Sussex Research Online

Programming language, natural language? Supporting the diverse computational activities of novice programmers

Article (Accepted Version)

Good, Judith and Howland, Kate (2017) Programming language, natural language? Supporting the diverse computational activities of novice programmers. Journal of Visual Languages and Programming, 39. pp. 78-92. ISSN 1045-926X

This version is available from Sussex Research Online: http://sro.sussex.ac.uk/id/eprint/65897/

This document is made available in accordance with publisher policies and may differ from the published version or from the version of record. If you wish to cite this item you are advised to consult the publisher's version. Please see the URL above for details on accessing the published version.

Copyright and reuse:
Sussex Research Online is a digital repository of the research output of the University.

Copyright and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable, the material made available in SRO has been checked for eligibility before being made available.

Copies of full text items generally can be reproduced, displayed or performed and given to third parties in any format or medium for personal research or study, educational, or not-for-profit purposes without prior permission or charge, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

http://sro.sussex.ac.uk

Programming Language, Natural Language? Supporting the Diverse Computational Activities of Novice Programmers

Judith Good, Kate Howland

Department of Informatics, University of Sussex, United Kingdom

Abstract

Given the current focus on teaching computational concepts to all from an early age, combined with the growing trend to empower end users to become producers of technology rather than mere consumers, we consider the issue of "computational notation". Specifically, where the goal is to help individuals develop their understanding of computation and/or use computation in real world settings, we question whether natural language might be a preferred notation to traditional programming languages, given its familiarity and ubiquity. We describe three empirical studies investigating the use of natural language for computation in which we found that although natural language provides support for understanding computational concepts, it introduces additional difficulties when used for coding. We distilled our findings into a set of design guidelines for novice programming environments which consider the ways in which different notations, including natural language, can best support the various activities that comprise programming. These guidelines were embodied in Flip, a bi-modal programming language used in conjunction with the Electron toolset, which allows young people to create their own commercial quality, narrative based role playing games. Two empirical studies on the use of Flip in three different real world contexts considered the extent to which the design guidelines support ease of use and an understanding of computation. The guidelines have potential to be of use both in analysing the use of natural language in existing novice programming environments, and in the design of new ones.

Email address: `J.Good@sussex.ac.uk` (Judith Good, Kate Howland)

Keywords:
novice programming languages, natural language, design, empirical evaluation.

1. Introduction

1.1. Computation for All

Ten years ago, Wing (2006) published a paper promoting the term "computational thinking", defined as "solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science" (p. 33). Wing noted that such a skill is not synonymous with knowing how to program, rather it involves the ability to think "at multiple levels of abstraction" (p. 35). Dijkstra proposed a similar idea 30 years earlier, noting that "it becomes clear what 'programming' really amounts to, viz. designing algorithmic solutions, and that activity requires the ability to think efficiently more than anything else." (Dijkstra, 1976, p. 2).

Wing's original article spawned numerous papers attempting to define the concept (Hu, 2011; Selby and Woollard, 2014), and consider the best ways to teach it (Guzdial, 2008; Hambrusch et al., 2009; Yadav et al., 2011), as well as assess it (Brennan and Resnick, 2012; Moreno-León and Robles, 2015; Zhong et al., 2015). Although computational thinking's relationship to computing, and to programming in particular, is not always clear, it has gained considerable traction at school level, with countries such as England making computing a mandatory part of the curriculum (Department for Education, 2013). More generally, there has been intense focus on developing children's computational thinking skills (Grover and Pea, 2013; Wohl et al., 2015), with numerous countries introducing computational thinking initiatives, whether part of a mandatory school curriculum or not (see the recent special issue of *ACM Transactions in Computing Education* for a number of country specific case studies (Tenenberg and McCartney, 2014)).

In parallel with this movement in education, computation has become integral to almost every discipline (Bundy, 2007), and computational techniques and methods are enriching both theory and research in multiple domains. It follows that since domain experts are best placed to understand the needs of their sector, they should have direct access to the computational power that can help them achieve their aims. Outside of the professional arena, there has been an intense surge of interest in "maker movements" and, more broadly, in the idea that the public should no longer content themselves with being mere consumers of the digital devices that increasingly permeate entertainment and leisure activities, but should also have the means to be producers, or at least "customisers" of these technologies.

Given this focus on putting computational power into the hands of the greatest number of users, and making computational understanding a goal that is within reach of everyone, many have questioned whether traditional programming languages are the best medium through which to achieve this and, in the case of teaching computational thinking, whether programming languages or even computers are actually required (see e.g. (Cortina, 2007; Bell et al., 2009; Lu and Fletcher, 2009)).

Even if it is possible to sidestep the debate of whether programming languages are necessary for the acquisition of computational thinking skills, it is clear that some form of notation is necessary for people to communicate with digital devices and, in an educational context, demonstrate their understanding of computation. In this article, we consider whether natural language might prove to be a suitable alternative to traditional programming languages. It is well established that the unnatural and complex syntax which is characteristic of many traditional programming languages is a major stumbling block for novices (Kelleher and Pausch, 2005) and could effectively act as a barrier to the use of computational techniques and understanding of computation more generally. Therefore, one obvious question to ask is, why can't people simply program using natural language? Given that both children and adults are already adept at using natural language to express ideas and concepts, it could provide a simple solution to syntax problems, while eliminating the need to learn a new language.

1.2. Developing Narrative and Computational Skills in Tandem

Although the question of using natural language for programming applies to general purpose programming languages, it applies equally, and perhaps even more so, to specialist programming languages. Within the broader "programming for all" context, our motivation for investigating the use of natural language for programming stems from our longstanding research into game creation for young people (Robertson and Good, 2005a,b; Good and Robertson, 2006; Howland and Good, 2015).

Game creation, and role-playing game creation in particular, provides a unique environment which interweaves narrative skills with computational skills. Good role-playing games have complex, interactive, branching plots, with richly developed characters and well thought out dialogue. As the complexity of an interactive narrative increases, so too does the computational complexity required to bring about the interactions. As such, as young people's narrative skills develop within this context, so too should their computational skills, and vice versa.

Game creation also have a very compelling motivational aspect: young people find it highly motivating to create games which are similar in style and appearance to the commercial games that they are used to playing, and are willing to devote considerable time and effort to doing so. Much of our research has focussed on how best to support them in their game creation tasks, whilst at the same time fostering the development of both their narrative and computational skills.

As the medium for our research on game creation, we used the Electron toolset (Figure 1), which ships with the Neverwinter Nights 2 (NWN2) game. The advantage of the Electron toolset is that it has a "low floor and high ceiling": specialist skills are not required to start creating a game with commercial quality 3D graphics. At the same time, the toolset allows users to produce games of substantial complexity: indeed it was used by the game developers to create the NWN2 in-game tutorial.

Although the Electron toolset provides good support for the visual aspects of game creation, and for the creation of character dialogue, the scripting language incorporated in the toolset, NWScript, presents a significant barrier. As young people progress with games development, they will find themselves needing to script more complex events to make their game work in the way they wish, and to create more interesting and interactive experiences for the player. For example, scripts can be written to reward the player with treasure when they slay the dragon, or cause a wizard to vanish when they cast a spell. However, NWScript is based on C, with a similarly complex syntax (see Figure 2 for an example).

We have run game making workshops with over 350 young people to date, and have consistently observed that they are able to use natural language quite accurately to describe what they want to happen in their games, for example "this troll should attack the player if they refuse to pay the toll". Although generally correct, descriptions are sometimes underspecified, requiring additional prompting from workshop staff before they are complete (Howland et al., 2007). Unfortunately however, after correctly specifying the rule verbally, none of the 350 participants were able to learn NWScript sufficiently well to script their own events, and had to rely instead on the

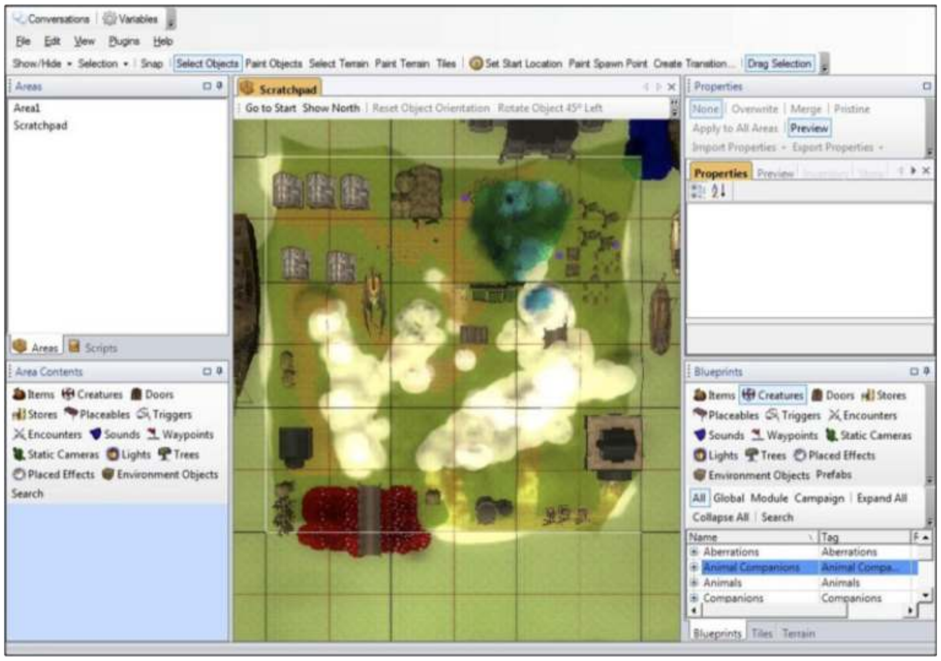
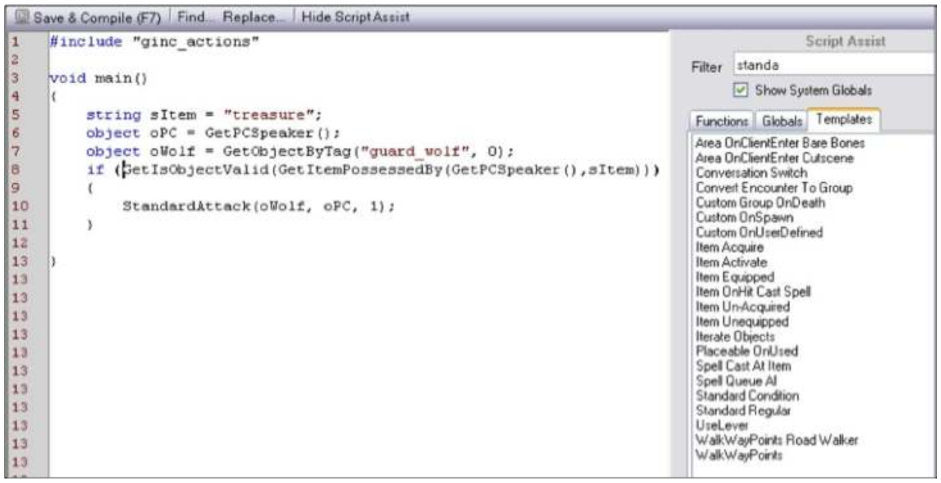

Conversations
Variables
File Edit View Plugins Help
Show/Hide Selection Snap Select Objects Paint Objects Select Terrain Paint Terrain Tiles Set Start Location Paint Spawn Point Create Transition... Drag Selection
Areas
Area1
Scratchpad
Scratchpad
Go to Start Show North Reset Object Orientation Rotate Object 45° Left
Properties
None Overwrite Merge Pristine
Apply to All Areas Preview
Import Properties Export Properties
Properties Preview
Areas Scripts
Area Contents
Items Creatures Doors
Stores Placeables Triggers
Encounters Sounds Waypoints
Static Cameras Lights Trees
Placed Effects Environment Objects
Search
Blueprints
Items Creatures Doors Stores
Placeables Triggers Encounters
Sounds Waypoints Static Cameras
Lights Trees Placed Effects
Environment Objects Prefabs
All Global Module Campaign Expand All
Collapse All Search
Name Tag
Aberrations Aberrations
Animal Companions Animal Compa...
Animals Animals
Companions Companions
Blueprints Tiles Terrain

Figure 1: Neverwinter Nights 2 Electron Toolset Interface



```
#include "ginc_actions"

void main()
{
    string sItem = "treasure";
    object oPC = GetPCSpeaker();
    object oWolf = GetObjectByTag("guard_wolf", 0);
    if (GetIsObjectValid(GetItemPossessedBy(GetPCSpeaker(),sItem)))
    {
        StandardAttack(oWolf, oPC, 1);
    }

}
```

Figure 2: An example of NWScript

workshop facilitators to translate their story ideas into scripts.

Therefore, although the toolset provided a motivating environment for the development of narrative skills, NWScript's complex syntax acted as a barrier to coding, and to the underlying computational concepts. However, we felt that young people's existing narrative understanding of game events and their ability to describe events in natural language was a good starting point for designing a new language which would allow them to engage with basic computational concepts.

Our initial aim was to develop a fully bi-modal programming environment to be incorporated into the Electron toolset. The environment would use both a full sentence natural language representation and a blocks-based representation. Both representations would be fully editable, with any code written in one representation dynamically rendered in the other. Therefore, users would be able to write scripts in natural language, similar to the way they described game events to the workshop staff, and the system would create and display the blocks-based equivalent of their script. As young people developed their understanding of computation and became more proficient at coding in the blocks-based language, the natural language equivalent generated by the system would allow users to check that their blocks-based program was correct, and worked as they had intended.

Given that blocks-based languages are already well established and currently very popular (see, e.g. (Cooper et al., 2000; Resnick et al., 2009; Gray et al., 2012)), our initial investigations focussed on the natural language aspect of the proposed environment. Specifically, we were interested in the extent to which natural language could be used for all of the diverse activities that comprise programming, including code generation, comprehension, debugging and, in many cases, collaboration.

In this paper, we present three design studies, each of which built upon the findings of the former, and addressed the role of natural language for programming. The results from our first study (Section 3) suggest that using unconstrained natural language for code generation leads to confusion about which specific natural language words and phrases are admissible, which in turn leads to numerous syntax errors. Further confusion arises when trying to ascertain where natural language is being used to issue commands (i.e. for coding) or as natural language (e.g. as strings within the program). In a second study (Section 4, and reported in full in (Good et al., 2010)), the researchers effectively acted as the language interpreter: although this reduced the number of incorrect programs, the number of incomplete programs remained high. The best results were obtained in our third design study, where we constrained the language by providing novices with a set of language primitives with which to construct computational rules (Section 5).

Overall, the design studies suggested that full-sentence natural language (at least in an unconstrained format) is not well suited to code generation, however, it can provide helpful support for code comprehension, debugging and collaboration tasks (Section 6). We distilled these findings into a set of design guidelines (Section 7) and implemented them in a bi-modal language called Flip (Section 8). A constrained blocks-based representation is used for program generation, while a dynamically generated (non-editable) natural language representation of the code has been designed to be used for comprehension, debugging and collaboration. Flip was evaluated in a series of real world empirical studies, and we report on two such evaluations which consider the extent to which the design guidelines embodied in Flip were able to provide effective support for novices (Section 9).

This paper builds on work presented at VL/HCC 2015 ((Good and Howland, 2015)), and includes a fuller discussion of previous work on natural language programming languages (Section 2) and a consideration of the relationship between programming tasks and notations (Section 6). It also presents a refined set of design guidelines, organised by programming activity (Section 7), as well as an additional study not reported in the original conference paper (the 'school' study reported in Section 9.2).

2. Previous Research on Natural Language for Coding

The question of whether it is possible to program in natural language is by no means new: similar arguments have been made at least as far back as 1966 (Sammet, 1966). Although sometimes conflated with the "natural programming" movement, proponents of natural programming aim to make programming more accessible by aligning it more closely to the ways in which people naturally think about the problems they are trying to solve. However, this does not imply that natural language is the appropriate notation for doing so (Pane et al., 2001).

A number of empirical studies have examined the feasibility of programming in natural language, and the results have been (unsurprisingly) mixed. Biermann et al. (1983) found that participants with limited programming experience were able to use a natural language programming system (NLC) to solve two distinct problems, one involving linear equations and the other involving gradebook averaging, with an overall success rate of 73.9%. Of the English sentences typed, 81% could be interpreted, and the authors suggest that many of the errors that did occur were easily resolvable. Capindale and Crawford (1990) also found that natural language is an effective means of expressing database queries, particularly for users with prior knowledge of the contents of the database, regardless of their aptitude with computers.

Miller carried out a number of studies investigating the potential for natural language to be used for programming (Miller, 1975, 1978, 1981). When examining natural language descriptions of programs, he noted that they contained many omissions. Similarly, when examining novice use of tables to write programs, he found that the information which participants entered was generally correct, but that the solutions were incomplete. Additionally, he found that natural language descriptions tend to state the action first, with conditions, when expressed, appearing in the form of qualifications to the action (Miller, 1978). Miller suggested that these results may have been due in part to the instructions given to participants: as they were asked to write the instructions for another person, it may be that certain things were left implicit, with the assumption that meticulous instructions were not required, a view shared by (Galotti and Ganong, 1985, p. 3), who note that it is "...bad form to belabour the obvious". Indeed, a subsequent study by Galotti and Ganong (1985) suggested that non-programmers do in fact include control statements in their instructions in situations where it seems appropriate (i.e. where one cannot assume that the recipient has the ability to go beyond the information given).

Capindale and Crawford (1990) provide a nice summary of the most suitable conditions for natural language interactions, suggesting that they are most effective for "question answering tasks" in a limited domain, with users who have prior knowledge of the domain and where the system provides adequate feedback about any restrictions to the language.

These empirical studies have been complemented by the design of systems which fall broadly into two categories. On the one hand, there are those which aim to automatically convert natural language input into code written in an existing programming language, even if not completely specified, for example, Metafor (Liu and Lieberman, 2005a,b) and MOOIDE (Lieberman and Ahmad, 2010). The code fragments which are generated can then be used as scaffolds, giving the user a framework which can be completed or modified, rather than having to write the code from scratch. On the other hand, there have been attempts to develop natural language-like programming languages which are themselves executable, of which Inform 7 is probably the most well known example (Nelson, 2006, 2011), and which we used as the basis for a study into the feasibility of natural language for coding, described in the following section.

3. Inform 7 Study

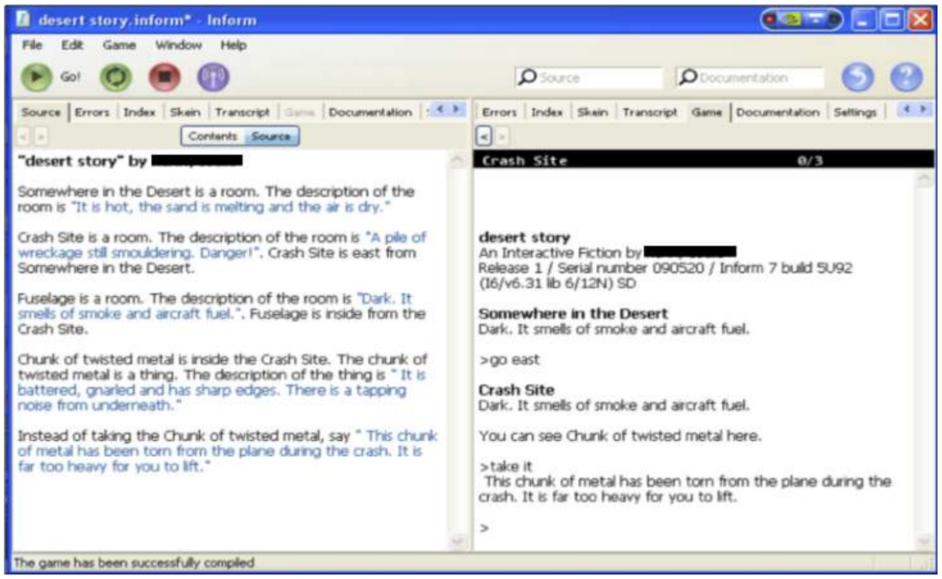
In order to gain an in-depth understanding of how programming languages based on natural language are used by novices, we conducted a design study using Inform 7 (inform7.com). Inform 7 is a programming language and environment for the creation of interactive fiction (digital text-based adventures), designed to be accessible to non-programmers. Our decision to use Inform 7 was motivated by the fact that, firstly, Inform 7 is one of the few fully functional natural language programming languages and, secondly, its focus on interactive narrative shares similarities with our focus on narrative-based interactive role-playing games.

Unlike previous versions of Inform, which used a C-like syntax, the Inform 7 programming language is very similar to natural language, and is designed to 'read like English'. Figure 3 shows the Inform 7 interface: the left-hand pane shows a sample of Inform 7 code, while the right-hand pane shows the interactive narrative from the player's perspective.

In order to examine the ease of use, learnability and comprehension of a natural language programming language, we ran a three hour workshop in which we observed non- and novice programmers as they used Inform 7 to create pieces of interactive fiction. Although our ultimate aim was to create a programming language for young people aged 11-16, there were no natural language programming languages in existence for this age group, therefore, we conducted this initial study with adult users.

3.1. Method

Nine university students (8 female and 1 male, aged 18-42) took part in the workshop. Participants were recruited via posters put up in the University's English Department and by emailing various English course groups. We specifically targeted students of courses with a composition element as we wanted people with skills and an interest in creative writing, but without much, if any, programming experience. We advertised the workshops as a chance for participants to learn how to create their own piece of interactive


desert story.inform* - Inform
File Edit Game Window Help
Go!
Source
Documentation
Source Errors Index Skein Transcript Game Documentation
Contents Source
Errors Index Skein Transcript Game Documentation Settings

```
"desert story" by [redacted]

Somewhere in the Desert is a room. The description of the
room is "It is hot, the sand is melting and the air is dry."

Crash Site is a room. The description of the room is "A pile of
wreckage still smouldering. Danger!". Crash Site is east from
Somewhere in the Desert.

Fuselage is a room. The description of the room is "Dark. It
smells of smoke and aircraft fuel.". Fuselage is inside from the
Crash Site.

Chunk of twisted metal is inside the Crash Site. The chunk of
twisted metal is a thing. The description of the thing is " It is
battered, gnarled and has sharp edges. There is a tapping
noise from underneath."

Instead of taking the Chunk of twisted metal, say " This chunk
of metal has been torn from the plane during the crash. It is
far too heavy for you to lift."
```

```
Crash Site                                   0/3

desert story
An Interactive Fiction by [redacted]
Release 1 / Serial number 090520 / Inform 7 build 5U92
(I6/v6.31 lib 6/12N) SD

Somewhere in the Desert
Dark. It smells of smoke and aircraft fuel.

>go east

Crash Site
Dark. It smells of smoke and aircraft fuel.

You can see Chunk of twisted metal here.

>take it
 This chunk of metal has been torn from the plane during the
crash. It is far too heavy for you to lift.

>
```

```
The game has been successfully compiled
```

Figure 3: Inform 7 Interface

fiction, and stated that no experience was required. A questionnaire administered at the start of the workshop confirmed that none of the participants had any significant experience in programming.

On arrival, we explained to participants that they had the option of participating in a research study during the workshop. Doing so would involve collecting basic demographic information as well as the works of interactive fiction created during the workshop, and taking a screen recording and an audio recording of their comments as they worked on their stories. We explained that we were interested in their thoughts and opinions on any problems they encountered, along with suggestions on how we might build a better language. We made it clear that participation was entirely voluntary and that attendees were under no obligation to take part in the study in order to participate in the workshop. We then obtained informed consent from those who were interested in taking part.

The workshop was designed in such a way as to give participants a gentle introduction to Inform 7, while also providing them with the skills to create their own piece of interactive fiction. After introducing the session, the facilitators gave a whole group demonstration of Inform 7. We started by showing participants how to play a game created with Inform 7 to introduce them to the idea of interactive fiction from the “reader/player’s” point of view, and to show the common commands a player might use. Following this, we went on to create a new Inform 7 story from scratch to introduce participants to the commands they would need to create their first piece of interactive fiction.

Participants were then asked to use Inform 7 to create a story, working in pairs or threes so that we could capture their discussions when coding and debugging their stories. Participants were provided with a worksheet which gave broad instructions for a simple game which they had to create. This was written in language which did not mimic the required syntax for implementing the story, so as to allow us to examine the way participants translated from everyday language to Inform 7 syntax. At the same time, we chose to start with a structured and constrained task so we could analyse participants' use of the language with some knowledge of what they were trying to achieve. Participants were also given two "cheat sheets" to remind them of what was covered in the demonstrations: one with common commands for playing the game and one with helpful syntax for writing a game.

Once they had finished this task, participants wrote a freeform piece of interactive fiction as a group. We encouraged them to discuss and plan out ideas on a whiteboard first, and then to try implementing those ideas. We finished with a debriefing session and an informal whole group discussion in which we invited any additional comments or thoughts.

The audio recordings made during the session were transcribed, paired with the screen recordings, and then coded to identify errors and participant misconceptions with respect to the Inform 7 language.

3.2. Results

To support reader understanding of the errors and misconceptions described, Figure 4 shows an annotated sample of Inform 7 code. The first line is a standalone phrase, and the following two lines are a rule consisting of a rule preamble and a phrase that executes when the rule preamble is met. Phrases and rules must follow a specific syntax, or 'pattern'. In Figure 4, the first phrase has the following pattern, in which the articles are optional: (The) [object] is (a) [description].

All participants found Inform 7 very challenging, and struggled to implement their story ideas and develop a working piece of interactive fiction. Program code must be written using very particular sentence structures and precise keywords and, although the syntax is documented in help files and online manuals, there is no dynamic syntax support when typing in code. As

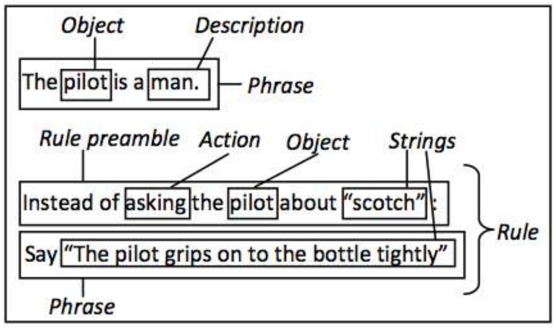

Object
Description
The pilot is a man.
Phrase
Rule preamble
Action
Object
Strings
Instead of asking the pilot about "scotch":
Say "The pilot grips on to the bottle tightly"
Rule
Phrase

Figure 4: Annotated Inform 7 Phrase and Rule

a result, syntax errors frequently stopped games from compiling, and hence prevented users from being able to test their games.

An analysis of the audio and screen recordings allowed us to identify frequently occurring errors (resulting in a failure to compile or undesired behaviour) and misconceptions (conceptual problems which could potentially lead to an error). This led to the development of the error/misconception taxonomy shown below (note that in a further stage of research, the taxonomy could be applied to other datasets to determine relative frequency counts, etc.):

1. Confusion between natural language as a programming language and 'free' natural language (strings)
   - 1a) Placing rule and phrase keywords within strings
   - 1b) Placing descriptive text outside of strings
   - 1c) Wrongly assuming string rules that do not exist
2. Errors using natural language as a programming language
   - 2a) Using synonyms in place of rule and phrase keywords
   - 2b) Incorrect syntax of rules and phrases
     - 2b.i) Incorrect ordering
     - 2b.ii) Adding additional words
     - 2b.iii) Omitting rule and phrase keywords
   - 2c) Using one keyword in place of another
   - 2d) Problems with object names
     - 2d.i) Typographical errors
     - 2d.ii) Inconsistent typing of object names
   - 2e) Wrongly assuming syntax rules that do not exist

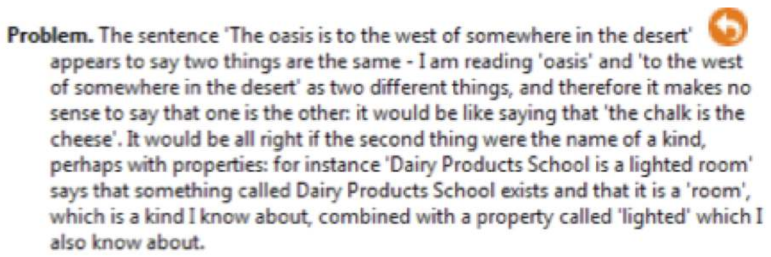
A primary source of errors resulted from confusion about when natural language was being used as a programming language (i.e. Inform 7 commands) and when it was 'unconstrained' natural language (i.e. strings). Participants sometimes placed rule keywords and phrases within strings rather than outside them (1a). For example, one participant included the keyword 'say', used to print to screen, within the string to be printed, i.e. 'Instead of asking the pilot about scotch: "The pilot says 'hands off' and grips on to the bottle tightly"'(the correct rule syntax is shown in Figure 4).

The other main category of errors concerned the use of natural language as a programming language. Semantically similar words were frequently used in place of the correct keyword (2a). For example, a participant who wanted players in the 'Crash Site' area to be able to enter the plane fuselage by typing "go inside" wrote 'the Crashed Plane Interior is in the Crash Site' whereas the correct code is 'the Crashed Plane Interior is inside from the Crash Site', admittedly, a not very English-like syntax. Participants also inserted extraneous words into rules and phrases (2b.ii). A participant who wanted a player who was currently 'somewhere in the desert' to enter an oasis by typing "go west" wrote 'the oasis is to the west of somewhere in the desert' whereas the correct syntax is 'the oasis is west of somewhere in the desert'.

When compilation errors occur, Inform 7 tries to offer helpful feedback. Although designed to simulate a conversation, many participants found the verbose style of these error messages confusing rather than helpful. The error message in Figure 5, generated in response to the error identified above (i.e. writing 'to the west of' rather than 'west of'), was of no use in helping the authors identify their small syntax error. In cases where the error message did help participants identify the problem, they were often unsure how to fix it. In a few cases, the pseudo-conversational style of the error messages led to feelings of frustration at not being able to reply, with one participant asking her group members, "So how do you say 'Yes, actually, you have misunderstood me!!!'?".

3.3. Discussion

Overall, the use of a natural language programming language did not seem to benefit novices: paradoxically, the errors and misunderstandings observed seemed to have stemmed from the very features designed to provide support, i.e. program code that 'reads like English'. Users struggled with the distinction between "natural language as natural language" and "natural



Problem. The sentence 'The oasis is to the west of somewhere in the desert' appears to say two things are the same - I am reading 'oasis' and 'to the west of somewhere in the desert' as two different things, and therefore it makes no sense to say that one is the other: it would be like saying that 'the chalk is the cheese'. It would be all right if the second thing were the name of a kind, perhaps with properties: for instance 'Dairy Products School is a lighted room' says that something called Dairy Products School exists and that it is a 'room', which is a kind I know about, combined with a property called 'lighted' which I also know about.

Figure 5: Inform 7 Verbose Error Message

language as programming language", and were unsure when the language must be constrained, and when syntactic variations were allowable.

Many of these issues reflect essential differences in the way that humans and computers typically "use language". Languages allow for dialogue between two entities, typically humans (although in this case, between a human and a computer). Natural languages have inbuilt redundancy, affording multiple ways of expressing a single idea, and humans are adept at using natural language to generate synonyms at word and phrase levels. Similarly, humans can correctly interpret multiple syntactic variations of a semantic idea. Programming languages, however, lack this redundancy, and compilers are not designed to deal with it: a synonym of a keyword means absolutely nothing.

Using natural language for code generation (i.e. for communication from human to computer) highlights this mismatch between the human ability to generate syntactic variations of a semantic idea, and the computer's ability to understand only one of these. Furthermore, having to write Inform 7 code from scratch, rather than by selecting from a set of words/phrases that the interpreter can understand, multiplies the possibility of syntactic variations.

Communication in the opposite direction, from computer to human, raises other issues. Given the human ability to correctly interpret syntactic variations, the actual phrasing of a communication, such as an error message, is less important as long as it is comprehensible (which is, admittedly, hard to operationalise). However, "human-like" compiler messages set up a false expectation that more extensive human-like exchanges are possible. Indeed, such a situation may exacerbate the formation of "superbugs", where learners believe there is a "hidden mind somewhere in the programming language that has intelligent, interpretive powers" (Pea, 1986, p. 25).

The difficulties experienced by Inform 7 users could potentially be addressed by increasing the levels of constraint and scaffolding. For example, constraint could be achieved by allowing users to select from a set of keywords. Alternatively, a more advanced system could potentially interpret natural language more generously, e.g. not throwing up a compiler error when a user types "description for" rather than "description of". Further problems might be avoided by ensuring that messages from the computer are comprehensible, and do not set up false expectations.

At the same time, a fully functional natural language programming language seems unattainable given that the computer's "knowledge" of natural language will always be limited compared to that of a human. As Rich notes in his paper, *Natural language understanding: How natural can it be?*, natural language in such contexts requires users to "learn not the interface language itself, from scratch, but rather the boundaries that sit within a language that they already know and that divide the recognizable sublanguage from the rest of the natural language." (as cited in (Nardi, 1993)).

4. Natural Expression of Game Rules

To further investigate the extent to which the difficulties experienced by Inform 7 novices were due to the limited subset of language that was understood by the compiler, we conducted a second study examining the ways in which our target users naturally described events and behaviours occurring in a computer game. This study was designed to investigate whether a hypothetical system with a level of natural language understanding similar to that of a human could allow novices to program effectively. More specifically, we were interested in whether, with no programming language syntax to worry about, novice programmers were able to write correct and complete rules.

4.1. Method

The participants were 64 pupils aged 11 and 12 years old (35 female and 29 male), spread across 3 IT classes, all taught the same material by the same teacher. The school was located in the south of England, and consistently performed below the national average in standardised tests.

The study used a game, created specifically for the study using the Electron toolset, which contained seven scripted encounters, each embodying a particular computational structure (e.g. conditionals, Boolean operators), and which increased in complexity. A worksheet was developed to accompany the game which contained seven questions, one for each scripted encounter. The first two questions required pupils to choose from a set of rules the one which they felt correctly captured the behaviour they had observed in the game, while the remaining five questions asked them to write the rule that would produce the behaviour (an example question is shown below):

> *There are two archers standing on top of a hill overlooking a pass, as various creatures run through the pass towards their town. The archers have been given specific instructions on what they should do with these creatures. What do their instructions seem to be? Remember, answer in full sentences, and take careful note of how the archers react to different creatures.*

The activity took place during the pupils' usual IT lesson, which lasted 50 minutes. The same procedure was used in all three classes. Before starting to play the game, pupils were given the following instructions: "Find the village at the heart of the forest. Follow the path to explore the game world and reach the village". The worksheet was distributed to pupils, and the game was available to the pupils throughout the activity so that they could replay the encounters and explore different options if they desired.

4.2. Results

In order to analyse the frequency and type of errors contained in pupils' rule descriptions, a coding scheme was developed, based on the error analysis used in (Good and Oberlander, 2002). An additional analysis considered the linguistic structure of the rules, based on work by Pane et al. (2001). Given the focus of the current paper, only a summary of the error analysis results is presented: a full account of the study can be found in (Good et al., 2010).

The overall rate of rule correctness was low: only 21% of attempted answers were fully correct, while 35% had one element (either trigger/condition or outcome) correct. Errors of omission, in other words, failing to include part of the rule, were by far the most common category, accounting for 74% of all errors. The most common error made with triggers/conditions was to miss this element out entirely (27%), with the second most common error being partially missing triggers/conditions (12%). For outcomes, the most common error was a partially missing outcome (27%), followed by a completely missing outcome (19%). Errors of commission, where participants included incorrect or vague information in their answers, accounted for a relatively small number of errors in both triggers and outcomes (7%).

Overall then, rules were much more likely to be incomplete than inaccurate. The low proportion of inaccurate rules is likely due to the impossibility of syntax errors: because the rules were "interpreted" by humans, multiple syntactic variations were acceptable, provided they were semantically correct. On the other hand, because the exercise was paper-based, pupils were unable to test their rules, meaning they received no feedback on whether their rules functioned as they had intended.

4.3. Discussion

The findings from this study suggest that natural language is perhaps not the most appropriate notation for code generation: although it eliminates syntax errors, it led to a surprisingly high number of omissions in the rules, a finding which is in line with much earlier work on natural language descriptions of code (Miller, 1975, 1978).

From a programming language design perspective, these findings suggest that support designed to minimise syntax issues is insufficient on its own. The environment should also provide robust support for errors of omission, ideally before compiling and testing. For example, rules which contain missing elements could be highlighted, and a "read back" function could allow users to check that their code matches with their intentions.

The following section describes a design study which tested developing hypotheses about the use of natural language for computation. We then synthesise the implications of all three studies into a set of design guidelines for the use of natural language in novice programming environments.

5. Low-Fidelity Card Prototype Study

Findings from the previous two studies indicated that when novice programmers use natural language to specify computational rules in a free-form manner, a surprisingly high number of errors result. However, we hypothesised that the problems arose not because of the natural language aspect per se, but because of the lack of language constraint and support during the code generation process. We therefore wanted to explore whether giving target users a restricted set of language primitives with which to assemble computational rules would result in greater accuracy. Providing this form of syntactical support would avoid the problems of multiple semantically correct, but syntactically uninterpretable, rules seen in the Inform 7 study.

We opted to use natural language for computational keywords, in part because of the known difficulty inherent in expressing abstract concepts, such as disjunction, using graphical representations (Stenning and Oberlander (1995)). However, for specifying objects and characters within a script, it was decided that images would provide a more accurate and easier to use method of picking the correct item and adding it to a script given that they are represented graphically in the NWN2 environment. The aim was to provide the most direct mapping for users, eliminating the need to translate between a visual representation of a character or object, and its name.

5.1. Method

The study was conducted in two different contexts: a classroom setting and an out-of-school holiday workshop setting. In total, 20 young people took part in the study. The participants in the school part of the study were 8 young people aged 11 and 12 (2 female, 6 male), who were taking part in a game making project in their IT class. In the workshop context, participants were 12 young people aged 11-15 (all male), who had elected to attend a game making workshop during their school holidays. The same procedure was used in both contexts.

The study used four types of laminated cards: *action cards*, *thing cards* (objects and characters), *connecting cards* (representing control logic such as 'if', 'until'), and *description cards* (representing object state). The *thing cards* graphically depicted common characters and objects in the game world, while the words used on the other three card types were drawn from the corpus of natural language rule descriptions generated from our study on young people's expressions of rules in natural language (Section 4). A number of blank cards were also included should participants wish to create new cards.

We were interested in determining the ease of both language comprehension and code generation. Participants were therefore first asked to read out the meaning of rules that had been constructed using the cards. There were 25 rules, which increased in complexity. This was followed by a program generation activity in which we read out a statement such as "The wolf attacks the player, but only if the player is carrying the treasure" (15 in total, again increasing in complexity) and asked participants to use the cards to write the corresponding rule, as shown in Figure 6.

5.2. Results

Overall, the majority of participants were able to construct correct rule descriptions using the cards. For the composition task, the percentage of fully correct rules was 77% in the school study, and 76% at the holiday

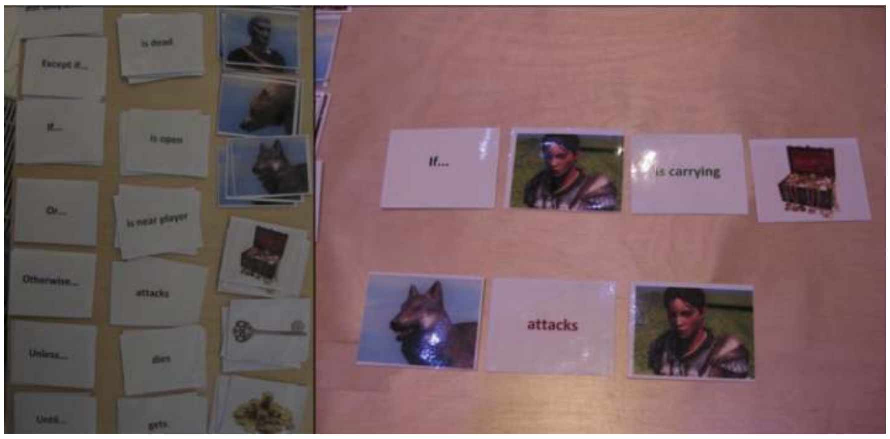


Figure 6: Constructing Rules with Paper Prototype Cards

workshop. There did, however, seem to be some confusion between different computational categories, particularly states and actions. A state 'is open' was often used instead of the action 'opens'. Similarly, 'when' and 'if' were often used interchangeably.

5.3. Discussion

In comparison to the study in which pupils wrote rules in unconstrained natural language (Section 4), the percentage of correct rules was considerably higher, indicating that choosing from a limited vocabulary reduced errors substantially. This is not surprising in and of itself, however, it is interesting to note that while graphical languages require users to choose from a set of pre-existing blocks by necessity, most text-based languages do not, instead requiring them to type code from scratch, which introduces syntax errors in the process. While some suggest that graphical languages are superior to textual languages for novices (see (Whitley, 1997) for an early discussion of this debate), it may be that any ease of use results from the constrained nature of graphical languages, rather than their graphicacy per se. Finally, the confusion between different types of computational constructs suggests that support should be provided to help users differentiate between them.

6. Summary of Design Studies: The Role of Natural Language for Computation

In our study investigating the use of a natural language programming language for code generation (Section 3), we found that unconstrained natural language introduces a number of significant issues. Some of these stem from confusion as to how language is being used in any given instance (i.e. is this "real" natural language, or "programming" natural language?). Even when the user is aware of the distinction, the human ability to generate multiple semantically equivalent phrasings comes up against the compiler's ability to understand only one of these phrasings. Finally, more "natural" communicative utterances from the computer, rather than helping, exacerbate the problem by suggesting to the user that it is possible to engage in a dialogue with the system when it is not in fact the case.

Our study of the ways in which young people use natural language to convey computational concepts (Section 4) suggested that eliminating syntax and compiler issues did not completely eliminate errors, however, the errors took the form of errors of omission, rather than errors of commission.

In our language design study (Section 5), we found that constraining the programming language results in far fewer errors. As noted earlier, although perhaps unsurprising, it suggests that visual languages may derive some of their cognitive tractability from the limited power of expression inherent in graphical representations, e.g. their difficulty in representing alternative possibilities (Stenning and Oberlander, 1995). Finally, when language is used for computational keywords, novices appear to require support not only for understanding a particular computational concept, but also for determining the computational category to which it belongs (e.g. trigger, state, condition).

The starting point for this series of studies was a question about how young people's narrative understanding of computational events, expressed using natural language, might be used as a basis for fostering the development of their computational skills (through computational activities which encompass not just program generation, but comprehension, debugging and, in many cases, collaboration).Considering the range of computational activities broadens the scope of enquiry and allows us to consider their common requirements. For example, all of these activities require some form of notation which can be used by the actors involved (both human and machine 'actors') in order to communicate with each other. At the same time, the specifics of each activity differ and by not focussing solely on determining the most appropriate notations for coding, we can consider whether different activities might require different notations. We concluded that natural language does not appear to be well suited to program generation, however, it is well suited to the activities of comprehension, debugging and collaboration, perhaps more so than programming languages themselves.

In the case of program generation, using natural language as the base notation leads to difficulties in that humans are asked to use a language with which they are familiar in a very different way, and for very different purposes. At the same time, computers require more computational sophistication to interpret natural language (as compared to a traditional programming language), and to respond appropriately: although substantial progress continues to be made in this area, misinterpretation and miscommunication on the part of the system can have serious consequences on a learner's ability to progress, as we found in our study of Inform 7.

Unlike code generation, activities such as comprehension, debugging and collaboration require notations that are optimised for human ease of use, as these activities typically involve communication with oneself (in the case of reading back the source code, or other notation, to try and understand and/or debug one's code) and with others (in the case of programmer collaboration, or where a teacher or peer may be trying to understand a student's program in order to provide support).

In keeping with our focus on using natural language to support computational activities, the next section presents a series of design guidelines, derived from the studies described above, and organised according to whether they focus on code generation, or on comprehension, debugging and collaboration.

7. Design Guidelines

Our three studies showed that full-sentence natural language remains problematic for program generation, but can be very useful for supporting comprehension, collaboration and debugging. We have synthesised our findings into a set of design guidelines for the effective use of natural language in novice programming environments. Our focus is on how such environments can support young people to 1) create correct and complete computational rule specifications and 2) develop an understanding of computational concepts, and the skills to use them. The design guidelines are separated into those supporting the use of natural language in program generation, and those supporting its use for comprehension, debugging and collaboration. The guidelines are described below, along with a reference to the study (or studies) from which they arose (indicated as “S1” for study 1, etc.).

For program generation:

1. **Constrain expression during program generation**: in novice programming environments, expression should be constrained to prevent syntax errors. This can be achieved by using draggable blocks with text for code composition, or through an through autocomplete functionality in the case of purely textual languages (S1, S2, S3);
2. **Clearly distinguish ‘code’ from free-text**: it should be clear to the user whether the language is true natural language (e.g. a conversation line in a story) or is being used as code. For example, computational constructs should appear in a different font/colour in textual languages, while in blocks-based languages, strings and code blocks should be clearly separated (S1, S3);
3. **Highlight distinctions between different computational categories:** make phrases from different computational categories visually distinct to avoid confusion, particularly when the natural language phrases are similar (e.g. the action “opens” vs. the state check “open”). Colour and shape can also be used to provide additional cues (S3);
4. **Make underlying structure visible to avoid errors of omission or commission**: when composing code it should be clear where a particular construct requires completion by another element (and of what type), and how constructs can be combined to make well-formed statements. Colour-coding, shape-matching and the use of prompts can help to avoid errors before they occur (S1, S2, S3);

For comprehension, debugging and collaboration:

5. **Provide a full-sentence natural language description of the code**: the programming environment should allow the user, and other users, to easily understand and review the code they have written. A full-sentence natural language representation of the program can allow users to more easily understand the behaviour the code will bring about, making it easier to identify semantic errors in their own code, and get up to speed with someone else’s code (S1);

6. **Use 'natural' natural language**: where full-sentence natural language is used, it should be consistent with typical everyday use, as far as possible, avoiding unnatural syntax or phrasing. Any error messages should be brief and understandable (S1);
7. **Do not suggest the system can engage in dialogue when it cannot**: care should be taken to avoid giving the impression that the system is able to engage in a dialogue with the user when it cannot, or more broadly, is more "intelligent" than it actually is. An environment that encourages a conversational style without being able to achieve it may actually exacerbate the formation of superbugs (S1).

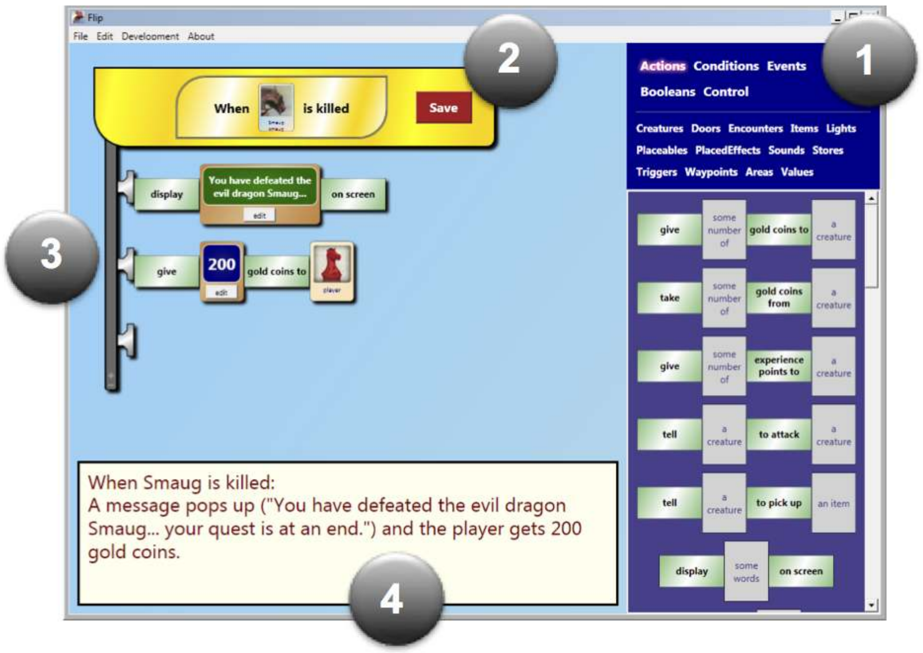
These empirically derived guidelines can be used in the design of new languages, and to analyse the properties of existing novice programming environments that make use of natural language. The guidelines were implemented in Flip, a bi-modal language for game creation, described below.

8. The Flip Language

8.1. An Overview of Flip

Flip is a language for scripting game events, and is designed to replace the need for NWScript when creating games using NWN2's Electron toolset. Novice programmers can create scripts by dragging and dropping blocks into a workspace and then connecting them together on a spine. The Flip interface also features a full natural language description of the script under construction: as the user adds blocks to their program, the natural language description updates dynamically. Figure 7 shows the basic Flip interface, while the points below describe each numbered feature (note that a full description of Flip can be found in (Howland and Good, 2015)) .

1. The **Block Box** contains the blocks used to create scripts, organised by computational category and object type. Selecting a category from the top panel highlights it, and displays all of the blocks in that category. Blocks represent computational concepts (actions, conditions, states), and are colour-coded by type. Blocks have slots that must be filled by objects of a certain type. The slots are similarly colour-coded to indicate which types of objects they can receive, and empty slots indicate, in natural language, the type of slot filler required.
2. The **Event Slot** takes a single event block, which dictates when the script will execute.


Flip
File Edit Development About
When is killed
Save
display
You have defeated the evil dragon Smaug...
edit
on screen
give
200
gold coins to
player
Actions Conditions Events
Booleans Control
Creatures Doors Encounters Items Lights
Placeables PlacedEffects Sounds Stores
Triggers Waypoints Areas Values
give some number of gold coins to a creature
take some number of gold coins from a creature
give some number of experience points to a creature
tell a creature to attack a creature
tell a creature to pick up an item
display some words on screen
1
2
3
4

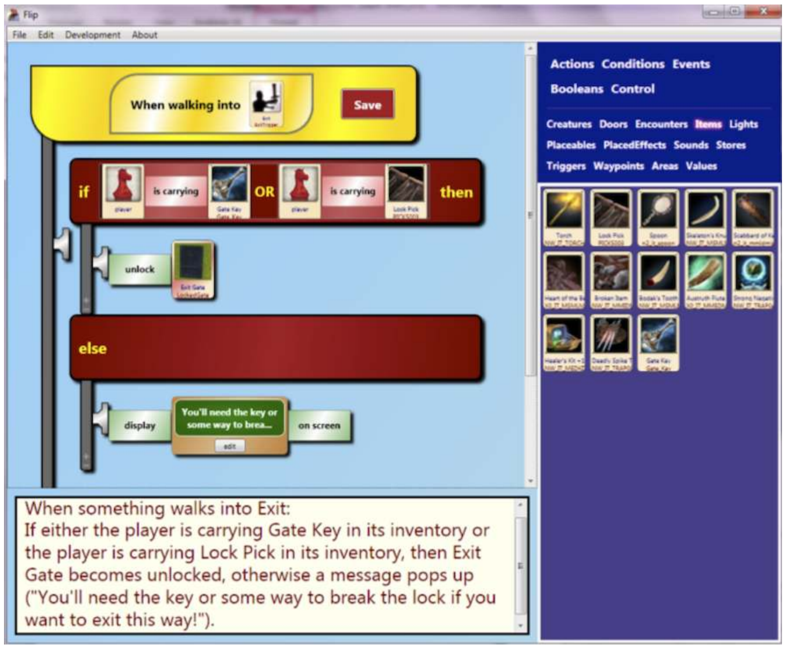
When Smaug is killed:
A message pops up ("You have defeated the evil dragon Smaug... your quest is at an end.") and the player gets 200 gold coins.

Figure 7: The Flip Interface

3. The **Spine** is where the script is composed, by attaching blocks to the silver pegs. The spine can be extended indefinitely to accommodate additional actions, and scripts execute in a top down manner.
4. The **Natural Language (or 'plain English') Box** shows a natural language description of the script under creation. It is automatically generated, and dynamically updated each time the user makes a change to their script. The natural language box gives a full description of the script meaning, explaining in detail what will happen and under what conditions. Whilst the words on the blocks are designed for brevity, and to reflect the most common way of describing the underlying computational concepts, the natural language description is more complete.

By using the blocks available in Flip, game designers can quickly create some fairly complex behaviours. In Figure 8 we show how the designer can set up a locked gate which can only be opened if the player has discovered the key, or if they have purchased a lock pick.

The event chosen for this script is someone walking into a trigger. The designer has marked out a trigger area in front of the gate within the game


Flip
File Edit Development About
When walking into
Save
if
is carrying
OR
is carrying
then
unlock
else
display
You'll need the key or some way to brea...
edit
on screen
Actions Conditions Events
Booleans Control
Creatures Doors Encounters Items Lights
Placeables PlacedEffects Sounds Stores
Triggers Waypoints Areas Values

When something walks into Exit:
If either the player is carrying Gate Key in its inventory or the player is carrying Lock Pick in its inventory, then Exit Gate becomes unlocked, otherwise a message pops up ("You'll need the key or some way to break the lock if you want to exit this way!").

Figure 8: Conditional Script Example

world, selected it from the Triggers panel within Flip, and added it to the empty slot in the event block. An If . . . Then . . . Else block has been chosen from the Control panel to create the desired conditional behaviour.

To create the conditional check, the user has used a Boolean Or block within the If slot, and filled the two sides of the Or block with the conditions under which the actions on the Then spine should be run: If the player is carrying a certain key, or if the player is carrying a set of lock picks. The Then spine contains one action the unlocking of a gate which will happen if either of the conditions placed in the Or block are true. If the condition within the If slot evaluates as false, that is if the player is not carrying either the key or the lock pick, the Else spine will run and the player will be shown a message explaining what they need to do to exit.

The natural language box gives a full description of the script meaning, explaining in detail what will happen and under what conditions. The natural language box automatically generates a description of all blocks attached to the spine, and is dynamically updated every time a change is made.

For program generation:

1. **Constrain expression during program generation**: achieved through the use of a set of graphical code blocks, eliminating the need to type code from scratch. Also achieved by making the natural language component of the language non-editable, thus preventing the user from inadvertently introducing syntax errors;
2. **Clearly distinguish 'code' from free-text**: code appears in Flip's graphical code composition interface, while its natural language equivalent appears in the plain English box. Natural language used in the script (e.g. messages to be displayed to the player) is shown as a separate, colour-coded block within the graphical language, and in quotes in the plain English box. Furthermore, Flip uses correct computational terms to represent the underlying concepts on all interface menus and graphical programming blocks, allowing young people to begin to learn and use the "language of computation". These computational keywords are automatically translated into fuller, everyday language in the plain English box, further distinguishing computational language from natural language;
3. **Highlight distinctions between different computational categories**: blocks are visually organised and colour coded by computational category (actions, conditions) to avoid confusion between terms that are similar from a natural language perspective ("opens" vs. "is open") but computationally distinct (action vs. state);
4. **Make underlying structure visible to avoid errors of omission or commission**: blocks and slot fillers are differentiated by colour and shape, with colour used to link blocks with their corresponding slot fillers. This feature aims to support users' understanding of differing types of command, and to help them avoid errors of commission. A further feature prevents blocks of the wrong type from snapping into place, thus preventing type errors. Errors of omission that prevent the script from compiling trigger a short error message (see point 7);

For comprehension, debugging and collaboration:

5. **Provide a full-sentence natural language description of the code**: in addition to the code blocks used for program generation,

Flip's "plain English" box provides novices with a description of the code they have written (whether complete or not) in full-sentence natural language. This allows them to not only read their code back to themselves (and identify any errors or omissions), but allows others to quickly get up to speed with their code.

6. **Use 'natural' natural language**: Flip's "plain English" box is designed to be as similar as possible to spoken English (e.g. when describing one's script to another person), avoiding any unusual turns of phrase. Phrasings were derived, as much as possible, from the corpus of rule descriptions gathered from target users (Section 4);
7. **Do not suggest the system can engage in dialogue when it cannot**: written error messages from the system to the user are rare, as syntax errors are prevented. The few error messages that are necessary (when a crucial element has been omitted) have been kept very short, in order to avoid suggesting that the compiler possesses a human-like capacity for language comprehension and dialogue.

9. Evaluating the Design Guidelines in Flip

Flip has been evaluated in a number of empirical studies in different settings, including an investigation focussing on whether its use led to an improvement in young people's computational abilities overall (Howland and Good, 2015). Here we describe two evaluations that considered the extent to which Flip is usable by young people for program generation, and supports their understanding of computation. We were also interested in how the design guidelines embodied in Flip contributed to the above two points, i.e. whether they contribute to Flip's usability and provide support for program generation and computational understanding.

The two evaluation studies described here were carried out in three different real-world game creation contexts. Study one was an observational study that took place in a secondary school classroom over the course of a 2 hour lesson. The study focussed, in detail, on the ways in which users interacted with the Flip interface. In our analysis we looked at how the range of features our design process implemented were used by our target users.

The second study was longitudinal, and looked at the use of Flip over the course of two complete game creation projects, one taking place in a school context, and the other taking place in a holiday workshop context. We examined the scripts created over the course of the projects, and conducted end of project interviews and surveys.

9.1. Flip observation study

9.1.1. Participants

Twenty-one pupils aged 11-12 (9 male, 12 female) took part in a two hour session in an IT suite at a secondary school in Scotland. All but one of the pupils had taken part in an 8 week game creation project in the previous year as part of their IT lessons. That project had used the Electron toolset, so pupils had prior knowledge of all other aspects of game creation, but had not previously used the Flip language.

9.1.2. Method

Pupils were told that they would be testing Flip, a new language designed to work with the game creation toolset they had previously used. We explained that we valued their previous experience with the toolset, and particularly wanted honest feedback on how Flip might be improved. We emphasised that Flip was designed to make it easier for them to make more complex events happen within their games. After this background, all pupils were shown an initial introductory video which demonstrated how to add a script which would execute when a conversation line is said.

Pupils worked in pairs (with one group of three), and were asked to try adding a script in the way demonstrated. After 20 minutes, pupils were called back for a second demonstration in which they were shown how to use event types other than conversation lines to trigger their scripts. After a further 20 minutes working on their games, the pupils were given a final demonstration on how to use control blocks to add conditionals to their scripts.

Four researchers were present in the classroom along with the classroom teacher. One researcher gave the demonstrations and took primary responsibility for answering questions from pupils, whilst the other three researchers acted primarily as observers. A number of video recordings of onscreen interactions were made, and log files and scripts were collected for analysis. Researcher notes were collected and written up after the session, and the video footage was transcribed and analysed.

9.1.3. Results

Flip usage, usability and understanding

All of the participants succeeded in creating at least one full script with an event and action(s), while seven of the ten pairs successfully added conditionals to their scripts. Pupils commented that Flip represented a definite improvement in terms of allowing them to write their own scripts, with one pair noting excitedly that Flip was "really good".

Action blocks were well understood, with no observed instances of confusion. Participants were also able to make use of the extending spine to create sequences of action which were much longer than the default spine length.

Events seemed to cause problems for a few pupils. Although event blocks were a different shape and colour to other types of block, the four instances of incomplete scripts that we observed were due to the lack of an event block. In one case, a user tried to save a script without an event, and received an error message, then quickly added an event and successfully saved it. In another case when trying to save with a missing event the pair in question did not understand what to put in the event block, but after consulting the pair sitting next to them they were successful in filling it out and saving the script. Two similar cases were observed where one user thought the script was ready to save, but their partner pointed out the missing event. There were no observed cases of users trying to save their script when missing other elements such as objects in actions or conditions.

There were some instances of initial confusion around the distinction between different computational categories, similar to the design study, but this appeared to be limited to instances involving the conditional slot of control blocks. A few users tried to drag the wrong type of block into the conditional slot, but could tell that something was wrong, as the block would not snap into place. This caused momentary frustration for one or two users, but in each case it appeared to alert them to the fact that the block was not of the right type to fit. In most cases this led to them trying another block type, with eventual success, and in one case it led them to ask for an explanation and help from a researcher. In another case, after attempting to add an action to a slot requiring a condition, one participant said to her partner "We need something that's that colour", pointing to the pink colour of the empty slot. On switching to the Conditions menu, they immediately noted that condition blocks were similarly pink, and successfully completed their control block. The same issue was observed with another pair, with a similar method of resolution, but without an explicit discussion of colour. Overall, the difficulties related to conditionals suggested an interface issue in relation to the distinction between different types of computational categories, rather than a more general misunderstanding around the use of conditionals.

Use of the natural language box

There were numerous examples of the natural language description being used as a sense checking mechanism, with ten recorded cases of participants reading the natural language aloud, before deciding whether it described their intentions accurately. In two of these cases, the reading of the natural language led to a revision of the script as the participants realised it would not do what they intended, and in the remaining cases it gave participants the confidence to move on to the next stage: testing the event in game.

There were two examples of pupils reading the natural language description and paraphrasing it in order to explain their code to either a researcher or a peer. There are likely to have been a number of other cases where pupils read the natural language description silently to themselves, but we were unable to capture these instances through observation only.

Overall, there were many examples of pupils confidently discussing computational concepts, including conditionals, by the end of the session. This observation is supported by teacher comments from a study reported in (Howland and Good, 2015) in which the teacher noted that Flip provides pupils with a language in which they can express computational concepts.

9.2. Flip longitudinal studies

Further to the small-scale observation study described above, we conducted two longitudinal studies in which Flip was used over the course of two game making projects: one taking place in a local secondary school, and the other in a voluntary holiday workshop.

9.2.1. Participants

In the school project, 56 young people aged 11-12 (30 female, 26 male) took part in a game creation project as part of their IT lessons. The pupils were spread over two classes; both were taught the same material by the same IT teacher. The school's most recent inspection report noted a wide spread of ability, with fewer higher attaining students compared to the national average. The school also had higher proportions of pupils with special educational needs, and pupils from lower socioeconomic backgrounds. The pupils in the study had not previously learnt any programming languages (textual or visual), or taken part in any game creation activities at school.

The project ran over one half-term (7 weeks), with 11 fifty minute lessons scheduled for each class (although both classes missed two lessons due to last minute timetable changes and end of term activities). As a result, pupils across both classes spent approximately 7.5 hours on the game creation project. Thirteen of the 56 school pupils were interviewed at the end of the project (7 girls and 6 boys, representing a range of ability levels), while 40 of the 56 pupils completed an end of project survey.

In the holiday workshop setting, 14 young people aged 11-15 (1 female and 13 males) took part in a game creation project for four days over half term. Participation was voluntary, in response to web and email advertisements. Three participants had experience of the Electron toolset (but without Flip): two from previous workshops run by the authors, and one from an unconnected workshop. Upon arrival, participants completed a survey on their prior programming experience: ten reported no programming experience, three reported 'minimal' experience, while one was unsure.

Participants attended the workshop over four days from 10am to 4pm. With a 45 minute lunch break per day, and one hour spent on related activities at the beginning and end of the workshop respectively, participants spent approximately 19 hours on the game creation project.

9.2.2. Method

In both settings participants were given demonstrations by the authors, introducing them to the key tools and functionality of the Electron toolset and the Flip language. These demonstrations took the form of instructional videos which presenters explained as they went along and included basic toolset features (using the terrain editor; adding objects and characters and playtesting; conversation writing; interior areas and area transitions) and creating scripts with Flip (using Flip to add actions to conversations; using events and conditionals). In addition, the workshop participants received an additional tutorial on more complex conditionals (using Boolean elements And, Or and Not) as they had more time on the game creation project.

At the end of the school project, semi-structured interviews were carried out with thirteen of the 56 pupils. Screenshots of the Flip interface were used as prompts during the interview, and interviews were audio recorded and later transcribed. Additionally, all pupils were asked to complete an online survey at the end of the project, which asked a range of questions about their experiences over the course of the game creation project, including 11 questions about Flip. As the survey was carried out at the end of term when normal classes were disrupted, only 40 of the 56 pupils completed the survey.

At the end of the holiday workshop all fourteen young people were interviewed using the same question framework as that used in the school. Again, all interviews were audio recorded and later transcribed. Additionally, in both settings, log files were collected for all participants as well as all modules created and scripts written.

9.2.3. Results

Flip usage stats

In the school project, 70% of pupils (39 of 56) successfully created working scripts. 202 complete and correct scripts were written and saved, however, some of these saves accounted for revisions to scripts a pupil had already worked on. Considering unique scripts only, a total of 132 scripts were written (a mean of 2.36 scripts across all participants, and a mean of 3.38 amongst those who created scripts).

At the holiday workshop, all 14 participants successfully created fully functional scripts. There were 410 script saves recorded, with 260 individual scripts written (a mean of 18.57 scripts per participant). Overall, participants at the workshop wrote considerably more scripts than those in the school project. This is unsurprising given that they spent over twice as much time on the project, and furthermore that they self-selected to attend a game making workshop during their school holidays.

In the school setting, a total of 132 events were used in the scripts (as every script is triggered by an event) while 252 actions were used (a mean of 3 actions per script). 71% of participants (10 of 14) included conditionals in their scripts, with 73 conditionals used in total. Only 15% of the school pupils (6 of 39) added a conditional to their scripts, and between them they added 12 conditionals in total.

In the workshop setting, 260 events were used in the scripts, along with 780 actions (a mean of 1.9 actions per script). 71% of participants (10 of 14) included conditionals in their scripts, with 73 conditionals used in total.

Flip usability

As noted above, during the interviews, we asked participants to discuss their usage of Flip and explain various components of the Flip interface in order to gauge their understanding.

Of the thirteen school pupils interviewed, eight noted that they had managed to make things happen in their game using Flip, three noted that they had used Flip but had not been able to create a working script, while two had not had time to use Flip as a result of missing lessons. Of the pupils who had used Flip, three said that their scripts always worked as expected, while five said that although their scripts did not initially work as expected, they had managed to fix them or find workarounds. Two pupils reported problems with their scripts that they hadn't yet resolved, and one person had not yet been able to test their scripts.

When the eleven pupils who had used Flip were asked about ease of use, one described it as "really easy" to use, and two said it was "pretty easy". Two people stated that it was between hard and easy, and a further three explained that there were both hard and easy bits. The three remaining participants said that they found Flip hard, although one said that after asking her friend for help it made more sense.

The interviews were supplemented by the school survey data, where 12 of the 40 respondents noted that creating Flip scripts was "hard". At the same time however, 20 or the 40 respondents noted that they "enjoyed" creating Flip scripts. The survey also asked pupils about their general understanding of Flip: 32% felt that they understood all aspects of Flip, 63% felt that they understood some of Flip, while only 5% felt that they didn't understand Flip at all.

Figure 9 shows the responses to survey questions on specific activities within Flip. Pupils were asked to express how easy or hard they found each activity on a 5-point Likert scale ranging from Very Easy to Very Hard. There was a relatively small proportion of responses of 'Very Hard' or 'Quite Hard' for all activities, while 'Finding the right blocks', 'Knowing what to put in the When box' and 'Knowing when your script was correct' had the highest number of such responses. The activities with the highest number of 'Very Easy' and 'Quite Easy' ratings were 'Placing the blocks in the right slots' and 'Knowing what action blocks to choose'.

All of the workshop participants interviewed stated that they found Flip easy to use, with one participant stating that it was because "it was very straightforward and the words aren't too complicated to understand", and another noting that "... it's easier than the proper one that came with the toolset".

Flip usage and understanding

When asked about the event block, four of the pupils in the school setting were able to give a full explanation of its purpose. These were mostly in non-technical language, but expressed the correct meaning, for example: "if you put something there then it means ... erm, that at a certain time, when that particular thing happens, erm, everything else in the script will run." One person knew that it needed to be "something that happens in the game",

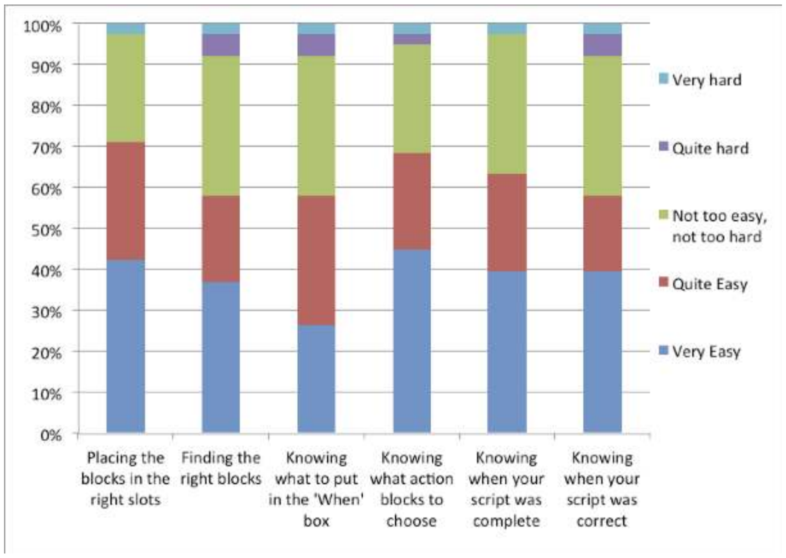


Figure 9: School Survey Results - Relative Difficulty of Flip Features

but was not able to elaborate further. Five pupils understood that the user needs to drag a block into the event slot, but did not realise that it had to be a When block, one of whom said "I think you put one of [the blocks] in there and then you base all of [the rest of the script] on it."

In the workshop setting, 12 participants gave a clear explanation of its purpose, while two were able to provide example of events, but could not provide a more generic explanation. When asked about control blocks, specifically an If ... Then ... block, 10 participants gave a clear explanation of its function, with some able to describe it in computational terms, and others expressing the meaning in more non-standard ways: "...It's an If ...err, equation, so like ...if whatever variable you specify is ...is one way, then it will do ...do whatever it is in the script, but if it isn't, then it won't." The remaining four participants seemed to understand the purpose generally, but could not find the language to express it unambiguously.

Use of the natural language box

When pupils interviewed in the school setting were asked to explain the purpose of the natural language box, five gave full and clear explanations, although one mistakenly thought that the description only appeared after the script had been saved. Two pupils thought it only described what had been put on the spine (although the Event box shown in the screenshot was empty, which may have led to confusion). One pupil thought that it only told the user what they had put in the event box, while another said that it "tells the computer what to do", suggesting that he thought this was a translation for the computer rather than the user. The two remaining people who had used Flip said that they did not understand the purpose of the natural language description.

In contrast, in the workshop setting, eleven of the fourteen participants gave clear, high-level explanations of the natural language box, while the remaining three appeared to understand its purpose, but focussed on specific examples. Seven people noted that they considered the natural language description to be a simplified explanation for when they did not understand the blocks above, with some mentioning that it was particularly useful for complex scripts. Three people described it as a way of checking that the script would work as intended, and spotting where corrections were necessary.

9.3. Evaluation Discussion: Novice Support and Design Guidelines

The evaluation studies showed that novice programmers found Flip fairly easy to use, and that most were able to use it to create scripts for their games. Although the figures indicate that a relatively small number of conditionals were used, particularly in the school setting, this relates to the way that Flip overlays the existing scripting language. Flip provides a trigger-action rule interface for the underlying scripting language, and the variety of custom triggers provided takes away the need for the explicit use of conditionals, except in particularly complex cases. For example, within the scripting language, writing a script which fires when the player enters an area would require a conditional to check whether the person entering the room is the player, as it fires when anyone enters the room. With Flip the user can fill the trigger with a player block and avoid the need for the explicit check.

When looking at the Flip evaluation findings in light of the design guidelines, the main benefits seemed to stem from the constraints built into the language, a clear separation between code and natural language, support for errors, and the use of natural language for comprehension and debugging. In brief, syntax errors were not observed due to the highly constrained nature of the language (DG1), with support built in to the graphical language and the environment as a whole (DG3, DG4). The use of a non-editable natural language representation for code comprehension, rather than code generation, also prevented the introduction of syntax errors (DG5). Pupils seemed to understand the distinction between free-text and code (DG2), with no attempts to edit the natural language code description (DG2).

The natural language representation seemed to be comprehensible to pupils (DG6) and to successfully support individual code comprehension and debugging (DG5). It also acted as a support for communication about computation, with pupils using it to explain their code to others (DG5). Finally, where errors in the scripts could either not be prevented or rectified prior to compiling the code, succinct and clearly worded error messages seemed effective at directing learners to the issue at hand (DG6, DG7).

Some users were observed having initial difficulty distinguishing between blocks of different computational categories, particularly conditionals, suggesting that improvements could be made to the colour and shape cues for this. Notably, this was an issue the design guidelines specifically highlight as a danger, and participants were able to resolve their confusion thanks to cues within the interface designed to tackle confusion of this sort.

Overall, the empirically derived design guidelines allowed us, in the first instance, to determine how to best incorporate natural language into the environment in a way that would be helpful to novice programmers, rather than hindering their progress or even leading to further misconceptions. In turn, reviewing the empirical evaluations of Flip in light of the design guidelines allowed us to see which features were having an impact on program generation, comprehension and debugging, and in what ways.

10. Conclusions and Future Work

This paper re-examined the use of natural language in novice programming environments, investigating whether it is a viable option for program generation, comprehension and debugging. We presented three design studies, the results of which suggested that full-sentence natural language is not well suited to code generation, but can provide helpful support for code comprehension, debugging and collaboration tasks. The paper contributes a set of seven design guidelines, empirically derived from our findings, and implemented in Flip, a bi-modal programming language designed for use by young people in the context of game creation. The findings from the empirical evaluation studies highlight the extent to which the design guidelines embodied in Flip were able to provide effective support for novices.

Although it was not appropriate to use these guidelines to evaluate general purpose languages designed without specific goals regarding the use of natural language, it is useful to briefly discuss the extent to which the guidelines are met by features of existing languages. The are many examples of environments that use a form of constrained natural language for program generation, in line with guidelines 1-4. For example, popular blocks-based languages such as Scratch (Resnick et al., 2009) and Alice (Cooper et al., 2000) use draggable blocks for code composition, make strings easily distinguishable from code, and use colour and shape to make visible different computational categories and the way in which blocks can be combined. However, our studies also suggest that a bi-modal representation of code, one of which takes the form of full-sentence natural language, can support novice comprehension, debugging and collaboration, in line with guidelines 5-7. Natural language is not used in this way by other existing novice programming environments. Our future work will investigate whether including a natural language view in established novice programming environments such as Scratch can provide additional support to novice users, and improve their understanding of and communication about their code.

Although the investigations described in this paper have focussed on the use of natural language for programming, and in a specific context, we believe that the findings and guidelines are of broader relevance to those considering notations in novice programming environments. In the current climate of "computation for all", discussed in Section 1, there has been a renewed focus on determining the "best" language for novices, with comparisons between blocks-based and text-based languages (Price and Barnes, 2015; Weintrop and Wilensky, 2015b)) which are reminiscent of the textual vs. visual programming language debates of the 1990s (Whitley, 1997; Green et al., 1991).

In parallel with this is an implicit assumption that blocks-based languages are a stepping stone to more "serious" text-based languages, a view shared by the learners themselves (Weintrop and Wilensky, 2015a). A number of recent initiatives are looking at how best to facilitate this transition, with systems such as PencilCode (Bau et al., 2015) and DrawBridge (Stead and Blackwell, 2014) allowing learners to move back and forth between the two.

However, whatever the notation used, there is a need to examine them from a finely-grained, cognitive perspective, considering the specific affordances of each one. The well-known 'match-mismatch conjecture' (Gilmore and Green, 1984) suggests that if a language (or notation) highlights a given type of information, then a task requiring that type of information should be easier to perform than one requiring a different type. This conjecture also seems to hold true at the higher level of programming activities such as code generation, comprehension, debugging and collaboration, and when considering the actors involved in the activities (human and machine). Therefore, while a programming language (whether blocks or text based) may represent the most effective means of interacting with a computer, it may not be the best notation for communicating about the program to other users or programmers.

Finally, although our studies showed that natural language descriptions of programs were helpful when communicating and collaborating with others, this is not to suggest in any way that natural language will be "best" in all such cases. As (Nardi, 1993, p. 24) notes, "Conversational communication can be quite constricting and unnatural when what is needed is fundamentally graphic", and research by Guo et al. (2015) similarly highlights the role of visualisations as a basis for communication and collaboration.

Therefore, when thinking about how to design environments for novice programmers, the focus should not be solely on the specifics of the notation used for code generation. Instead, an awareness of the multiple and varied tasks and activities that comprise computation should lead, in turn, to a careful consideration of the notations which can best support each task. By developing notations that help learners to write code, understand it, debug it, and share their knowledge and understanding with others, we hope to create novice programming environments that lower the barriers to computation for all.

11. Acknowledgements

Thank you to the many teachers and young people who so willingly took part in our design work and evaluation studies, and to Keiron Nicholson for his work programming Flip. This research was supported by grant EP/G006989/1 from the Engineering and Physical Sciences Research Council, UK.

12. References

Bau, D., Bau, D. A., Dawson, M., Pickens, C., 2015. Pencil code: block code for a text world. In: Proceedings of the 14th International Conference on Interaction Design and Children. ACM, pp. 445–448.

Bell, T., Alexander, J., Freeman, I., Grimley, M., 2009. Computer science unplugged: School students doing real computing without computers. The

New Zealand Journal of Applied Computing and Information Technology 13 (1), 20–29.

Biermann, A. W., Ballard, B. W., Sigmon, A. H., 1983. An experimental study of natural language programming. International journal of man-machine studies 18 (1), 71–87.

Brennan, K., Resnick, M., 2012. New frameworks for studying and assessing the development of computational thinking. In: Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada.

Bundy, A., 2007. Computational thinking is pervasive. Journal of Scientific and Practical Computing 1 (2), 67–69.

Capindale, R. A., Crawford, R. G., 1990. Using a natural language interface with casual users. International Journal of Man-Machine Studies 32 (3), 341–361.

Cooper, S., Dann, W., Pausch, R., 2000. Alice: a 3-d tool for introductory programming concepts. In: Journal of Computing Sciences in Colleges. Vol. 15. Consortium for Computing Sciences in Colleges, pp. 107–116.

Cortina, T. J., 2007. An introduction to computer science for non-majors using principles of computation. In: ACM SIGCSE Bulletin. Vol. 39. ACM, pp. 218–222.

Department for Education, 2013. National curriculum in england: Computing programmes of study. Tech. rep., Department for Education, London. URL `https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study`

Dijkstra, E. W., 1976. On the teaching of programming, i.e. on the teaching of thinking. In: Language hierarchies and interfaces. Springer, pp. 1–10.

Galotti, K. M., Ganong, W. F., 1985. What non-programmers know about programming: Natural language procedure specification. International Journal of Man-Machine Studies 22 (1), 1–10.

Gilmore, D. J., Green, T. R. G., 1984. Comprehension and recall of miniature programs. International Journal of Man-Machine Studies 21 (1), 31–48.

Good, J., Howland, K., 2015. Natural language and programming: Designing effective environments for novices. In: Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on. IEEE, pp. 225–233.

Good, J., Howland, K., Nicholson, K., 2010. Young people's descriptions of computational rules in role-playing games: An empirical study. In: Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on. IEEE, pp. 67–74.

Good, J., Oberlander, J., 2002. Verbal effects of visual programs: Information type, structure and error in program summaries. Document Design 3 (2), 120–134.

Good, J., Robertson, J., 2006. Learning and motivational affordances in narrative-based game authoring. In: the Proceedings of the 4th International Conference for Narrative and Interactive Learning Environments (NILE), Edinburgh. pp. 37–51.

Gray, J., Abelson, H., Wolber, D., Friend, M., 2012. Teaching cs principles with app inventor. In: Proceedings of the 50th Annual Southeast Regional Conference. ACM, pp. 405–406.

Green, T. R., Petre, M., Bellamy, R., 1991. Comprehensibility of visual and textual programs: A test of superlativism against thematch-mismatchconjecture. ESP 91 (743), 121–146.

Grover, S., Pea, R., 2013. Computational thinking in k–12: A review of the state of the field. Educational Researcher 42 (1), 38–43.

Guo, P. J., White, J., Zanelatto, R., 2015. Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing. IEEE.

Guzdial, M., 2008. Education: Paving the way for computational thinking. Communications of the ACM 51 (8), 25–27.

Hambrusch, S., Hoffmann, C., Korb, J. T., Haugan, M., Hosking, A. L., 2009. A multidisciplinary approach towards computational thinking for science majors. In: ACM SIGCSE Bulletin. Vol. 41. ACM, pp. 183–187.

Howland, K., Good, J., 2015. Learning to communicate computationally with flip: A bi-modal programming language for game creation. Computers & Education 80, 224–240.

Howland, K., Good, J., Robertson, J., 2007. A learner-centred design approach to developing a visual language for interactive storytelling. In: Proceedings of the 6th international conference on Interaction Design and Children. ACM, pp. 45–52.

Hu, C., 2011. Computational thinking: What it might mean and what we might do about it. In: Proceedings of the 16th annual joint conference on Innovation and technology in computer science education. ACM, pp. 223–227.

Kelleher, C., Pausch, R., Jun. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. ACM Computing Surveys 37 (2), 83–137.
URL `http://doi.acm.org/10.1145/1089733.1089734`

Lieberman, H., Ahmad, M., 2010. Knowing what you're talking about: Natural language programming of a multi-player online game. In: Cypher, A., Dontcheva, M., Lau, T., Nichols, J. (Eds.), No Code Required: Giving Users Tools to Transform the Web. Morgan Kaufmann, Ch. 17, pp. 331–344.

Liu, H., Lieberman, H., 2005a. Metafor: visualizing stories as code. In: Proceedings of the 10th International Conference on Intelligent User interfaces. ACM, pp. 305–307.

Liu, H., Lieberman, H., 2005b. Programmatic semantics for natural language interfaces. In: CHI'05 Extended Abstracts on Human Factors in Computing Systems. ACM, pp. 1597–1600.

Lu, J. J., Fletcher, G. H., 2009. Thinking about computational thinking. In: ACM SIGCSE Bulletin. Vol. 41. ACM, pp. 260–264.

Miller, L. A., 1975. Naive programmer problems with specification of transfer-of-control. In: Proceedings of the May 19-22, 1975, National Computer Conference and Exposition. ACM, pp. 657–663.

Miller, L. A., 1978. Behavioral studies of the programming process. Tech. rep., IBM Thomas J Watson Research Center, Yorktown Heights, NY.

Miller, L. A., 1981. Natural language programming: Styles, strategies, and contrasts. IBM Systems Journal 20 (2), 184–215.

Moreno-León, J., Robles, G., 2015. Dr. scratch: a web tool to automatically evaluate scratch projects. In: Proceedings of the Workshop in Primary and Secondary Computing Education on ZZZ. ACM, pp. 132–133.

Nardi, B. A., 1993. A small matter of programming: perspectives on end user computing. MIT press.

Nelson, G., 2006. Natural language, semantic analysis, and interactive fiction. IF Theory Reader, 141.

Nelson, G., 2011. Natural language, semantic analysis, and interactive fiction. In: Jackson-Mead, K., Wheeler, J. R. (Eds.), IF Theory Reader. Transcript On Press, pp. 141–188.

Pane, J. F., Ratanamahatana, C. A., Myers, B. A., 2001. Studying the language and structure in non-programmers' solutions to programming problems. International Journal of Human-Computer Studies 54 (2), 237–264.

Pea, R. D., 1986. Language-independent conceptual "bugs" in novice programming. Journal of Educational Computing Research 2 (1), 25–36.

Price, T. W., Barnes, T., 2015. Comparing textual and block interfaces in a novice programming environment. In: Proceedings of the eleventh annual International Conference on International Computing Education Research. ACM, pp. 91–99.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., et al., 2009. Scratch: programming for all. Communications of the ACM 52 (11), 60–67.

Robertson, J., Good, J., 2005a. Childrens narrative development through computer game authoring. TechTrends 49 (5), 43–59.

Robertson, J., Good, J., 2005b. Story creation in virtual game worlds. Communications of the ACM 48 (1), 61–65.

Sammet, J. E., 1966. The use of English as a programming language. Communications of the ACM 9 (3), 228–230.

Selby, C., Woollard, J., December 2014. Refining an understanding of computational thinking. Author's original, 1–23.
URL `http://eprints.soton.ac.uk/372410/`

Stead, A., Blackwell, A. F., 2014. Learning syntax as notational expertise when using drawbridge. In: du Boulay, B., Good, J. (Eds.), Proceedings of the Psychology of Programming Interest Group Annual Conference 2014. pp. 41–52.

Stenning, K., Oberlander, J., 1995. A cognitive theory of graphical and linguistic reasoning: Logic and implementation. Cognitive science 19 (1), 97–140.

Tenenberg, J., McCartney, R., 2014. Editorial: Computing education in (k-12) schools from a cross-national perspective. ACM Transactions on Computing Education (TOCE) 14 (2), 6.

Weintrop, D., Wilensky, U., 2015a. 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. ACM, pp. 199–208.

Weintrop, D., Wilensky, U., 2015b. 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. Vol. 15. pp. 101–110.

Whitley, K. N., 1997. Visual programming languages and the empirical evidence for and against. Journal of Visual Languages & Computing 8 (1), 109–142.

Wing, J., 2006. Computational Thinking. Communications of the ACM 49 (3), 33–35.

Wohl, B., Porter, B., Clinch, S., 2015. Teaching computer science to 5-7 year-olds: An initial study with scratch, cubelets and unplugged computing. In: Proceedings of the Workshop in Primary and Secondary Computing Education on ZZZ. ACM, pp. 55–60.

Yadav, A., Zhou, N., Mayfield, C., Hambrusch, S., Korb, J. T., 2011. Introducing computational thinking in education courses. In: Proceedings of the 42nd ACM technical symposium on Computer science education. ACM, pp. 465–470.

Zhong, B., Wang, Q., Chen, J., Li, Y., 2015. An exploration of three-dimensional integrated assessment for computational thinking. Journal of Educational Computing Research, 0735633115608444.