Should We Leverage Natural-Language Knowledge? An Analysis of User Errors in a Natural-Language-Style Programming Language

Amy Bruckman and **Elizabeth Edwards**
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280 USA
asb@cc.gatech.edu, lizzie@cc.gatech.edu

ABSTRACT

Should programming languages use natural-language-like syntax? Under what circumstances? What sorts of errors do novice programmers make? Does using a natural-language-like programming language lead to user errors? In this study, we read the entire online interactions of sixteen children who issued a total of 35,047 commands on MOOSE Crossing, an educational MUD for children. We counted and categorized the errors made. A total of 2,970 errors were observed. We define "natural-language errors" as those errors in which the user failed to distinguish between English and code, issuing an incorrect command that was more English-like than the correct one. A total of 314 natural-language errors were observed. In most of those errors, the child was able to correct the problem either easily (41.1% of the time) or with some effort (20.7%). Natural-language errors were divided into five categories. In order from most to least frequent, they are: syntax errors, guessing a command name by supplying an arbitrary English word, literal interpretation of metaphor, assuming the system is keeping more state information than is actually the case, and errors of operator precedence and combination. We believe that these error rates are within acceptable limits, and conclude that leveraging users' natural-language knowledge is for many applications an effective strategy for designing end-user-programming languages.

Keywords

Natural language, novice programming, programming language design, end-user programming.

A HISTORICAL PERSPECTIVE

Since the very beginning of computing, the use of natural-language-like syntax for programming languages has been controversial. In fact, the use of words of any kind was initially hotly debated. Admiral Grace Murray Hopper, speaking at the history of programming languages conference in 1978, told this story:

> "In the early years of programming languages, the most frequent phrase we heard was that the only way to program a computer was in octal. Of course a few years later a few people admitted that maybe you could use assembly language. But the entire establishment was firmly convinced that the only way to write an efficient program was in octal. They totally forgot what happened to me when I joined Eckert-Mauchly. They were building BINAC, a binary computer. We programmed it in octal. Thinking I was still a mathematician, I taught myself to add, subtract, multiply, and even divide in octal. I was really good, until the end of the month, and then my checkbook didn't balance! [Laughter] It stayed out of balance for three months until I got hold of my brother who's a banker. After several evenings of work he informed me that at intervals I had subtracted in octal. And I faced the major problem of living in two different worlds. That may have been one of the things that sent me to get rid of octal as far as possible." [1]

A somewhat puritanical spirit pervaded the early days of computing. Computers were astronomically expensive, and many argued that their resources shouldn't be squandered to cater to the weakness of human programmers. If coding in octal was time-consuming or error-prone, the coders were simply not working hard enough. It took time to recognize that those delays and errors are inevitable, and better accommodating the needs of the human programmer is not indulgent coddling but simply good business sense. Today, computers are no longer so expensive, but elements of the underlying attitude remain: technologies that are too user-friendly are often denigrated as "not serious."

In 1959, a committee with representatives from industry and government was formed to design a "Common Business Language"—what eventually became COBOL. At one of their first meetings, they made a list of desired characteristics of the new language. It began with these two points:

> "a) Majority of the group favored maximum use of simple English language; even though some participants suggested there might be advantage from using mathematical symbolism.

CHI '99 Pittsburgh PA USA

b) A minority suggested that we steer away from problem-oriented language because English language is not a panacea as it cannot be manipulated as algebraic expressions can." [2]

As these early observations indicate, how natural-language-like to make a programming language is a matter of trade-offs. The COBOL committee was concerned primarily with *manipulability*—in other words, expressive power for mathematical applications. A second common concern is *ambiguity*: words may mean something different in typical English usage than in a program [3, 4]. Another key issue and the primary concern of this paper is *learnability* and the slippery slope of natural language: will novice programmers be able to draw a distinction between English and code? Will they try to insert arbitrary English sentences into programs?

More than twenty-five years after the design of COBOL, the designers of Hypertalk had similar goals and strategies. When asked about the language ancestors of Hypertalk, designer Bill Atkinson replied "The first one is English. I really tried to make it English-like" [5]. Ted Kaehler, another member of the Hypertalk design team, comments that "One principle was 'reads as English, but does not write as English.' Like an ordinary programming language, it depends on exactly the right syntax and terms" [6]. English-like scripting languages are becoming more common, but few empirical studies have addressed the pros and cons of this design approach.

A NATURAL-LANGUAGE-STYLE LANGUAGE

Work on the MOOSE programming language began in mid-1993, and it has been in public use since October 1995. The language was designed for one restricted application: for children to create places, creatures, and other objects that have behaviors in a text-based multi-user virtual world (or "MUD"[1]). The fundamental goal is for children to learn reading, writing, and computer programming through the process of creating such objects [13]. This is an unusual design goal: the process of programming and what is learned from that process is more important than the product (the program created).

The design of MOOSE borrows liberally from the MOO language (on top of which it is built [14-16]) and from Hypertalk. Another significant influence is Logo, the first programming language designed explicitly for kids [17]. The designers[2] deliberately tried to make the MOOSE language as natural-language-like as possible while maintaining a regular syntax. While some researchers are investigating the use of free-form natural language [18], we felt that a natural-language-like approach which still maintained a degree of formal syntax was a more promising compromise. The following is a MOOSE program written by Wendy (girl, age 10-12)[3], one of the randomly selected subjects of this study. The program choreographs a sequence of events as a magical book is opened:

```
on read blue book
  tell context "You take an old and musty blue
    book off of the shelf. As you blow the dust
    off the cover, a symbol painted in gold
    appears. It resembles a circle with a ~ in
    the middle" + "."
  announce_all_but context context's name +
    "carefully takes an old,large,and musty
    blue volume off of the shelf" + "." +
    context's psc + " blows gently. The dust
    swirls up in a flurry of gray mysts[4]. A
    symbol imprinted in gold on the cover
    emerges. It resembles a circle with a ~ in
    the middle" + "..."
  fork 5
    tell context "You hesitantly open this
      strange book. As you peer inside, you see
      a life like painting of a brook behind a
      poppy field and infront of an apple
      orchard...."
    announce_all_but context context's psc + "
      hesitantly opens the strange book."
    fork 15
      announce "A strong wind blows in from the
        open windows. It grows stronger and
        stronger and STRONGER....."
      fork 5
        announce_all_but context context's name
          + "Is suddenly lifted up into the
          air, and carried off...."
        tell context "You are lifted off your
          feet and are carried off...Up over
          the trees, houses, lakes,
          meadows...."
        fork 3
          move player to #4551
        endfork
      endfork
    endfork
  endfork
end
```

[1] "MUD" stands for "Multi-User Dungeon." The first MUDs were violent adventure games [7]. More recently, the technology has been adapted for a variety of purposes including professional communities [8-10] and educational applications [11, 12].

[2] The MOOSE language was designed by Amy Bruckman with guidance from Pavel Curtis, Mitchel Resnick, and Brian Silverman, and assistance from MIT students Austina DeBonte, Albert Lin, and Trevor Stricker.

[3] The children's online pseudonyms have been changed to protect their identities.

[4] Children's spelling and grammar have been left unchanged.

When you run this program by typing "read blue book," you are magically transported to a babbling brook. This is what you see:

```
You take an old and musty blue book off of the
shelf. As you blow the dust off the cover, a
symbol painted in gold appears. It resembles a
circle with a ~ in the middle.

[pause]

You hesitantly open this strange book. As you
peer inside, you see a life like painting of a
brook behind a poppy field and infront of an
apple orchard....

[pause]

A strong wind blows in from the open windows.
It grows stronger and stronger and
STRONGER.....

You are lifted off your feet and are carried
off...Up over the trees, houses, lakes,
meadows....

[pause]

Babbling Brook

You are in a small meadow filled with poppies.
As the breeze frolicks above the flowers, the
dance and sway like the sea. Behind you is a
forest of apple trees, pear trees, orange
trees, and peach trees.Underneath them is a
carpet of green green moss, soft and springy.
Beside you is a babbling brook which giggles
and laughs as it slides down over the sMOOth
pebbles. As you stick your foot in you are
suprised. This stream is not cold like all the
others, but warm, and soothing. Tiny mare's
tails walts across the sky.Can this last
forever?  It is late-afternoon summer. A
bright sunny day with few clouds.
```

The syntax of a basic MOOSE command is a verb followed by some number of arguments. Arguments can be string constants, numbers, or references to objects. Quoting of strings is optional as long as those strings don't contain words that function as logical operators (such as "and"). The environment includes both a command-line language and scripting language, which were designed to be as nearly identical as possible. This allows the learner to try most commands out at the command line, and later use them in programs. A more complete description of the language and principles that underlie its design appears in [13].

The language was designed with eight basic heuristics:

1. Have a gently-sloping learning curve.
2. Prefer intuitive simplicity over formal elegance.
3. Be forgiving.
4. Leverage natural-language knowledge.
5. Avoid non-alphanumeric characters wherever possible.
6. Make essential information visible and easily changeable.
7. It's OK to have limited functionality.
8. Hide nasty things under the bed. [13]

Are these heuristics useful? Under what circumstances? Of particular interest is rule four, "Leverage natural-language knowledge." The designers felt that a natural-language-like programming language would increase accessibility to young children. However, we worried about the slippery slope of natural language: would children understand the differences between MOOSE and English? This paper attempts to address that question systematically.

NAME	AGE	TIME OF ACTIVE MEMBERSHIP	COMMANDS ISSUED	OBJECTS CREATED	SCRIPTS PROGRAMMED	SCRIPTING LEVEL ACHIEVED
Percy	6	7 min.	15	1	0	
Jessica	7	3 days	79	1	0	
Wowzers	8-9	7 mo.	190	11	1	Basic
Snickers	9-10	15 days	2398	31	12	Basic
Hope	10	2 days	145	5	0	
Liono	10	2 mo.	217	4	0	
Reebok	10-11	6 mo.	1379	17	10	Basic
Wendy	10-12	21 mo.	15719	56	49	Advanced
Sheriff	11-12	2 mo.	609	17	0	
Mike	11-15	33 mo.	40182 (1275)	129	234	Expert
Oracle	12	7 days	467	1	2	Basic
Darcy	13	3 days	158	7	0	
Altair	15	49 min.	68	1	0	
Lucy	15	2.5 mo.	70	2	2	Basic +
Pedro	15-16	6 mo.	3617	17	21	Advanced
Sven	16	1 mo.	8641	21	12	Intermediate

Table 1: Randomly selected study subjects

Note that this paper addresses the risks and possible downsides of natural-language-style programming, but not the benefits. Three years of observation of children using the MOOSE language in the virtual world called MOOSE Crossing have led us subjectively to believe that it has significant benefits. Children as young as seven have been able to program in MOOSE. Kids can immediately read other children's programs and use them as examples to learn from. The intuition that reliance on natural language is part of what makes this possible is based on years of participant observation, clinical interviews, and log file analysis. A systematic analysis of the benefits of natural-language-style programming would be desirable. However, that is beyond the scope of this study, and is left for future work. In this study, we attempt to examine the downside risks systematically.

THE STUDY

At the time of this writing, the MOOSE language has been used for almost three years by 299 children and 211 adults. All input to and output from the system is recorded, with written informed consent from both parents and children. A total of 1.1 Gb of data has been recorded as of July 31st, 1998.[5] To re-evaluate the language's design and principles underlying it, we randomly selected 16 children, and categorized every error each child made. While this retrospective analysis is not a controlled study, the data is intriguing and we believe sheds light on general questions of programming language design for children.

Data about the random sample of children appears in Table 1. The children range in age from six to fifteen at the start of their participation. Their length of involvement ranges from seven minutes to thirty-three months. The total number of commands they typed into the system (which ranges from 15 to 40,182) is perhaps a better measure of their varying degrees of involvement. Seven of the children wrote no programs; five attained basic or slightly above basic programming knowledge; one, intermediate knowledge; two, advanced knowledge; one, expert knowledge. Definitions of coding categories are:

Basic	Simple output.
Intermediate	Conditionals, property references, variables.
Advanced	List manipulation, control flow.
Expert	Complex projects using all language features and constructs.

The children's level of achievement is based on what language constructs they were able to use independently in original programs. For example, Snickers has a number of programs with intermediate language constructs; however, he received significant assistance in writing that code and never demonstrated that he understood everything he was shown. Consequently, he is listed in the Basic category.

For each child, Elizabeth Edwards read the child's entire online experiences, and categorized each error the child made. (With one significant exception: Mike's degree of participation was so high that it was logistically impossible for us to read his entire log file. Instead, we sampled his participation by randomly selecting one month per year for a total of 1,275 of his 40,182 commands typed.) "Errors" most typically are times when the system returned an error message; however, we also subjectively inferred situations in which the output from the system was likely not what the child desired. For example, Wendy typed:

```
describe here as the way it was before!!
```

We can reasonably infer that the outcome (the room was described literally as "the way it was before!!") was not what she intended.

NUMBER OF ERRORS	NUMBER OF COMMANDS TYPED	ERROR RATE (ERRORS/TOTAL COMMANDS)
2970	35047	8.5%

Table 2: Over-all error rate observed

KID NAME	ERRORS	PERCENT NATURAL LANGUAGE (NAT.-LANG. ERRORS/TOTAL ERRORS)	SCRIPT LEVEL
Percy	1	0.0% (0)	
Jessica	3	0.0% (0)	
Wowzers	22	9.1% (2)	Basic
Snickers	394	15.2% (60)	Basic
Hope	29	10.3% (3)	
Liono	67	7.5% (5)	
Reebok	148	7.4% (11)	Basic
Wendy	975	11.3% (110)	Advanced
Sheriff	134	12.7% (17)	
Mike	155	12.3% (19)	Expert
Oracle	27	3.7% (1)	Basic
Darcy	45	6.7% (3)	
Altair	3	33.3% (1)	
Lucy	12	0.0% (0)	Basic +
Pedro	432	13.7% (59)	Advanced
Sven	523	4.4% (23)	Intermediate

Table 3: Errors for each child

[5] Most data for one roughly six-month period (6/10/97-12/1/97 was lost due to a technical problem. Most of Lucy's participation was during this time. The other subjects are less directly affected.

For the sixteen children, a total of 2,970 errors were observed (see Table 2). They are broken down per child in Table 3. There is no apparent correlation between the child's age or level of programming achievement and the number of natural language or other errors they make.

Errors are divided into seven basic categories (see Table 4). From most to least frequent, they are: object manipulation, command-line syntax, typos, scripting syntax, movement, system bugs, and communication/interaction errors. A more detailed breakdown appears in Table 5.

Interaction in the virtual world takes place at the interactive command-line prompt. Scripts are written in a separate window, in a client program (MacMOOSE or JavaMOOSE) designed to give the child a supportive programming environment. Clicking "save" in the client compiles the script and returns feedback to the user. Note that command-line errors are counted per individual line typed; however, scripting errors are counted *per compile*.

In each of these error categories, some errors can be categorized as natural-language errors, and some can not. Examples appear in Table 6. Generally speaking, we define natural-language errors as those errors in which the incorrect command is more English-like than the correct.

In total, 10.6% of errors found were judged to be natural-language related. A total of 314 natural-language errors were found. Of those, 73% (229/314) were command-line syntax errors. In most cases, such errors involve a child guessing at a command's name or the syntax of its arguments. The "examine" command will tell you what commands are available for a particular object and what their exact syntax is; however, children frequently guess rather than use "examine."

In a study of novice Pascal programmers, Jeffrey Bonar and Elliot Soloway found error rates attributable to "step by step natural-language knowledge" from between 47% to 67% [19]. Certainly the measures used in the two studies are not directly comparable, and the definitions of "natural-language errors" differ. However, if it were the case, broadly speaking, that natural-language errors were less common in MOOSE than Pascal, this finding wouldn't be surprising. In an English-like language such as MOOSE, relying on natural-language knowledge is often a successful strategy. In a more formal language like Pascal, this approach is more likely to lead to errors.

TYPE OF ERROR	NUMBER OF ERRORS	PERCENT NATURAL LANGUAGE (NAT.-LANG. ERRORS/TOTAL ERRORS)
Object manipulation	799	7.6% (61/799)
Command-line syntax	701	32.7% (229/701)
Typos	701	0% (0/701)
Scripting syntax	352	2.0% (7/352)
Movement	318	5.3% (17/318)
System bugs	54	0% (0/54)
Communication and Interaction errors	45	0% (0/45)
TOTAL	2970	10.6% (314/2970)

Table 4: Categorization of errors

ERROR	DETAILED BREAKDOWN
Object manipulation	Assuming presence of object that doesn't exist (243) Assuming script that doesn't exist (240) Incorrect number of arguments (128) Trying to run script that never compiled (98) Ambiguous object reference (35) Permissions errors (31) Wrong type of argument (24)
Command-line syntax	Syntax errors (336) Guessing at commands (263) Errors creating objects (67) Difficulties with tutorial system (26) Disallowed characters in object names (9)
Typos	Misspellings (440) Forgotten "say" or "emote" (174) Key banging (87)
Scripting syntax	Quoting errors (117) Scripting syntax errors (111) Mismatch of script name (38) Nonexistent property or variable (38) Missing script structure ("on", "end", returns) (28) Problems with alternate line editor (7)
Movement	Assuming exit which doesn't exist (201) Teleporting to random non-existent room name (64) Type room name instead of exit name (53)
System bugs	Mail system problems (39) Other system bugs (15)
Communication and interaction errors	Saying something instead of doing it (15) Typing desired output instead of command to generate desired output (12) Talking to non-player characters (9) Talking to person not in the room (7) Addressing person by real rather than character name (2)

Table 5: Detailed error breakdown

Roy Pea comments:

> "[Students'] default strategy *for making sense* when encountering difficulties of program interpretation or when writing programs is to resort to the powerful analogy of natural language conversation, to assume a disambiguating mind which can understand. It is not clear at the current time whether this strategy is consciously pursued by students, or whether it is a tacit overgeneralization of conversational principles to computer programming "discourse." The central point is that this personal analogy should be seen as *expected* rather than bizarre behavior, for the students have no other analog, no other procedural device than "person" to which they can give written instructions that are then followed. Rumelhart and Norman have similarly emphasized the critical role of analogies in early learning of a domain—making links between the to-be-learned domain and known domains perceived by the student to be relevant. But, in this case, mapping conventions for natural language instruction onto programming results in error-ridden performances." [20]

Pea's conclusions are based on his analysis of student errors in traditional programming languages. One approach to countering this problem is deliberately to leverage students' natural-language knowledge in the programming-language design.

Table 7 sorts the 314 natural language errors into different categories—categories more descriptive of the nature of natural-language errors we observed. The most common are again syntax and guessing errors. Many of these errors demonstrate a lack of understanding of underlying computer-science concepts. In the first example, Wendy apparently wants to make her pet follow her around the virtual world. She expresses that in an English-like fashion ("set Roo to follow me"). However, she evidently fails to understand that making a pet follow you involves setting a property on the pet's object (the correct command would be "set Roo's following to me".) Wendy demonstrates an understanding of the use of properties in other contexts, but not in this instance.

Perhaps the most intriguing category of error is literal interpretation of metaphor. For example, to get rid of an object that you no longer want, you "recycle" it. Recycling is a metaphor for a process that can be more precisely described as deleting a database entry. Interpreting that metaphor somewhat literally, a number of participants have tried to "reuse" objects.

The next most prevalent category is assuming the system tracking or aware of state more than it is. When travelling through the virtual world, children will often type "back" to try to retrace their steps. No such command exists, (though implementing one is not hard and actually might be a good idea.)

OBJECT MANIPULATION:	
NON-NL	`set Rocky's following 1` (Correct command would be: set Rocky's following to 1)
NL	`feel Napoleon`
COMMAND-LINE SYNTAX:	
NON-NL	`create #100 josephine` (Correct command would be: created #100 named josephine)
NL	`examine me more`
SCRIPTING SYNTAX:	
NON-NL	Missing end, endif, etc.
NL	`if number < 20 and > 10`
MOVEMENT:	
NON-NL	Trying to use exit that doesn't exist.

NL	`Back`
	`Go to tree house`
	(There are no such commands.)

Table 6: Examples of non-natural language (Non-NL) and natural language (NL) errors

TYPE	INSTANCES (ERRORS OF TYPE/TOTAL ERRORS)	EXAMPLE
Syntax	46.8% (147/314)	`set Roo to follow me` (To make a pet follow you, you need to set its "following" property. Correct command is: set roo's following to me.)

Guessing	21.7% (68/314)	`make new thing`
		(Correct command would be to type "create" and wait for prompts or type "create <parent> named <object name>")
		`tie hair with ribons`
		(Child has created an object called "ribons" but not programmed any scripts on it.)

Literal interpretation of metaphor	18.5% (58/314)	`reuse Harper` (You can "recycle" an object, but not "reuse" it.)
Assuming system is tracking/ aware of state	4.5% (14/314)	`back` `describe here as the way it used to be`
Operator precedence or combination	2.2% (7/314)	`if number < 20 and > 10`

Table 7: Types of natural language errors

Interestingly enough, the least common category is the one we were most worried about before we began data analysis: operator precedence and combination. The conditional clause "if A is B or C" is parsed by the computer as equivalent to "if (A is B) or (C is true)"; however, it's often the case that the user meant "if (A is B) or (A is C)".

Another type of operator error involves the insertion of extra operator words. For example, children often write statements of the form "if x is member of y," inserting an extra "is" before the "member of" operator. This particular problem can be automatically detected and is corrected by the MOOSE compiler. However, the compiler currently is not able to correct the error in the example "if number < 20 and > 10."

Concern about operator errors was the original motivation for undertaking this study. However, only seven of 314 natural language errors and 2,970 total errors fell into this category. It's worth noting that only four of sixteen children demonstrated an understanding of the use of conditionals. Those four children had a total of 2125 errors. Operator precedence and combination errors represent only 0.3% of the total.

ERROR RECOVERY?

But how serious are these natural language errors? Certainly an error that is immediately corrected is quite different from one that causes the child to abandon a project in frustration. We divided error recovery into six categories:

Immediate As soon as feedback is received, the next command directed towards the problem solves it.

Short The problem takes more than one attempt but is solved in that particular sitting.

Long The child doesn't solve the problem in the initial attempt, but returns to it later (time ranging from minutes to days) and solves the problem then.

Workaround Child does not determine how to execute this particular command, but constructs a different string of commands that produce the desired results.

Interrupted Child is interrupted by a message, arrival of another child, parental threat of grounding if they don't get off the computer, etc., and does not appear to return to the problem.

Never Problem not solved.

For each of the 314 natural language errors observed, we categorized the recovery time. This data appears in Table 8. Table 9 analyzes how quickly errors were recovered by type, grouping them into easily recovered (immediate and short), recovered with difficulty (long and workaround), not recovered (never), and unclear (interrupted). Error recovery rates were not calculated for non-natural-language errors. This would be an interesting topic for future work.

At first glance it surprised us that guessing errors were the most "serious"—aren't operator errors, for example, conceptually deeper? However, it's likely that this is simply a reflection of the depth of the child's engagement with the task at hand. A guessing error may often be a whim—if the task isn't easy, it is readily abandoned. An operator error, on the other hand, occurs in the context of

	IMMEDIATE	SHORT	LONG	WORK-AROUND	INTER-RUPTED	NEVER	TOTAL
Syntax	42	27	32	13	2	61	177 (56.4%)
Guessing	10	9	10	2	0	37	68 (21.7%)
Metaphor	21	6	1	1	0	19	48 (15.3%)
State	6	7	0	0	1	0	14 (4.5%)
Operator	1	0	0	6	0	0	7 (2.2%)
Total	80 (25.5%)	49 (15.6%)	43 (13.7%)	22 (7.0%)	3 (1.0%)	117 (37.3%)	314

Table 8: Error recovery times for natural language errors

	EASILY RECOVERED (IMMEDIATE + SHORT)	RECOVERED WITH DIFFICULTY (LONG + WORKAROUND)	NOT RECOVERED (NEVER)	UNCLEAR (INTERRUPTED)
Syntax	39.0% (69/177)	25.4% (45/177)	34.5% (61/177)	1.1% (2/177)
Guessing	27.9% (19/68)	17.6% (12/68)	54.4% (37/68)	0.0% (0/68)
Metaphor	56.3% (27/48)	4.2% (2/48)	39.6% (19/48)	0.0% (0/48)
State	92.9% (13/14)	0.0% (0/14)	0.0% (0/14)	7.1% (1/14)
Operator	14.3% (1/7)	85.7% (6/7)	0.0% (0/7)	0.0% (0/7)
Total	41.1% (129/314)	20.7% (65/314)	37.3% (117/314)	1.0% (3/314)

Table 9: Recoverability of natural language errors

a project in which the child has already invested significant time and effort. Consequently, the child is more likely to spend the time to solve the problem or in most cases find a workaround. It makes sense then too that syntax errors are more likely to be successfully resolved than guessing errors: with a syntax error, the child has found a command and simply needs to learn to use it correctly. In the case of a guess, no such command or concept may exist.

CONCLUSIONS

Is it advisable to "leverage natural-language knowledge" in designing programming languages? The question of course can't be answered in the general case, because different applications and target audiences have different needs. A more focused question might be: is it wise to leverage natural-language knowledge in the design of a programming language for children designed to promote learning? We began in 1993 with the intuition that the answer was "yes." This study supports that conclusion.

This work primarily addresses the risks of natural-language-style programming. A formal analysis of its benefits of would be desirable, but is beyond the scope of this study.

In total, we found that 16 children made a total of 2,970 errors. Of those, 314 were natural-language-related. Most of those errors were easily recovered (41.1%) or recovered with some difficulty (20.7%). Those that were not recovered represent 37.3% of the natural language errors and only 4.2% of total errors. We believe these rates to be within acceptable limits. Leveraging users' natural-language knowledge does not appear to cause serious problems. We believe that making use of people's pre-existing natural language knowledge is an effective strategy for programming language design for children, end users, and others new to coding.

In future work, we hope to continue to analyze this set of data to shed light on other aspects of programming-language design for novice users.

REFERENCES

1. Hopper, G.M., *Keynote Address,* in *History of Programming Languages,* R.L. Wexelblat, Editor. 1981, Academic Press: New York. p. 7-20.
2. Sammet, J., *The Early History of COBOL,* in *History of Programming Languages,* R. Wexelblat, Editor. 1981, Academic Press: New York.
3. Spohrer, J. and E. Soloway, *Analyzing the High Frequency Bugs in Novice Programs,* in *Empirical Studies of Programmers,* E. Soloway and S. Iyengar, Editors. 1986, Ablex Publishing: Norwood, NJ.
4. Boulay, B.D., *Some Difficulties of Learning to Program,* in *Studying the Novice Programmer,* E. Soloway and J.C. Spohrer, Editors. 1989, Lawrence Erlbaum Associates: Hillsdale, NJ. p. 283-299.
5. Goodman, D., *The Complete HyperCard Handbook.* 2nd ed. 1988, New York: Bantam Books.
6. Kaehler, T., 1996, personal communication.
7. Bartle, R., *Interactive Multi-User Computer Games.* 1990, MUSE Ltd: ftp://ftp.lambda.moo.mud.org/pub/MOO/papers/mudreport.txt
8. Bruckman, A. and M. Resnick, *The MediaMOO Project: Constructionism and Professional Community.* Convergence, 1995. **1**(1): p. 94-109.
9. Glusman, G., E. Mercer, and I. Rubin, *Real-time Collaboration On the Internet: BioMOO, the Biologists' Virtual Meeting Place.,* in *Internet for the Molecular Biologist.,* S.R. Swindell, R.R. Miller, andG.S.A. Myers, Editors. 1996, Horizon Scientific Press: Norfolk, UK.
10. Van Buren, D., *et al., The AstroVR Collaboratory,* in *Astronomical Data Analysis Software and Systems IV,* R. Hanish and H. Payne, Editors. 1994, Astronomical Society of the Pacific: San Francisco.
11. O'Day, V., *et al., Moving Practice: From Classrooms to MOO Rooms.* Computer Supported Cooperative Work, 1998. **7**: p. 9-45.
12. Bruckman, A., *Community Support for Constructionist Learning.* Computer Supported Cooperative Work, 1998. **7**: p. 47-86.
13. Bruckman, A., *MOOSE Crossing: Construction, Community, and Learning in a Networked Virtual World for Kids.*1997, MIT, Ph.D. dissertation: http://www.cc.gatech.edu/~asb/thesis/
14. Curtis, P. *Mudding: Social Phenomena in Text-Based Virtual Realities.* in *DIAC.* 1992. Berkeley, CA: ftp://ftp.lambda.moo.mud.org/pub/MOO/papers/DIAC92.txt
15. Curtis, P., *LambdaMOO Programmer's Manual.* 1993: ftp://ftp.lambda.moo.mud.org/pub/MOO/ProgrammersManual.txt
16. Curtis, P. and D. Nichols. *MUDs Grow Up: Social Virtual Reality in the Real World.* in *Third International Conference on Cyberspace.* 1993. Austin, Texas: ftp://ftp.lambda.moo.mud.org/pub/MOO/papers/MUDsGrowUp.txt
17. Papert, S., *Mindstorms: Children, Computers, and Powerful Ideas.* 1980, New York: Basic Books.
18. Miller, L.A., *Natural language programming: Styles, strategies, and constrasts.* IBM Systems Journal, 1981. **20**(2): p. 184-215.
19. Bonar, J. and E. Soloway, *Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers,* in *Studying the Novice Programmer,* E. Soloway and J. Spohrer, Editors. 1989, Lawrence Erlbaum Associates: Hillsdale, NJ.
20. Pea, R.D., *Language-Independent Conceptual "Bugs" in Novice Programming.* Journal of Educational Computing Research, 1986. **2**(1): p. 25-36.