (discussed in Sec. 5.2.6 **Correct Mappings and Difficulties Executing Solution Plans**). Many user interface actions in the early stages of a task could support that users were searching for how to execute their solution plans.
2. Incorrect mapping and correct task:
   - Though we based our mapping task on those used in psychology studies in the past, there are differences in blocks programming environments that

may have made the mapping task unclear (discussed in Sec. 5.2.7 **Mapping Task Design**).

We now describe each of these three cases and the data that supports these as possible reasons for the low correlation between mapping and performance.

5.2.5. Correct Mappings and Incomplete Solution Plans

For 25.4% of tasks with correct mappings, participants' behavior suggests that they began with an incomplete solution plan and tested multiple variations to arrive at a solution. Adding the missing code element for a problem marks the boundary between forming a plan and beginning to carry out that plan. In our stage-based model, participants implemented their solution plans during the *Assembling Stage*.

If participants shaped solutions through working with the programming environment, we would expect to see more testing behavior through higher numbers of edits and program executions. To explore this, we divided the tasks in the *Assembling Stage* with correct mappings into low editing (0 or 1 edits beyond those necessary to solve the task's problem) and high editing (two or more extra edits) groups. A Wilcoxon Signed-Ranks test revealed that there is a significant difference in the number of program executions between the high (Mdn = 2) and low (Mdn = 1) editing groups, $p < .001$, r = -.38. Behavior in the low editing group (74.6% of tasks) is consistent with Gentner's separation between planning a problem solution and executing it [70]. Behavior in the high editing group suggests that for 25.4% of tasks, participants began to execute an incomplete solution plan, supporting Novick and Holyoak's findings that mappings are not always sufficient to enable problem solving [71]. These types of difficulties spurred our third question about the ability of programming behavior using examples to predict success or failure.

5.2.6. Correct Mappings and Difficulties Executing Solution Plans

Difficulties executing a solution plan may explain some task failures, but we failed to find evidence suggesting that execution difficulties are a large source of task failures.

If a participant is able to generate a plan but struggles to execute their plan, we would expect that task to end in the *Exploring & Searching Stage* (no progress) or *Ready-to-Program Stage* (interface correct) with correct mappings and a higher rate of user interface actions due to search behavior. A Wilcoxon Signed-Ranks test revealed that there is a marginally significant difference between the number of user interface actions among tasks ending in the *Exploring & Searching Stage* and the *Ready-to-Program Stage* with correct mappings (Mdn = 110) and without correct mappings (Mdn = 84), p = .05, r = -.14. This suggests that there are likely some tasks in which participants had a plan but struggled to execute it. However, it seems unlikely that difficulties executing a solution plan account for a large proportion of failed tasks.

5.2.7. Mapping Task Design

In 27.3% of the tasks that reached the *Completed Stage* (9.1% of all tasks), participants produced incorrect mappings but correctly completed the program. Based on reviewing a random selection of 20% of these mappings, we observed that:

- Some participants struggled to represent mappings where a code element present in the example was related to something that needed to be added to their program. This meant that one correct mapping was to map a block in the example to an empty space in the task code. It was not clearly specified how to do this, so this was a weakness of the mapping task design. Since many tasks require that participants add programming constructs, we may need to explore alternative methods for capturing these mappings.
- In some cases, participants mapped sub-elements of a statement rather than full code statements. For example, a participant might map the method callers, names, and parameters for two statements. We did not rate sub-element mappings as correct, even when a mapping between their parent statements was correct. In these cases, it seemed more likely that participants were drawing lines between everything that was in the same location in the code, rather than understanding that the methods as a whole were the important related components. However, it is possible that participants who created sub-element mappings may have correctly understood the code.

5.3. Where in the process of solving a programming problem using an example do novices struggle and which behaviors predict success and failure?

In order to understand what challenges participants were having, we wanted to determine what kinds of programming behavior predict success and failure at each of the stages (*Exploring & Searching*, *Ready-to-Program*, *Assembling*, and *Completed*). To do this, we (1) identified predictive features from among the programming behavior features, as shown in Table 3, and then (2) used the subset of predictive features to train a decision tree that predicts successful completion of each of the stages in our stage model.

In this process, we used two classifiers: random forests and decision trees. A random forest is "a classifier consisting of a collection of tree-structured classifiers" where the input to each of the classifiers is an independent identically distributed random vector and each classifier "votes" for the most popular [75]. In our analysis, we used R's randomForest package, which implements Breiman and Cutler's algorithm for random forests [76]. A decision tree is a classifier that partitions the space based on the values of the internal nodes. Each leaf of the tree has the most likely target value based on the paths from the root that reach that leaf.

To identify predictive features for each stage, we trained a random forest of 500 trees using a combination of performance based features (see Table 1) and demographic features. The demographic features included age, condition, and gender.

Stage	Predictive Features(%MSE explained)
Exploring & Searching	Idle Time (40.26) Num. of Runs (28.94) Num. of Code Edits (28.73) Num. of UI Actions (19.47) Num. of Tinker Edits (16.18) UI Time (15.08) Num. of Irrelevant Code Edits (13.78) Num. of Semi-relevant Code Edits (11.23) Age (10.16)
Ready-to-Program	UI Correct Time (27.95) Num. of Runs (20.24) Num. of UI Actions (17.21) UI Time (14.04) Idle Time (13.41) Num. of Code Edits (12.80) Example Time (10.74) Num. of Irrelevant Edits (10.47)
Assembling	Num. of UI Actions (16.63) UI Time (13.47) Num. of Irrelevant Edits (12.63) Num. of Relevant Edits (10.60) Num. of Tinker Edits (10.11)

Table 3: Predictive features for each stage

Note that all of the performance features are stage-specific. In predicting which tasks would achieve the *Ready-to-Program Stage*, we used only performance features for the previous stage, the *Exploring & Searching Stage*. Then, for each of the forests, we examined the variable importance statistics and identified the subset of features that improved the mean-squared error by more than 10% for use in constructing the decision tree.

Next, we trained an individual decision tree for each stage using our selected subset of features (see Table 3). We use the resulting decision trees to pull out behavioral differences between successful and unsuccessful participants at each stage. In training both the random forests and decision trees, we included only tasks that achieved the previous stage. Notice that most of the features used for this analysis do not explicitly focus on the example, but instead attempt to measure the behaviors that indicate difficulties using the example. There are several reasons for this: (1) we did did not use eye-tracking, so the main ways to measure example use were mouse movements and example executions, and (2) participants very rarely executed the example code.

First, we will discuss how we assessed our decision tree models and then we will explore the decision trees for each of the stages and discuss the features that predict success and failure for those stages.

5.3.1. Decision Tree Model Quality

We assessed the quality of our three decision tree models in two ways:

First, we constructed a Baseline Model that always predicts the most common classification (success or failure) for that stage. Using a binomial test, we evaluate whether the decision tree performs significantly better than this baseline (see Table 4). We acknowledge that comparing to the Baseline Model is a relatively weak test of significance.

To provide additional insight, we also constructed a Null Mixed Logistic Regression model with two random factors: task ID and participant ID. This model leverages the fact that knowing the difficulty of the task and the general performance of a participant is often sufficient to make good performance predictions. As expected, the Null Mixed Logistic Regression models perform fairly well. It is important to note that they leverage task and participant information that we intentionally excluded from our decision tree model. Yet, the decision tree models achieve similar accuracy using purely behavioral data (see Table 4).

In training both the random forests and decision trees, we included only the subset of tasks that successfully achieved the previous stage.

5.3.2. Predicting Ready-to-Program Stage Success

Figure 6 shows the decision tree that predicts whether a given task will achieve the *Ready-to-Program Stage* (correct interface state) given the programming behavior during the *Exploring & Searching Stage*. Any amount of code editing during the *Exploring & Searching Stage* is a strong predictor that task will not achieve the *Ready-to-Program Stage*. Specifically, tasks without the *Exploring & Searching Stage* code editing successfully reach the *Ready-to-Program Stage* 91% of the time; those with code editing successfully reach the *Ready-to-Program Stage* only 20% of the time. Among the tasks without code editing, those with task idle times of more than two minutes are dramatically less successful, reaching the *Ready-to-Program Stage* only 25% of the time. This behavior may indicate that participants did not know what to do and were reluctant to explore. Among the tasks with code edits during the *Exploring & Searching Stage*, running the program once or not at all increased the chances of successfully reaching the *Ready-to-Program Stage* to 60%. These participants made and tested a small number of changes before moving on to searching for the necessary code elements in the interface.

5.3.3. Predicting Assembling Stage Success

A task successfully reaches the *Assembling Stage* when the participant adds a code element needed for task completion. Figure 7 shows the decision tree that predicts success at reaching the *Assembling Stage* based on the programming behavior in the *Ready-to-Program Stage*. The strongest predictor of successfully achieving the *Assembling Stage* is the number of user interface actions that occur in the previous stage. If a participant makes a large number of user interface actions (i.e. 26 or more), this may suggest that they reached the *Ready-to-Program Stage* (the correct interface state) by chance; 77% of these tasks end in the *Ready-to-Program Stage*. Additionally, tasks with a large number of user interface actions are less likely to have correct mappings. For tasks with 26 or more user interface actions, 52% have correct mappings. For those with fewer than 26 user interface actions, 69% have correct mappings. Finally, we note that if participants reached the

Stage	# of Tasks	Baseline Model Accuracy	Null Mixed Logistic Regression Accuracy	Decision Tree Model Accuracy
Ready-to-program	531	51.79%	88.89%	87.76%**
Assembling	256	80.08%	89.06%	92.97%**
Completed	205	78.05%	82.93%	84.39%*

Table 4: Decision tree model quality. **p< .0001, *p< .05

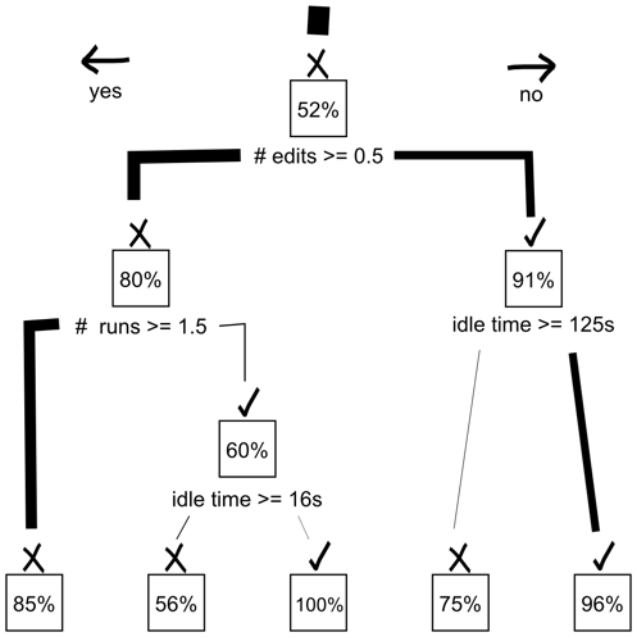


Figure 6: The decision tree predicting success at achieving the *Ready-to-Program Stage* based on *Exploring & Searching Stage* performance.
Yes/no: value of the inequalities.
Line thickness: percentage of tasks following each path.
Percentages in boxes: accuracy of each node.
'X': predict failure to reach stage.
Check marks: predict success at reaching stage.

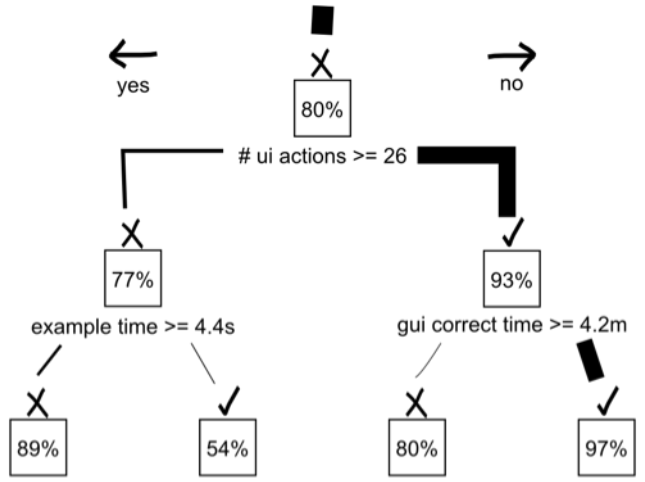


Figure 7: The decision tree predicting success at achieving the *Assembling Stage* based on the *Ready-to-Program Stage* performance.
Yes/no: value of the inequalities.
Line thickness: percentage of tasks following each path.
Percentages in boxes: accuracy of each node.
'X': predict failure to reach stage.
Check marks: predict success at reaching stage.

Ready-to-Program Stage late in the task, that often predicted failure.

5.3.4. Predicting Completed Stage Success

Figure 8 shows the decision tree that predicts success at reaching the *Completed Stage*, based on the programming behavior in the *Assembling Stage*. Effectively, tasks break into three categories based on the number of user interface actions. Overall, tasks with fewer than 35 user interface actions were most successful: 88% achieve a correct task solution. Tasks with a mid-range (ranging from 35 to 85) number of user interface actions were least successful, only 22% arrived at a correct solution. Interestingly, tasks with the highest number of user interface actions were more successful than those in the mid-range: 54% achieved a correct solution. We note that user interface actions are strongly correlated with code editing, $p < .001$, $r(205)$=.64.

Once participants reach the *Assembling Stage*, they have all of the code elements necessary to correctly solve the task. So, moving from the *Assembling Stage* to the *Completed Stage* is a matter of placing the code elements in the right positions. The group with the lowest number of user interface actions shows a more selective approach to making code changes. In the middle, participants made a larger number of edits, but likely with less deliberation about each individual change. Since even with relatively short programs, there are a large number of potential edits that can be made, this strategy tended towards failure. Finally, the group that made the largest number of code edits shows an increase in overall success rates. This may be a result of a fast guess and test approach.

6. Discussion

We first go into further discussion on each of our primary topics: (1) example annotations, (2) analogical reasoning, and (3) programming behavior analysis as compared to a qualitative study on example use. Then, we discuss the importance of how this work fits into the larger picture of blocks-based programming environments.

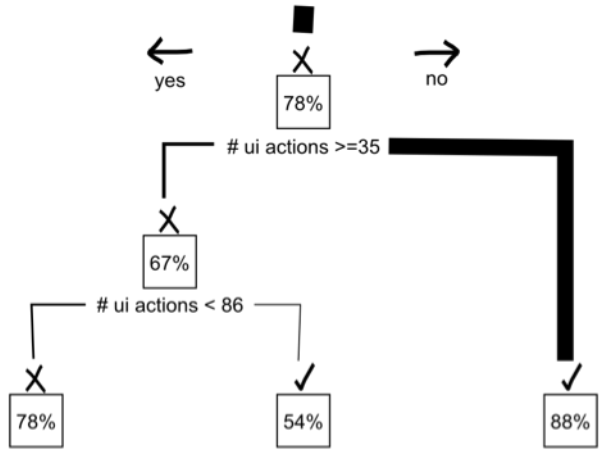


Figure 8: The decision tree predicting success at achieving the *Completed Stage* based on the *Assembling Stage* performance.
Yes/no: value of the inequalities.
Line thickness: percentage of tasks following each path.
Percentages in boxes: accuracy of each node.
'X': predict failure to reach stage.
Check marks: predict success at reaching stage.

6.1. High Failure Rates Regardless of Annotation

While we hypothesized that the different annotation conditions would provide different affordances for similar and dissimilar example mappings and performance, we did not find significant performance differences between the individual annotation styles for mapping or for task performance. One reason for this might be that each of the annotations provided extra information to lead participants to better mappings and solutions [17]. Yet, novices still had significant problems completing tasks in all conditions. Looking at the results, the two main issues causing low overall success were (1) inability to move beyond the *Exploring & Searching Stage*, and (2) incomplete solution plans with correct mappings. In order to help novice programmers in using examples, future work should address these two problems.

Our study results suggest that supporting novices in taking the first steps in using an example is crucial, similar to a finding that getting started is generally difficult for end-user programmers working on specified tasks [77]. In 51% of tasks, participants made no discernible progress towards a solution. Of those who made any progress, 62% arrived at the correct solution. One reason for this is that novice programmers may be overwhelmed at first, trying to figure out what to focus on and how to start. Only 45% of participants at this stage made correct mappings, so it may be critical to nudge novices back toward looking at and using an example if they have made a certain number of edits with no progress. Another issue might be that a programmer does not understand the example if their mapping is wrong, which could indicate the need for multiple examples of varying surface similarity, allowing novices to search for another example if the first one is confusing.

We showed that one reason behind failure is that once novice programmers reach the *Assembling Stage*, they often do not have a complete or correct plan for how to reach the solution. This means that although they understand the relationships between the task and the example, they do not know how to formulate a plan. The first step in helping novices in this situation is to be able to identify that they are having this specific issue. Based on our results, we believe systems should automatically keep track of whether a novice has added the correct component, whether they have reached a correct solution, and whether they have made multiple edits. Essentially, a system could leverage the stage model discussed in our results to keep track of progress and provide strategic advice. Future work could use this model as a starting point for educational strategies in programming, similar to the problem solving strategies introduced by Loksa et al. [78].

6.2. Programming Examples and Analogical Reasoning

In nearly 75% of tasks, participants' behavior is consistent with the prediction that mapping success is sufficient to enable task success. However, in the remaining 25% of tasks, participants made a series of code edits and program executions that suggests they did not have a full solution in mind when they began to make changes to the target program. If we can think of example code use as analogical problem solving, this opens up this topic to being able to apply other findings from analogical problem solving research to support for novice programming with examples. Two applications of analogical reasoning that could apply to novice programming are (1) using visual analogies as hints and (2) work on analogical reasoning across ages.

Research has found that visual analogies can be used as cues to hint at an analogy that had been presented earlier [79]. This could apply to novice programming because once a novice programmer has learned a concept once, they may still not realize that they should use it in another situation. In this case, a visual analogy might be useful because it would cue memory of previously provided information, without being repetitive. One study on programmers found that they naturally use examples as reminders [9], so this might be useful for novices as they are learning as well.

Work on analogical problem solving and the development of reasoning in children may also apply to example use in novice programming, since blocks-based environments exist for all ages. One study found that middle school students were stronger at solving analogy problems, but pre-schoolers were also capable of using an analogy to solve a new problem [80]. Further, middle school students often had similar performance as adults. This implies that we may be able to use similar support for middle school children and adults. Examples are also likely to be applicable for very young novice programmers, but the analogies must be very carefully selected [80].

6.3. Programming Behavior and Relation to Qualitative Study on Example Use

This study inspired a qualitative study on the barriers of novice programmer example use that looked at what novice programmers talked about during similar tasks to this study [53]. In the qualitative study, participants worked in pairs and only saw examples with the Visual Emphasis annotation. Findings included hurdles to success like being distracted by other exciting aspects of the programming environment, not understanding the example, not knowing where to find components in the interface, and trouble implementing solutions due to misunderstandings about the code.

Three hurdles from the qualitative study seem to align well with predictions in the decision trees: the context distraction hurdle, the example comprehension hurdle, and the programming environment hurdle.

- In the context distraction hurdle, participants talked about exploring the programming interface and trying out ideas they had to solve the task based on what they saw in the programming environment (rather than the example). This aligns with the decision tree in which having edits in the *Exploring & Searching Stage* often leads to failure, shown in Figure 6.
- The example comprehension hurdle captured participants talking about not understanding how an example worked or how it was relevant to a task. Spending long amounts of time with the example (as in the *Assembling Stage* decision tree in Figure 7) may indicate that a participant was trying to understand an example but struggling.
- In the programming environment hurdle, participants discussed having trouble finding a block they needed to solve a task, or having trouble editing their code in some manner. This aligns with the fact that a large number of UI actions in the *Ready-to-program Stage* likely means that a participant will not reach the *Assembling Stage*, as shown in the first prediction branch in Figure 7.

Novice programmers' descriptions of their difficulties on similar tasks aligning with the behavioral predictions in this analysis supports the value of using decision trees and behavioral data to explore example use and predict success.

6.4. Examples and Blocks-Based Programming

While examples are readily available for programmers in text languages, there are many fewer resources for programmers in blocks-based programming environments to find examples. Part of this may be due to the fact that it is not as easy to quickly copy and paste code to a forum like it is for text languages. However, the popularity of using example code in programming implies that it is important for the research community to address example use in blocks-based programming languages. This research topic is further compounded by the fact that most blocks-based programming language users are novices, so using examples is not as straightforward as it would be for experts. This study contributes a better understanding of novice blocks programmers using examples with varying annotations and similarities. This study also begins to explore two ways to predict success or failure of novice programmers using examples in blocks-based programming environments.

For the analogical mapping task, participants drew lines between components that were related in the example and the task. While there were several challenges in the implementation of this task, the challenges for a text-based language would be different. Instead of having discrete components, or blocks, that can be connected with lines, the participants would have to somehow decide which part of the code they wanted to connect and mark that in an understandable way. For novices, this might be more challenging because blocks of code are not necessarily as obvious in a text language. Furthermore, it would be much more difficult to group correct and incorrect mappings because participants would be able to draw lines between many more combinations of characters, whereas in a blocks-based programming environment, there are more constraints.

Our task stages and decision tree predictive modeling were also highly blocks-specific. For example, some of the stages were based on having the UI in the correct configuration to access the blocks needed to succeed in the task. This works across almost all blocks-based programming environments, where blocks are organized in palettes that programmers must correctly select to find the component that they need. In a text language, the intermediate steps would be different and likely more difficult to observe. Furthermore, code edits are easier to define in a blocks-based programming environment because it is easy to track when a code block has been added to a program, modified in a discrete way, or removed. In a text programming language, it would be more challenging to define when a code change has started and ended. While heuristics could be created for this purpose, the strategy used in this paper suits a blocks-based programming language more easily and effectively.

6.5. Limitations

In this study, we focused exclusively on the behavior of young novices as they attempt to use examples. Although young novice programmers may share some challenges with older novices, there are likely unique features about their approach to using examples. While we suspect that analogical reasoning has similar behavior in older children and adults, we do not know which aspects of the programming behavior we observed would apply to adult novices.

Additionally, participants were unfamiliar with both the target program and the example. Some prior work suggests that adult novices often attempt to integrate several snippets of found code to solve a problem [13]. This results in a situation similar to that of our study: novice programmers have code to modify that they do not fully understand and an example they want to apply to it. We think this is an important type of example use, but we acknowledge that it does not capture

all example use. The combination of a familiar, understood target program and an unfamiliar example is important and not addressed through this study.

7. Conclusions and Future Work

Overall, our results support other findings that completing programming tasks using examples is challenging for novice programmers. While similar examples and annotations help novice programmers perform better, it is clear that these are not enough support for novice programmers using examples. However, it is interesting that the three annotation conditions were not significantly different, indicating that the simple visual emphasis may be enough help even for novices to draw attention to the part of the example related to their task. This could be highly valuable if a system needed to automatically annotate examples, as visual emphasis is the only of the three where automation would be a viable option.

In terms of predicting success, analogical mapping seems like a promising method, but needs some improvement. Although we did not want to interrupt task flow to assess analogical mappings, future educational systems may benefit from integrating analogical mapping tasks into learning materials, which would allow them to measure analogical mappings during tasks. Furthermore, if future work could reduce the number of errors in mapping due to the way the mapping task was operationalized, there might be a higher correlation between mappings and success. Specifically, adding more constraints about which components can be mapped and how to map blocks in an example to missing blocks in a task are two important future directions for this evaluation method. We believe better constraints and a stricter time limit could vastly improve the consistency and accuracy of this method.

The stage-based analysis and decision tree models provide information about which programming behaviors impact success at each stage of the problem solving process. We found that in most tasks that do not succeed, participants did not progress past the *Exploring & Searching Stage*. This has important implications for the beginning of a task, when participants seem to need the most help. The large number of UI actions leading to failure in later stages gives some support to the idea of adding programming environment assistance to examples if programmers do not have the option to directly insert the example code into their program. These stages are highly applicable to other blocks-based programming environments where programmers normally must follow the same process to find the components they need in the interface, add them to their program, and then modify the program to complete the task. One promising future direction for this analysis of programming behaviors is in designing educational systems, such as intelligent tutoring systems, where a system could assess programmer behavior in real-time and use that to provide feedback to the learner.

References

[1] M. Resnick, J. Maloney, A. Monroy-Hernndez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[2] "MIT App Inventor | Explore MIT App Inventor." [Online]. http://appinventor.mit.edu/explore/

[3] "Looking Glass Community." [Online]. https://lookingglass.wustl.edu/

[4] M. Guzdial, "Why the U.S. is not ready for mandatory CS education," *Communications of the ACM*, vol. 57, no. 8, pp. 8–9, Aug. 2014.

[5] K. J. Harms, D. Cosgrove, S. Gray, and C. Kelleher, "Automatically generating tutorials to enable middle school children to learn programming independently," in *Proceedings of the 12th International Conference on Interaction Design and Children*. ACM, 2013, pp. 11–19.

[6] "Anybody can learn | Code.org." [Online]. http://code.org/

[7] M. J. Lee and A. J. Ko, "Personifying programming tool feedback improves novice programmers' learning," in *Proceedings of the Seventh International Workshop on Computing Education Research*. ACM, 2011, pp. 109–116.

[8] C. J. Butz, S. Hua, and R. B. Maguire, "A web-based Bayesian intelligent tutoring system for computer programming," *Web Intelligence and Agent Systems: An International Journal*, vol. 4, no. 1, pp. 77–97, 2006.

[9] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2009, pp. 1589–1598.

[10] J. Brandt, P. J. Guo, J. Lewenstein, S. R. Klemmer, and M. Dontcheva, "Writing code to prototype, ideate, and discover," *IEEE Software*, vol. 26, no. 5, pp. 18–24, 2009.

[11] B. Dorn and M. Guzdial, "Graphic designers who program as informal computer science learners," in *Proceedings 2nd International Workshop on Computing Education Research*, 2006, pp. 127–134.

[12] M. B. Rosson, J. Ballin, and J. Rode, "Who, what, and how: A survey of informal and professional web developers," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005, pp. 199–206.

[13] M. B. Rosson, J. Ballin, and H. Nash, "Everyday Programming: Challenges and Opportunities for Informal Web Development," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2004, pp. 123–130.

[14] P. Gross and C. Kelleher, "Non-programmers identifying functionality in unfamiliar code: strategies and barriers," *Journal of Visual Languages & Computing*, vol. 21, no. 5, pp. 263–276, 2010.

[15] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: integrating web search into the development environment," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 513–522.

[16] K. S.-P. Chang and B. A. Myers, "WebCrystal: understanding and reusing examples in web authoring," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2012, pp. 3205–3214.

[17] M. L. Gick and K. J. Holyoak, "Analogical problem solving," *Cognitive Psychology*, vol. 12, no. 3, pp. 306–355, 1980.

[18] D. Gentner, "Structure-mapping: A theoretical framework for analogy," *Cognitive Science*, vol. 7, no. 2, pp. 155–170, 1983.

[19] "Hoogle." [Online]. https://www.haskell.org/hoogle/

[20] "Java Examples - JExamples.com." [Online]. http://www.jexamples.com/

[21] "Google Code." [Online]. https://code.google.com/

[22] "Open Hub Code Search." [Online]. http://code.openhub.net/

[23] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding API components and examples," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2006, pp. 195–202.

[24] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings of the 27th International Conference on Software Engineering*. ACM, 2005, pp. 117–125.

[25] O. Hummel, W. Janjic, and C. Atkinson, "Code conjurer: Pulling reusable software out of thin air," *IEEE Software*, vol. 25, no. 5, pp. 45–52, 2008.

[26] D. Mandelin, L. Xu, R. Bodk, and D. Kimelman, "Jungloid mining: Helping to navigate the API jungle," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 48–61.

[27] N. Sahavechaphan and K. Claypool, "Xsnippet: Mining for sample code," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 413–430.

[28] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2007, pp. 204–213.

[29] Y. Ye and G. Fischer, "Supporting reuse by delivering task-relevant and personalized information," in *Proceedings of the 24th International Conference on Software Engineering*. ACM, 2002, pp. 513–523.

[30] P. Brusilovsky, "WebEx: Learning from Examples in a Programming Course." in *WebNet*, vol. 1, 2001, pp. 124–129.

[31] R. Burow and G. Weber, "Example explanation in learning environments," in *Intelligent Tutoring Systems*. Springer, 1996, pp. 457–465.

[32] J. M. Faries and B. J. Reiser, "Access and use of previous solutions in a problem solving situation," Cognitive Science Laboratory, Princeton University, Tech. Rep. CSL Report 29, Jun. 1988.

[33] M. Guzdial and C. Kehoe, "Apprenticeship-based learning environments: A principled approach to providing software-realized scaffolding through hypermedia," *Journal of Interactive Learning Research*, vol. 9, pp. 289–336, 1998.

[34] L. Hohmann, M. Guzdial, and E. Soloway, "SODA: A computer-aided design environment for the doing and learning of software design," in *Computer Assisted Learning*. Springer, 1992, pp. 307–319.

[35] M. C. Linn, "How can hypermedia tools help teach programming?" *Learning and Instruction*, vol. 2, no. 2, pp. 119–139, 1992.

[36] M. C. Linn and M. J. Clancy, "Can experts' explanations help students develop program design skills?" *International Journal of Man-Machine Studies*, vol. 36, no. 4, pp. 511–551, 1992.

[37] H. Ramadhan, "An intelligent discovery programming system," in *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: Technological Challenges of the 1990's*. ACM, 1992, pp. 149–159.

[38] A. S. Bruckman, "MOOSE Crossing: Construction, community, and learning in a networked virtual world for kids," Ph.D. dissertation, Massachusetts Institute of Technology, 1997.

[39] K. J. Harms, J. H. Kerr, M. Ichinco, M. Santolucito, A. Chuck, T. Koscik, M. Chou, and C. L. Kelleher, "Designing a community to support long-term interest in programming for middle school children," in *Proceedings of the 11th International Conference on Interaction Design and Children*, ser. IDC '12. New York, NY, USA: ACM, 2012, pp. 304–307.

[40] J. Cao, I. Kwan, R. White, S. D. Fleming, M. Burnett, and C. Scaffidi, "From barriers to learning in the idea garden: An empirical study," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2012, pp. 59–66.

[41] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: suggesting solutions to error messages," in *Proceedings 28th International Conference on Human Factors in Computing Systems*, 2010, pp. 1019–1028.

[42] S. Oney and J. Brandt, "Codelets: linking interactive documentation and example code in the editor," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2012, pp. 2697–2706.

[43] L. R. Neal, "A system for example-based programming," in *ACM SIGCHI Bulletin*, vol. 20. ACM, 1989, pp. 63–68.

[44] D. F. Redmiles, "Reducing the variability of programmers' performance through explained examples," in *Proceedings of the INTERACT'93 and CHI'93 Conference on Human Factors in Computing Systems*. ACM, 1993, pp. 67–73.

[45] G. Weber and A. Mollenberg, "ELM-PE: A Knowledge-based Programming Environment for Learning LISP." in *Proceedings of ED-MEDIA 94–World Conference on Educational Multimedia and Hypermedia*, 1994, pp. 557–562.

[46] K. Østerbye, "Minimalist documentation of frameworks," in *ECOOP Workshops*, 1999, pp. 172–173.

[47] T. Vestdam, "Generating consistent program tutorials," in *Proceedings of NWPER2002-Nordic Workshop on on Programming and Software Development Tools and Techniques*, 2002.

[48] J. M. Carroll, P. L. Smith-Kerker, J. R. Ford, and S. A. Mazur-Rimetz, "The minimal manual," *Human-Computer Interaction*, vol. 3, no. 2, pp. 123–153, 1987.

[49] J. M. Carroll, *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill*. MIT Press, 1990.

[50] J. L. Kolodner, M. T. Cox, and P. A. Gonzlez-Calero, "Case-based reasoning-inspired approaches to education," *The Knowledge Engineering Review*, vol. 20, no. 03, pp. 299–303, 2005.

[51] E. Domeshek and J. Kolodner, "Using the points of large cases," *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing*, vol. 7, no. 02, p. 87, May 1993.

[52] J. L. Kolodner and M. Guzdial, "Theory and practice of case-based learning aids," *Theoretical Foundations of Learning Environments*, pp. 215–242, 2000.

[53] M. Ichinco and C. Kelleher, "Exploring novice programmer example use," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2015, pp. 63–71.

[54] J. Sweller and G. A. Cooper, "The use of worked examples as a substitute for problem solving in learning algebra," *Cognition and Instruction*, vol. 2, no. 1, pp. 59–89, 1985.

[55] A. Rourke and J. Sweller, "The worked-example effect using ill-defined problems: Learning to recognise designers' styles," *Learning and Instruction*, vol. 19, no. 2, pp. 185–199, Apr. 2009.

[56] J. J. Van Merrinboer and M. B. De Croock, "Strategies for computer-based programming instruction: Program completion vs. program generation," *Journal of Educational Computing Research*, vol. 8, no. 3, pp. 365–394, 1992.

[57] B. B. Morrison, L. E. Margulieux, B. Ericson, and M. Guzdial, "Subgoals Help Students Solve Parsons Problems," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 2016, pp. 42–47.

[58] F. G. Paas and J. J. Van Merrinboer, "Variability of worked examples and transfer of geometrical problem-solving skills: A cognitive-load approach." *Journal of Educational Psychology*, vol. 86, no. 1, p. 122, 1994.

[59] J. Sweller, "Element interactivity and intrinsic, extraneous, and germane cognitive load," *Educational Psychology Review*, vol. 22, no. 2, pp. 123–138, 2010.

[60] "Scratch - Videos." [Online]. https://scratch.mit.edu/help/videos/

[61] "Tutorials for App Inventor | Explore MIT App Inventor." [Online]. http://appinventor.mit.edu/explore/ai2/tutorials.html

[62] S. Pongnumkul, M. Dontcheva, W. Li, J. Wang, L. Bourdev, S. Avidan, and M. F. Cohen, "Pause-and-play: automatically linking screencast video tutorials with applications," in *Proceedings of the 24th annual ACM Symposium on User Interface Software and Technology*. ACM, 2011, pp. 135–144.

[63] K. J. Harms, N. Rowlett, and C. Kelleher, "Enabling independent learning of programming concepts through programming completion puzzles," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2015, pp. 271–279.

[64] "Blockly Games." [Online]. https://blockly-games.appspot.com/

[65] M. J. Lee and A. J. Ko, "Comparing the effectiveness of online learning approaches on CS1 learning outcomes," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. ACM, 2015, pp. 237–246.

[66] "Kodu | Home." [Online]. http://www.kodugamelab.com/

[67] "TouchDevelop - create apps everywhere, on all your devices!" [Online]. https://www.touchdevelop.com/

[68] S. Surisetty, C. Law, and C. Scaffidi, "Behavior-based clustering of visual code," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2015, pp. 261–269.

[69] L. E. Richland, K. J. Holyoak, and J. W. Stigler, "Analogy use in eighth-grade mathematics classrooms," *Cognition and Instruction*, vol. 22, no. 1, pp. 37–60, 2004.

[70] D. Gentner and C. Toupin, "Systematicity and surface similarity in the development of analogy," *Cognitive Science*, vol. 10, no. 3, pp. 277–300, 1986.

[71] L. R. Novick and K. J. Holyoak, "Mathematical problem solving by analogy." *Journal of Experimental Psychology: Learning, Memory, and Cognition*, vol. 17, no. 3, p. 398, 1991.

[72] S. Hudson, J. Fogarty, C. Atkeson, D. Avrahami, J. Forlizzi, S. Kiesler, J. Lee, and J. Yang, "Predicting human interruptibility with sensors: a Wizard of Oz feasibility study," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2003, pp. 257–264.

[73] B. A. Spellman and K. J. Holyoak, "If Saddam is Hitler then who is George Bush? Analogical mapping between systems of social roles." *Journal of Personality and Social Psychology*, vol. 62, no. 6, p. 913, 1992.

[74] L. Beckwith, C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrance, A. Blackwell, and C. Cook, "Tinkering and gender in end-user programmers' debugging," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2006, pp. 231–240.

[75] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[76] L. Breiman and A. Cutler, "Random forests – classification description," 2007. [Online]. https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

[77] J. Cao, S. D. Fleming, and M. M. Burnett, "An exploration of design opportunities for "gardening" end-user programmers' ideas." in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2011, pp. 35–42.

[78] D. Loksa, A. J. Ko, W. Jernigan, A. Oleson, C. J. Mendez, and M. M. Burnett, "Programming, problem solving, and self-awareness: Effects of explicit guidance," *Proceedings International Conference on Human factors In Computing Systems*, 2016.

[79] M. Beveridge and E. Parkins, "Visual representation in analogical problem solving," *Memory & Cognition*, vol. 15, no. 3, pp. 230–237, 1987.

[80] K. J. Holyoak, E. N. Junn, and D. O. Billman, "Development of analogical problem-solving skill," *Child Development*, pp. 2042–2055, 1984.

Design Considerations to Increase Block-based Language Accessibility for Blind Programmers Via Blockly

Stephanie Ludi
Department of Computer Science & Engineering
University of North Texas
Denton, TX, USA
Stephanie.ludi@unt.edu

Mary Spencer
Department of Information Science and Technology
Rochester Institute of Technology
Rochester, NY, USA
mc3630@rit.edu

Abstract *Block-based programming languages are a popular means to introduce novices, specifically children, to programming and computational thinking concepts. They are tools to broaden participation in computing. At the same time, block-based languages and environments are an obstacle in broadening participation for many users with disabilities. In particular, block-based programming environments are not accessible to users who are visually impaired. This lack of access impacts students who are participating in computing outreach, in the classroom, or in informal settings that foster interest in computing. This paper will discuss accessibility design issues in block-based programming environments, with a focus on the programming workflow. Using Google's Blockly as a model, an accessible programming workflow is presented that works alongside the conventional mouse-driven workflow typical of block-based programming. The project presented is still in progress.*

Keywords *accessibility; block-based languages; visually impaired*

1. Introduction

Block-based systems have gained prominence in recent years as a means of introducing novices to programming. The Blockly framework was developed by Neil Fraser's team at Google [1]. Blockly (sometimes with modifications) is often used as a framework for other block-based languages and activities (such as App Inventor, Gameblox, Code.org Hour of Code activities, as well as a newly announced version of Scratch [2]).

In some cases, such as MIT's Scratch, online communities exist and the tools are integrated into pre-college computer science curricula (e.g. Exploring Computer Science, CSPrinciples) [3]. As these systems have become components of curricula, after-school camps, and outreach, the lack of accessibility for many students with disabilities creates an obstacle for participation in these activities that are devised to increase participation in computing. In particular, users with visual impairments are generally not able to use block-based tools unless they have enough vision to view the screen comfortably. So one motivation for our work is to make popular block programming environments and activities more accessible to the visually impaired.

The other motivation is that, as with novice programmers who are sighted, the learning curve of dealing with the syntax of text-based code persists with novice programmers who are visually impaired. For example, many pre-college students are not familiar with curly braces and their location on the keyboard. The benefit of focusing on programming concepts over syntax is relevant to for the visually impaired as well as for sighted novice programmers. By extension, the ability to create and modify programs may be easier with the structural features of blocks compared to text, if the tools are designed appropriately.

Visually impaired individuals may have some sight or have little to no vision. Assistive technology needs vary according to the degree of sight a person possesses. People who can read magnified text may use screen magnification software to view the contents of the screen by a specified magnification factor, adjust foreground and background colors, or increase the size of the cursor. These users typically use the mouse alongside the keyboard. Individuals who are blind do not use the mouse. Instead, navigation relies on the keyboard (often through keyboard shortcuts). Screen readers (and for some, refreshable Braille displays) are the means to access information that is displayed on the screen. When a website or program is designed correctly, the screen reader will read the content, including menus and other navigational elements. In the case of typical block-based programming environments, the screen reader reads no content or navigation elements.

Google has recently undertaken work on an accessible version of Blockly [4]. Their approach focuses on screen reader compatibility, which is the foundation for access for blind users. Google's Accessible Blockly demo re-imagines the UI, so that text is used over blocks, as shown in Figure 1.

Our approach to redesigning Blockly's user interface focuses on preserving the current drag-and-drop user interface for creating block-based programs, while adding additional features to increase access to visually impaired users (or others who cannot use a mouse). Our approach includes the addition of a keyboard interface, screen reader compatibility, appearance customization, and related features to increase access to Blockly-based systems. Once complete, the authors envision that our additions could be applied to any tool that uses the block-based Blockly framework.

This paper also presents accessibility issues with the design of block-based languages. Users with visual impairments, including blindness, are the focus of this paper. Our redesign of the Blockly framework, specifically features to facilitate the

DOI reference number: 10.18293/VLSS2017-013

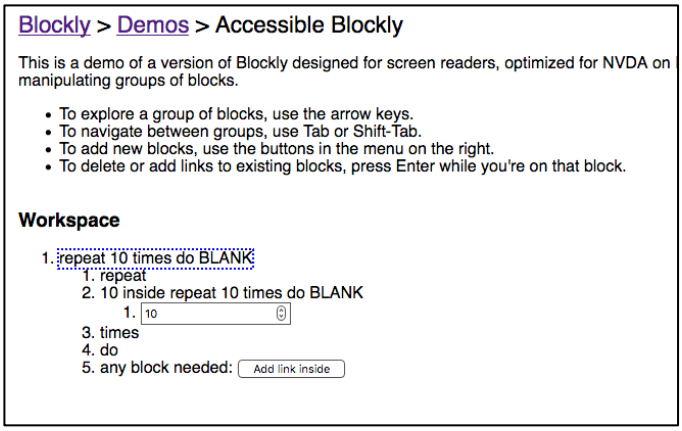


Figure 1. Screenshot of Accessible Blockly demo depicting the **`repeat`** block as text.

programming workflow and access to the block-based programs created in Blockly, are discussed. Our work is ongoing and has not had formal evaluation with users. However, the work seeks to make block-based programming accessible to the blind and other users with visual impairments, and can help other developers. Furthermore, the goal is to add the functionality on top of the Blockly framework via files that can be added to any Blockly-based project, thus making accessibility a seamless option for developers.

2. Research Questions

Using the Blockly platform, we are re-engineering the system to enable use by users who are visually impaired. As such, our preliminary research questions are:

- How can Blockly be made accessible to the visually impaired, while at the same time remain usable to sighted users?
- What features will both visually impaired and sighted users appreciate?

3. Block-Based Programming Environment Features that affect Accessibility

Each block-based project team makes design decisions for their project that in some cases have unforseen consequences. Like any software project, our team prioritizes features based on a variety of needs that are considered. Examples of these design decisions are described in the following subsections.

3.1. Technology and Platform Choices

The technologies used to develop block-based systems can have a large impact on accessibility of said systems to the disabled. The impact can be significantly positive or negative.

Scratch 2.0 uses Actionscript/Flash. A positive implication is that Scratch runs in the browser, making installation seamless and thus easy to access for many people using various operating systems. At the same time, the technology selection makes accessiblity impossible. In 2010, Adobe announced accessibility suport for Flash/Flex in terms of the Accessible Rich Internet Applications (ARIA) specification [5]. While ARIA is supported in terms of roles and states for HTML, Flash objects and Actionscript do not support ARIA roles and states, so presenting and interacting with a system is not possible (additional details are provided in Section 4.1). Another issue for screen reader users, who rely on keyboard shortcuts, is that Flash can override those keyboard shortcuts.

The Lego Mindstorm block-based robotics-programming environment is traditionally a desktop application that uses LabView as the underlying technology. While the software runs on the Windows and Mac operating systems, the software is not compatible with screen readers. As a result, users who are blind will not hear anything.

On the positive side, Blockly is developed in CSS, SVG, and JavaScript. As such, Blockly also runs on the web browser. However, Blockly does not have the same accesibility issues, because it can leverage ARIA specification from the W3C's Web Accessibility Initiative. Developers need to adhere to the ARIA specification since compatibility does not happen automatically for any web application. By following ARIA, a screen reader can read the structure of the web page, the labels on buttons and menus, as well as the graphical objects (e.g. blocks). Widgets, such as sliders and tree items, can be described. Support also includes keyboard navigation, as well as clearly articulated properties for drag-and-drop, widget states, or areas of a page that can be updated over time or based on an event. [6]

3.2. Mouse-centric Input

Many block-based systems rely on mouse input as the primary means of accessing features, including selecting blocks and adding them to programs, selecting attributes, and running the created program. One cannot use the keyboard to locate, select, and place a block onto the workspace in Scratch or Blockly as they were originally designed to rely on the mouse for such interaction.

Many users with visual impairments rely on the keyboard as the primary input device. As such, keyboard-focused commands and shortcuts are key to making interaction possible. Systems must be designed in order to utilize the keyboard as input in terms of menu navigation, programming, and accessing various panes in the programmng environment (e.g. changing focus to access specific information). In addition, the keys used to access features and information needs to be consistent with said standards and not conflict with keyboard shortcuts used by screen readers. In order to provide appropriate access to both sighted and visually impaired students, designing the system to allow for interaction via the mouse or keyboard is needed. This aspect is the foundation for much of the work in our project.

Wagner and Gray [7] discuss the use of a Vocal User Interface for Scratch. While a voice-based user interface can be used by users who are blind, their work focused on users with physical disabilities where using the mouse is not feasible or comfortable for long periods of time. The use of voice is another option for some users, though the audio presentation of

DOI reference number: 10.18293/VLSS2017-006

the system and blocks are the focus for this paper. Nonetheless, their work shows that there is not a simple one size fits all for accessibility.

3.3. Feedback

User feedback in block-based programming environments are often visual in nature (e.g. a visual change on the workspace, pop-up messages, dialog boxes) without an audio feedback mechanism. Examples include a successful compilation or incompatible blocks that the user tries to connect together. Providing associated audio-based feedback can take various approaches depending on the nature of the feedback.

Students who use screen readers need to have all content, including errors and any status messages, provided audibly. The audio capabilities of many block-based systems are limited, whether it be the ability for a robot to play a tone or recorded sound or for a Microsoft Kodu game to play a sound effect when an event occurs. The audio capabilities, including the ability for a form of audio description when an animation is played or dialog are displayed, is needed. Tapping into the location attributes or dialog text to enable compatibility with a screen reader is possible in many technologies (e.g., JavaScript, Java, C++, C#).

Audio-based feedback can be in the form of speech or sound. As a pane or dialog box gets focus, the error or status text can be read (assuming it is programmed to enable a screen reader to access the text). Other feedback may be in the form of sounds (e.g. an audio icon or earcon) that correspond to a specified meaning. An example of an audio icon is the sound of crumpling paper when a file is moved to the trashcan [8]. An example of an earcon is a tone or chord that is abstract in terms of the sound itself, but it is given meaning according to the association, such as a deep sound may correspond to an unsuccessful download of the program to a robot [8]. We are leveraging related work that has been conducted earlier to assess the use of audio cues in programming for programmers, though the study was conducted in a traditional, text-based programming environment [9]. Our work uses a working prototype of Blockly as the environment, enabling the developers to get early user feedback that helped direct the work described here.

3.4. Block-based Programs Created by the User

Each block-based program is designed for a particular programming environment. Scratch programs can be in the form of animations or games. Lego Mindstorms NXT-G programs allow a robot to move and interact with its environment. Programming environments based on the Blockly framework enable users to create programs in a variety of languages. Some Blockly-based tools produce JavaScript or Dart, where other tools are used to program robots.

4. Redesigning Blockly for Accessibility

Our work in modifying Blockly to provide access to users who are visually impaired is ongoing. The following sections provide an overview of system modifications. Our team uses only the technologies that Blockly uses, with one exception: the addition of a single JavaScript library to provide specific audio feedback.

The overarching work focuses on program creation and program navigation. The key goals of program preparation are:

- Allowing the user to change the highlight color for the current block or the connection point of a block in the workspace to improve discernability. This is in contrast to the findings in Fraser [10], where a borderless look prevails.
- Allowing the user to change the color of the workspace in order to minimize visual discomfort and improve readability.
- Enabling the blocks to be read on the workspace and in the toolbox (the menu where blocks are chosen).
- Visually linking blocks with any associated comments, where the user can jump from the block to the comment and back as desired.
- Providing a unique identifier with each block to enable visual and audio-based understanding of blocks.

While program creation is critical, one tends to program incrementally in terms of adding features or fixing defects. As such, the need to enable visually impaired users to be able to navigate their code is critical. Our team designed the following features to facilitate the navigation of code:

- Each block as well as each block part (e.g. mutator or inner block) can be read as a single block or in the program as a whole, depending on user need, as described in Section 4.1.
- The keyboard can be used to navigate between blocks and within a block (e.g. mutators), as described in Section 4.2.
- Each block (including vertical blocks) is given a unique identifier to provide a visual and auditory structure for the program. The identifier is presented in the tree view, as described Section 4.2.
- Audio cues are used in order to reinforce the level of nesting. A comparative study is underway.

The following subsections outline our efforts to increase access to Blockly in these areas.

4.1. Block Content Presentation by the Screen Reader

Blockly had no screen reader support initially. Our team added screen reader support. The text read by the screen reader is generated by our modified Blockly function that converts a

given block into a string. The original Blockly function returns a string containing both the given block and all child blocks combined. This meant entire blocks of code were read in one shot, providing too much information to the user. The function was altered to return only the selected block and any child blocks that would translate into a single line of code, for instance, the multiple blocks needed to create an **`if`** statement. Figure 2 shows a sample program that contains an **`if`** statement.

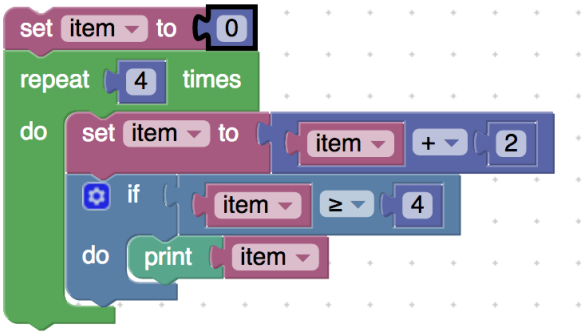


Figure 2. A sample program created in Blockly.

In the sample program in Figure 2, there is an **`if`** block inside of a **`repeat`** block. The screen reader reads the **`if`** statement as *if item greater than or equal to 4.*

The original Blockly function also replaced empty connections with question marks (e.g., create a list with items ?, ?, ?). As symbols, question marks do not translate well verbally nor do they indicate the location of the connection in relation to other connections. To resolve this, we replaced the question marks with the letters 'A' through 'Z' in order. This allows for 26 empty connections per block and the default blocks require 4 at most.

Referring back to the **`if`** block from Figure 2, the removal of *item* and *4* would result in the screen reader portraying the **`if`** block as *if block A is greater than or equal to B*. This is an improvement to the screen reader saying *if ? is greater than or equal to ?*, as stating *question mark* multiple times in quick succession may be jarring to the listener. Further testing will assess how best to provide information without overloading short-term memory.

Our team added screen reader support by dynamically adding Web Accessibility Initiative – Accessible Rich Internet Application (WAI-ARIA) attributes to the page. WAI-ARIA is the World Wide Web Consortium (W3C) standard for increased accessibility [6].

Three particular WAI-ARIA attributes are used to make Blockly accessible: aria-live, aria-label and role. These attributes are applied to a hidden div on the page that is updated with the necessary string to be read aloud. In order to have cross-browser support, the attributes are added both statically when the program loads (Chrome, Firefox, Edge/Internet Explorer) and dynamically each time the div is updated (Safari). The dynamic update each time the div is updated in Safari is a workaround at the time, but as web browsers evolve the approaches may become more streamlined.

The aria-live region attribute is placed on an element to inform the screen reader when the content of that element is updated. This region has 2 potential values: polite and assertive. A live region with a value of polite will only read the updated content after the current screen reader buffer is emptied. This ensures that the user does not lose their place mid-page. A live region with a value of assertive will interrupt the current screen reader buffer to immediately read the updated information. For Blockly, the assertive value is used so that the screen reader can immediately respond to user input when a block is selected or placed on the workspace.

While the aria-live region technique reads the content of the div on most browsers, Safari requires the div to also have an aria-label attribute. An aria-label defines what is read when an element is selected. Examples include reading the content of a block that is selected when browsing in the block menu or reading the name of each block category in the block menu (e.g. Logic, Math). Usually a div element would not need an aria-label. However, Safari only responds when the aria-label changes in the live region, not when the inner text of the div changes.

4.2. Keyboard Support for Block Menu and Block Navigation

Blockly is designed to use a mouse as the primary form of input. The only parts of the interface that are keyboard accessible are the non-Blockly-specific ones that were created using Google Closure (i.e., the outer toolbox menu). Users who are blind use only the keyboard to navigate the web, meaning all mouse events in Blockly need to be replaced by our team. This includes selecting, adding, connecting, and editing blocks as well as interacting with the mutator and context menus.

Screen readers usually have their own hotkeys for web navigation. The hotkeys are also not necessarily consistent across screen readers. This is a problem for applications that require the use of keyboard shortcuts. To mitigate conflicts, we selected a set of keys to facilitate navigation. Selecting and navigating between blocks in Blockly, for instance, are mapped to the W (up), A (left/back), S (right/next) and D (down) keys because the arrow keys are already used navigate through the HTML content of the page. These keys were chosen because they are commonly used for directional navigation in games. The WASD keyboard convention also allows the user to use their left hand for navigation and their right hand for selecting items with the enter key.

In addition to navigating among blocks and menus, it is also necessary to add comments to blocks and go back and forth between comments and blocks. As part of our redesign of the user interface, the user has a box on the side of their screen to view comments in a tree view (Microsoft Excel does this as well). The screen reader reads the comments as desired. The tree view is updated automatically, and the structure will match that of the program using the hierarchy of identifiers associated

with each block. In addition, a line will connect the current block with the comment line in the tree view.

The hierarchy was created using an n-ary search tree that cycles through a block and all of its children. Each block is given an alphanumeric label to help give the user a sense of location while navigating. The outermost blocks are lettered (A), while each inner block is given a number (A1), and each additional nested level is given a decimal (A1.1). In Figure 2, the outer **`set`** block is A, the **`repeat`** block is B, and the two blocks nested with the **`repeat`** block are the inner **`set`** block (B1) and the **`if`** block (B2). By extension, the **`add`** block that is connected of the **`repeat`** block is noted as B1.1.

4.3. Connecting and Editing Blocks on the Workspace

In order to use the keyboard to add blocks to the workspace, an edit mode was provided. This mode is needed to avoid conflicts with other Blockly features. If a user presses the E key when a block is selected, the WASD navigation keys transition from selecting different blocks to selecting any of the connections or fields on an individual block. Each time the user switches connections, the screen reader announces the new connection. Pressing the Enter key on one of these connections or fields allows the user to attach another block (setting an insertion point) or edit the field. If a block is added to the workspace unconnected, it will automatically be attached to the last outermost block. If a block cannot be attached to another block, then it will be unconnected and need to be moved or otherwise managed (e.g. deleted if undesired). As noted previously, users insert and move blocks by setting an insertion point if the desired location is different from the current location. The authors are currently assessing how to best accomplish this critical feature, including how to clearly articulate the location of the insertion point.

In Blockly, some blocks have mutator menus that allow the user to drag additional inputs or statements onto a block. For example, an **`if`** block can turn into an **`else if`** block. The mutator menu is a pop-up dialogue where users can drag and drop blocks onto the existing block to alter it. This mouse-based menu was revised in two ways by our team. Some mutators were turned into individual blocks such as the **`else if`** and **`else`** blocks. Other mutators, namely the ones that changed the number of outputs on a block, were given a drop-down menu that would dynamically add and remove outputs as necessary.

4.4. Additional Features to Enhance Access

Additional changes were made to enhance the overall experience for visually impaired users, which may also appeal to sighted users. These changes include adding the ability to change the text size and color of the workspace, an accessible custom help guide, and a comment display window.

Themes

Three themes were added to the workspace: high contrast, off-white, and matte blue. These colors were added after a participant in an early study commented on the eyestrain caused by the original bright white workspace. All of the colors were tested with a color blindness simulator and a member of the team who is color blind to ensure that the blocks were distinguishable from each other and the background.

Accessible Help

An accessible custom help site was created with references for each block. When a block is selected, a hotkey can be used to open that particular block's help information on the site. The default help pages for the Blockly library led to a page that is not fully accessible and provided limited information. The new site was specifically designed by our team to navigate with a screen reader and has detailed information on each block.

5. Next Steps

The initial version of accessible Blockly should be completed during Winter 2017. The results of a prior audio feedback study provided early feedback on the user interface, as well as assessed the impact of various types of audio feedback modalities for code navigation and the understanding of nesting. The feedback will be used in implementing code-based audio feedback during code navigation, in conjunction with the option for screen reader use, if needed. The accessibility features will also be studied in order to compare the usability impact for users with and without sight in order to ascertain what value may be found for sighted users as well as those who are visually impaired. Concerns include the verbosity of the information being presented, as well as the issues that young users may have as they are often users of Block-based systems. The formal studies will allow the team to refine the system and provide greater access for users, while providing a model for developers of other Blockly-based systems or block-based systems in general.

In addition to block-based languages, hybrid text and block-based languages such as Pencil Code [11] also have the potential for increased accessibility. Our team is also looking at applying our strategies to Pencil Code, but the teams behind Pencil Code, Code.org's App Lab, and other block languages can integrate accessibility into their system's designs by following the ideas we have discussed here. Some changes are at the user interface level while other accommodations are deeper in terms of new or redesigned features. An example of this is the Stride frame-based editor used in tools such as Greenfoot [12]. This editor enables users to work with operations and constructs at an abstract level. This innovation, and others like it, may help increase access to computer science for students with disabilities, as well as students overall.

Acknowledgments

Thanks to all the hard work on the RIT Blockly team, as well as the participants who gave initial feedback. This project is supported by the National Science Foundation (CNS-1240856, CNS-1240809).

References

[1] Blockly Development Website. [Online]. https://developers.google.com/blockly

[2] J. Goode. "Exploring computer science: An equity-based reform program for 21st century computing education," *Journal for Computing Teachers*. Summer, 2011.

[3] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, (2010). "The Scratch programming language and environment." *ACM Transactions on Computing Education*, vol. 10, no. 4, pp. 16:1–16:15, Nov. 2010.

[4] Demo of Accessible Blockly from Google. [Online]. https://blockly-demo.appspot.com/static/demos/accessible/index.html

[5] A. Kirkpatrick. (2010) Adobe Accessibility / Flash Player and Flex Support for IAccessible2 and WAI-ARIA. [Online]. http://blogs.adobe.com/accessibility/2010/03/flash_player_and_flex_support.html

[6] W3C-WAI (2014). WAI-ARIA Overview. [Online]. http://www.w3.org/WAI/intro/aria

[7] A. Wagner and J. Gray. "An empirical evaluation of a vocal user interface for programming by voice," *International Journal of Information Technologies and System. Approach,* vol. 8, no. 2, pp. 47-63, Jul. 2015.

[8] T. Dingler, J. Lindsay, and B. N. Walker, "Learnability of sound cues for environmental features: Auditory icons, earcons, spearcons, and speech," *Proceedings of the International Conference on Auditory Display (ICAD 2008)*, Paris, France, 24-27, Jun. 2008.

[9] A. Stefik, C. Hundhausen, and R. Patterson. "An empirical investigation into the design of auditory cues to enhance computer program comprehension," *The International Journal of Human-Computer Studies*, vol. 69, no. 12, pp. 820-838, 2011.

[10] N. Fraser, "Ten things we've learned from Blockly", *IEEE Blocks and Beyond Workshop*, 2015, pp. 49-50.

[11] D. Bau, D. A. Bau, M. Dawson, and C. S. Pickens, "Pencil code: Block code for a text world," in *Proceedings of the 14th International Conference on Interaction Design and Children (IDC '15),* ACM, 2015, pp. 445–448.

[12] T. W. Price, N, C.C. Brown, D. Lipovac, T. Barnes, and M. Kölling, "Evaluation of a frame-based programming editor." In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16),* ACM, 2016, pp. 33-42.

Reviewers

Ameer Armaly	University of Notre Dame
Michal Armoni	Weizmann Institute of Science
Dimitar Asenov	ETH Zurich
David Bau	MIT
Sayamindu Dasgupta	University of Washington
Brian Dorn	University of Nebraska at Omaha
Michael Eisenberg	University of Colorado
Scott Fleming	University of Memphis
Mark Friedman	Google
Mark Gross	University of Colorado
Philip Guo	UC San Diego
Brian Harvey	University of California, Berkeley
Andrew Ko	University of Washington
Clayton Lewis	University of Colorado
John Maloney	CDG Labs
Fred Martin	University of Massachusetts Lowell
Yoshiaki Matsuzawa	Aoyama Gakuin University
Jens Mönig	Y Combinator Research (HARC)
Yoshiki Ohshima	Y Combinator Research (HARC)
Marian Petre	Open University
Eric Rosenbaum	Google
Krishnendu Roy	Valdosta State University
Natalie Rusk	MIT Media Lab
Ben Shapiro	University of Colorado, Boulder
Steve Tanimoto	University of Washington
Rob Thompson	University of Washington

Author Index

Keyword Index