Natural Language in Programming

An English Syntax-based Approach for Reducing the Difficulty of First Programming Language Acquisition

Master's Thesis

Presented to

The Faculty of the Graduate School of Arts and Sciences
Brandeis University
Department of Computer Science
James Pustejovsky, Advisor

In Partial Fulfillment
of the Requirements for

Master's Degree

by

Andrew Riker

May 2010

ABSTRACT

Natural Language in Programming: An English Syntax-based Approach for Reducing the Difficulty of First Programming Language Acquisition

A thesis presented to the Computer Science Department

Graduate School of Arts and Sciences
Brandeis University
Waltham, Massachusetts

By Andrew Riker

Programming is an intrinsically difficult skill to learn, because first-time programmers are confronted with a large number of new concepts at the same time. This paper explores several other obstructions to first programming language acquisition, proposes modifications for modern programming languages to address them, explores the advantages and disadvantages of natural language programming, and introduces a new natural language programming language designed specifically to address the issues confronting first-time programmers

Table of Contents

Introduction

The process of learning a first programming language is a daunting task, because it requires explicit knowledge of high-level programming theory, syntax, semantics, logic, programming terminology, and problem solving. These issues are not insurmountable, but current programming languages do not provide an optimal platform for first programming language acquisition.

The goal of Board Game Language (BGL) is to introduce new programmers to the concept of formal languages by imitating a constrained portion of the syntax of the English language in a familiar context, board games. This language specification, in conjunction with a tutorial that introduces programming concepts via analogies to everyday life, should ease the burden of the requirements of learning a programming language.

In this paper, I will explore the reasons why current programming languages are suboptimal for first programming language acquisition. I will then propose a number of modifications to current languages that would increase their ease of acquisition. Next I will introduce the specification for BGL and explain how it implements the proposed modifications. The paper will include a discussion of the BGL compiler, which is being implemented in conjunction with the writing of this paper. I will evaluate the BGL implementation by ease of non-programmer comprehension.

Section 1. Learnability Issues

1.1 Introduction

Non-programmers often see programming as a difficult and unnatural task, while experienced programmers tend not to see the same level of difficulty, because they already know the problem solving and debugging techniques, theory, syntax, and semantics of programming languages. The scenario here seems very familiar; the expert finds an activity to be easier than a novice. This much is true, but this does not provide the most accurate picture of learning to program.

To clarify at an intuitive level, take the example of learning the fundamental elements of basketball. In basketball, you learn to dribble, shoot, and pass. If you double-dribble, travel, or perform another violation of the rules, there is no harm done. Someone explains the issue, and you continue with your practice. To make this more like learning to program, we need to make a few adjustments. First, if you violate a rule, your basketball deflates, and you need to refill it before you continue. Second, someone explains to you what you did wrong in a language you do not understand. Clearly learning to play basketball just became harder. I would not be surprised if few people bothered to learn it at all.

While the idea of learning basketball with such constraints is completely ridiculous, it is very much like learning to program. The ball deflating is analogous to a error in a program. This forces the programmer to try to understand what went wrong.

However, the language compiler or people explaining the issue speak a language the programmer does not understand. He or she is left with two options: trial and error to figure out how to program or learn the language through books and instructors. The first option is implausible, so it is always the second.

1.2 Confluence of unfamiliar topics

This analogy begs the question "why is it inherently difficult to learn to program?". A complete answer to this question is not easy either. The simple answer is that there is a lot to learn at once (Robins et al, 2003)(Du Boulay 1980):

1. High-level programming theory
2. New language syntax
3. New language semantics
4. Logic/problem solving theory
5. Debugging
6. Complex terminology

This is a very high-level list of requirements. It is fairly easy to create a much longer list of low-level requirements, but these six requirements cover a large enough area for our purposes. Du Boulay defines the similar terms of general orientation, notational machine, structures, and pragmatics, which loosely map to the first five requirements (Du Boulay, 1980).

Each of these requirements is fairly complex on its own. Consider High-level programming theory. There could be several possible definitions for this, but for this paper it will refer to the concepts relating to the parts of a programming language and how they work together, including but not limited to:

- Variables/Arguments/Parameters
- Functions
- Objects/Types

- Inheritance
- Flow of control
- Abstraction
- Scope

A programmer must learn explicitly each of these topics and how they relate to others. This is a daunting task, one that requires several semesters of classes to master. However, there are five requirements beyond this one.

New language syntax and semantics are separate in the list, because both are complex in their own right, but they are combined in practice. It would make no sense to learn them separately. Many times the reason it is so difficult to grasp, is that the syntax is so different from anything the programmer has experienced previously. Consider the following Python function, which makes a list of the specified number of supplied elements:

```
1 def make_list_of(num,element):
2     if num==None or type(num)!=int: return None
3
4     elements=[]
5     for index in range(num_elements):
6         elements.append(element)
7
8     return elements
```

This is a fairly simple function for experienced programmers to understand, but someone just starting their first language (assuming that it is Python) would likely have little clue as to how the function works. The reason for this is not necessarily that the concepts are difficult, but that mapping between the syntax and semantics is unfamiliar. A new programmer would immediately face the following questions:

	Questions	**Answers**
1.	What does “def” on line 1 mean?	(function definition)
2.	Why are there parentheses on lines 1, 2, 5, and 6?	(function parameters)

3. What are inside the parentheses on lines 1, 2, 5, and 6? (function call)
4. What do "==" and "!=" on line 2 mean? (equality comparison)
5. What does "return" do on line 2? (value to prev. scope)
6. How is "==" on line 2 different from "=" on line 4? (compar. vs. assign.)
7. What is "[]" on line 4? (list declaration)
8. What does "for" mean in this context? (loop)

The new programmer likely has more questions than this, but this is a good place to start. Notice that many of the concepts seen in the answers are not difficult to explain. Consider an equality comparison. This is essentially "is thing x the same as thing y?". The programmer can most likely answer this question about real-world objects quite easily. The issue is how the problem is presented syntactically. There are some answers, like function call, that are more difficult to explain, because there is no obvious real-world analogue.

For the uninitiated, formalizing a vague problem into a step-by-step process (algorithm) is challenging. It involves fully defining the problem, analyzing the tools available to solve the problem, and writing a fully-specified series of instructions to solve the problem (Skiena, 1998). Without formal training in algorithm construction, it can be difficult to know at exactly which point the problem is fully defined. Actually writing the algorithm involves knowledge of the capabilities of the programming language, which is something learned over time.

Debugging is a generic term for finding and fixing errors in a program (What went wrong? Finding and Fixing Errors through Debugging). These errors can be syntactic, which are normally detected by the compiler or interpreter, or logical, which cause unwanted behaviors and can be very difficult to detect (Halpern, 1966). The former are hardest to detect for novice programmers and become easier as the

programmer gains experience with the language's syntax. The latter, however, can be extraordinarily difficult to find. It involves knowing exactly what each line in the program does. If what the programmer believes a statement to do an what it actually does are disjointed, there is the potential for catastrophe. On more than one occasion I have found it easier to rewrite part of an application than finding particularly elusive logic errors. Since novice programmers are, by definition, just starting to program, the chances of them misunderstanding a statement are very high, which makes isolating the problematic statement difficult.

Terminology is related to each of the other requirements. In order to understand the language, the programmer must first understand the words that describe the language. This is perhaps the least programming-specific requirement, because almost every subject has some terminology that a newcomer will need to internalize. In programming, like in other complex subjects, there is so much terminology that it takes more than one semester to define enough of the basics to allow the programmer to do something useful.

Each of these requirements is complex, but they must be learned at the same time as each of the others. This confluence of new material makes the task of the novice programmer arduous and frustrating. Dropout rates for introductory programming classes are quite high. Päivi Kinnunen and Lauri Malmi report dropout rates between 30 and 50 percent for Helsinki University of Technology and between 20 and 40 percent for other institutions (Kinunnen & Malmi, 2006). Azad Ali and Fredrick Kohun found a significantly more alarming percentage of 25 to 80 percent (Ali & Kohun, 2009). Clearly many would-be programmers never finish learning that first language due to this additive

learning curve.

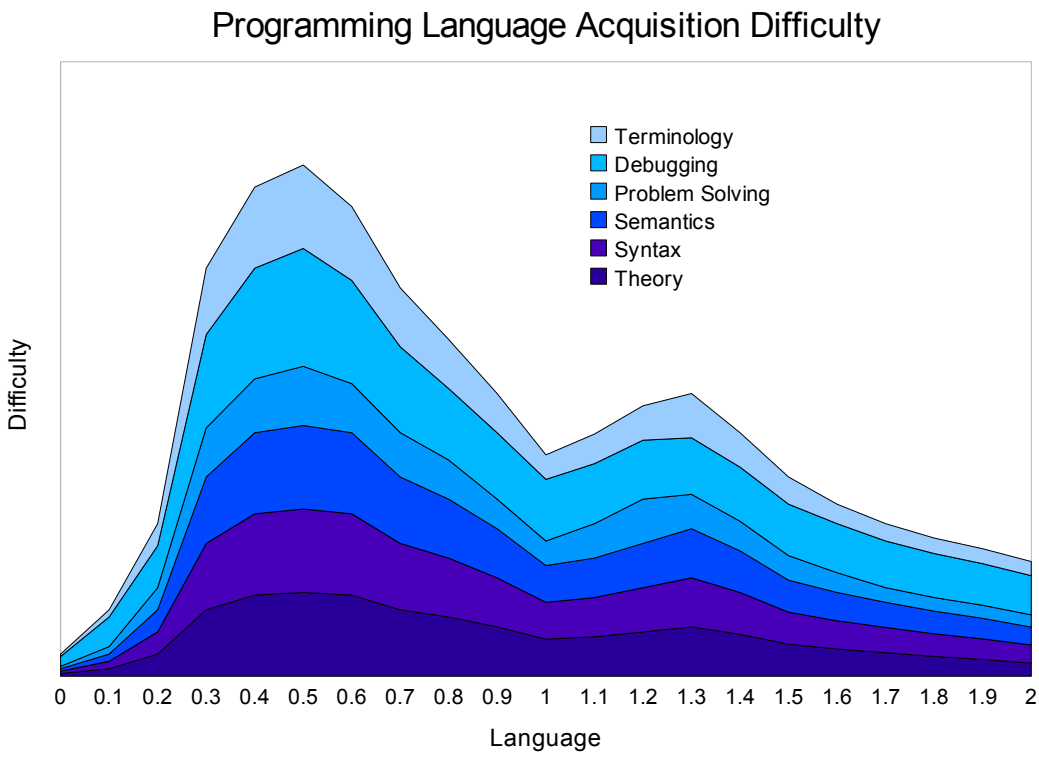


Illustration 1.1:Theorized representation of programming language acquisition for 2 languages

As illustrated in the theorized learning curve, the difficulty of programming increases very quickly as the six requirements combine. The graph starts at low difficulty, because the programmer starts by learning conceptually simple things like variables, types, and inheritance (in object-oriented programing). Soon, however, the programmer must learn how to manipulate these variables, understand the relationship between variables and functions, and understand how order affects evaluation (flow-of-control). These issues are theoretically more complex, so they require more knowledge of syntax and semantics. In order to address the the increased theoretical complexity efficiently, the language must introduce more terminology. As the programmer learns more of the language, the

complexity of what the programmer can express increases, resulting in a need for problem solving strategies.

By the time the programmer is proficient in his or her first language, he or she must have learned the syntax and resulting semantics (not necessarily all the language's libraries), so he or she must understand the theory behind them. The proficient programmer will have written several complete applications in the language, so he or she must have created plans for creating the application (problem solving). The expert programmer will even have learned strategies for problem solving (Robins et al, 2003). The second language becomes much easier, because the programmer has an understanding of what each requirement is. It should be noted, however, that debugging is a difficult task regardless of experience.

Helping the first-time programmer to learn his or her first programming language by reducing the learning curve is the goal of this thesis.

1.3 Tutorial Presentation

Programming languages are typically presented to the programmer in three separate parts: the compiler or evaluator, the API, and instructions of how to create applications with the compiler or evaluator. The instructions typically tie the compiler or evaluator together with the API. Some languages, like Python and Java, provide official tutorials to help programmers understand the language by working through examples (The Java Tutorials)(Python Tutorial, 2009). Every author has his or her own style for presenting the material, but in all the programming tutorials I have read, it seems that each part of the tutorial is completely separate from the others. This separation makes it

difficult to remember the lessons from previous sections of the tutorial.

1.4 Picking a First Language

A critical factor in learnability is which language the would-be programmer decides to try. Some languages are inherently easier to learn than others. There are various factors that contribute to this, including but not limited to: automatic garbage collection, automatic pointer control, and mechanisms for accomplishing frequent tasks. Since some programming languages are easier to learn, choosing one that is more difficult clearly negatively affects the general programming language acquisition curve.

This decision is complicated by the fact that every person the would-be programmer inquires about the best first programming language will invariably tell him or her something different. The answer may be Python, Java, JavaScript, Pascal, Basic, Logo, or any number of other languages that the more experienced programmer considers easy.

There is a wealth of information on picking a first programming language or the constraints a first programming language must satisfy. Michael Schneider provides ten principles on introductory programming courses. The particularly relevant principles include choosing a language simple, but rich in control structures (conditionals, loops), data structures, and subprogram structures (functions, variables) (Schneider, 1978).

K. N. King claims that a year of a programming language that has the syntax of the target programming language, but removes the advanced features to remove some of the difficulty of learning it (King, 1997). He goes on to claim that Java is a good vehicle for those whose ultimate goal is to learn C++, but he does not mention how Java would

help in the move to other programming languages.

Diwaker Gupta wrote an article for ACM on the subject of choosing a first language. He claims that designers of first programming languages should strive to achieve simplicity, consistency, feature set orthogonality, common construct coverage, and helpful debugging (Gupta, 2004). He further claims that these languages ought to avoid excess brevity and verbosity in syntactic constructions, because the confuse first-time programmers and detract from the learning process. Java, with its mandatory class structures violates the excess verbosity condition. Likely his most important observation is that those who develop programming languages are not doing so with the pedagogical requirements of first-time programmers in mind.

Therefore, there should be one language whose target audience is only beginners (Robins et al, 2003) . There have been some attempts at these type of languages. Seymour Papert and Wally Feurzeig developed the LOGO programming language to make programming language suitable for children (Papert, 1997). It has seen a significant impact for some of the teachers who introduced it to their students. The students developed an enthusiasm for programming, especially when LOGO was combined with a LEGO computer system, because they had more opportunities to relate to the concepts. Section 4 introduces a new language, which bridges a separate set of concepts for a more mature target audience.

1.5 Target Domain

With general-purpose languages, it can be very difficult for the first-time programmer to decide on his or her first large project. By virtue of being a general-

purpose language, the programmer should be able to apply the language to almost any problem that can be specified as an algorithm. It is simply a case of having too few constraints to be able to narrow the field of project possibilities. The other issue is that the would-be programmer might not have experience in the domain that a limited-purpose language requires. What the novice programmer needs is a limited-purpose language in a domain in which he or she has experience.

1.6 Inadequate Model of Semantic Relatedness

Humans tend to retask the same verbs with different arguments when speaking about semantically similar events. Instead of changing the verb, humans will change the types and number of the arguments (Levin, 1993). The result of such an operation is called a diathesis alternation or verb alternation, which simply means that a verb has multiple possible subcategorization frames (Barnett et al, 1996). The subcategorization frame is a linguistic construct that allows a verb to specify number, order, and types of arguments (The XTAG Research Group).

The issue is that if one takes imperative verbs to be analogous to functions in programming, current programming languages do not provide enough semantic precision to specify semantically related but distinct functions. The subcategorization frame dictates the meaning of the function. Adding or removing an argument can drastically affect what the function needs to do to fulfill its semantic requirements. Take the following cases where the lexical entry is translated into pseudo-code:

1. Lexical Entry: [run]
Function: λx run(x)

2. Lexical Entry: [run TOWARD loc]
Function: λxλloc run(x,loc)

3. Lexical Entry: [run TO loc]
Function: λxλloc run(x,loc):
while(not(at(loc))):
run(x,loc)

4. Lexical Entry: [run FROM loc1 TO loc2]
Function: λxλloc1λloc2
run(x,loc1,loc2):
start-at(loc1):
while(not(at(loc2))):
run(x,loc2)

5. Lexical Entry: **[**lend object TO person]
Function: λobjectλperson
lend(object,person)

6. Lexical Entry: [lend object TO person FOR amount]
Function: λobjectλpersonλamount
lend(object,person,amount):
take_money(
person,amount)
lend(object,person)

7. Lexical Entry: [lend object TO person FOR time]
function: λobjectλpersonλtime
lend(object,person,time):
for(time):
lend(object,person)

Clearly each of the entries in the left and right columns are semantically related to the other entries in the column. Real-world programming languages tend to be able to distinguish between the entries in the right-hand column, because there are different numbers or types of arguments. However, seemingly small differences between words like “to” and “toward” show the disadvantage of current approaches to function polymorphism in programming languages. Looking at the function's name, arity, argument order, and argument types for 2 and 3, there is a significant issue. The functions have identical method signatures when looking at only these features. A language which defines the method signature in such a way would be unable to distinguish between the two at run-time.

This issue translates directly into problems for real-world programing languages.

Java allows a programmer to define methods (functions) with the same name so long as Java's parser can differentiate them (ie, they have different method signatures) (Sun Microsystems, 2009). In Java this is called method overloading (Sun Microsystems, 2009). A method signature is the function name and the types of its arguments in order (Sun Microsystems, 2009). Methods in Java require the types of their arguments to match (Campione, & Walrath, 1996), so it is possible to have functions of the same name with the same arity. The following functions may be specified in the same Java class without compile-time errors or runtime exceptions (lend/int specifies the arity of the function):

```
54. public void lend(Person person,Object object) {…}                    lend/2
55. public void lend(Person person,Object object,Amount amount) {…}      lend/3
56. public void lend(Person person,Object object,Time time) {…}          lend/3
57. public void lend(Person person,Time time,Object object) {…}          lend/3
```

54 has a different arity from 55 and 56. 55 and 56 have the same arity but different arguments. 56 and 57 have the same argument types but a different order. Since each of these methods have different method signatures, they may appear together in the same class. However Java does have issues representing the difference between [run TOWARD loc] and [run TO loc]:

```
58. public void run(Person person,Location toLocation) {…}                run/2
59. public void run(Person person,Location towardLocation) {…}           run/2
```

Since 58 and 59 have the same method signature: run(Person,Location), the Java parser would not be able to differentiate them. Thus Java's method signature definition cannot represent the subtle semantic differences between some functions without the aid of Minimal Semantic Deviation Naming outlined above.

Python implements a different form of polymorphism. Instead of allowing the programmer to specify multiple functions with the same name, it allows programmers to specify one function that can be called multiple ways by specifying default arguments (The Python Tutorial, 2009). This allows the programmer to create one function with all the arguments that might be required during any particular function call. Default arguments have quite a natural analogue in natural language. Consider the sentence "I saw Fred climbing the mountain". If someone were to tell you this, you probably would not need to ask whether Fred was climbing up or down the mountain. This is because climb, with many syntactic objects, is associated with the default direction property "up" (Levin, 2006). Looking at this sentence from the Python perspective:

```
60. def climb(object,direction="up"): …          (function declaration)
61. climb(mountain,"up")
62. climb(mountain,"down")
63. climb(mountain)
```

60 defines the climb function, which takes the object being climbed and the direction of the climbing. The direction is also a variable declaration. If no direction is specified in the function call, the function assigns the value of the direction to value specified in the variable declaration. Otherwise the function assigns the value of direction to value specified in the function call. This creates a default argument analogous to the default direction property associated with "climb". 61 and 62 specify the value of the direction argument in the function call. 63 only specifies the object and relies on the default value of the direction argument. Function calls 61 and 63 are equivalent.

Default arguments have some connection to natural language, but default values are not appropriate for all arguments of all functions, which causes issues for semantic

relatedness. Consider the possible default arguments for a function that includes all verbal alternations listed in 36-39 for the verb “run”:

64. def run(from,to,toward): …

“from” may have a default argument associated with it, because, supposedly, the object doing the running is a physical object. If the object is instantiated inside a location-based environment, it must have a location at any given time, so the default value of “from” would be the current location of the object. It is far more difficult to entertain the notion of default arguments for “to” or “toward”, because in most environments, there are no “most prominent” locations associated with “to” or “toward”. For instance, consider placing a man in the middle of Washington D.C. If someone were to instruct him to “run from the White House”, he would not automatically fill in a destination. It could be the Washington Monument, the Lincoln Memorial or any of thousands of conspicuous or inconspicuous locations inside Washington D.C. In fact, he would probably interpret that instruction as an indication that there was a reason to fear being at the White House, and that he should get as far away as possible from it. A “toward” location would not be identified for the same reasons. This issue results in a very contrived function declaration:

65. def run(from=here,to=None,toward=None): …

Now considering that each verbal alternation in 36-39 requires different semantics, there isn't an elegant solution for creating the function semantics. The best method would be to check which arguments are defined and then execute a sub-function that contains the semantics for that combination. This solution would indicate semantic

relatedness, because there is only one function declaration, but for more than a few arguments, it becomes cumbersome.

Section 2. Learnability Solutions

2.1 Introduction

Despite the number of obstacles to learning a first programming language, many people do succeed. Some manage it through university classes or tutors. Others find help in online tutorials or in books. The question is how to increase the success rate for learning that first language. Clearly there is no way to achieve a 100 percent success rate, because the programming problem is intrinsically difficult (Robins et al, 2003)(McIver et al YEAR)(Du Boulay 1986). However, by closely integrating learning resources, narrowing the domain, and simplifying its syntax, it should be possible to reduce the learning curve for first programming language acquisition.

It is necessary to focus on both the language itself and the learning materials, because the first-time programmer will not learn the language from the API, as more experienced programmers might. Almost the entire acquisition experience will be mediated by a third party. In some cases this will be classroom instruction or sessions with a tutor, but in many cases the first-time programmer will attempt to learn the language by his or herself. Even in the cases where there is outside instruction, the instructor (or tutor) might integrate the learning materials into his or her lectures (or session) or refer the student to them for reference purposes. Therefore it is absolutely vital that they be effective, easy to follow, and easy to remember.

Narrowing the domain of the language can be a difficult choice, because while

this might make it simpler to learn, it also reduces the utility of the language (Du Boulay, 1999). This means that the language will solve fewer problems and will therefore gain a smaller user base than more general languages. However, this is not an issue for a language whose only goal is aiding the first-time programmer to acquire the principles and techniques of programming. The real advantage of narrowing the domain is that it focuses the language on a particular problem, which clarifies the potential projects and the direction of the learning resources.

Syntax that is simple to understand is essential for learnability for obvious reasons. Simple syntax is easy to learn and easy to remember, while complex syntax is just the opposite. The question explored here is how to simplify syntax to a degree where it is almost natural for first-language learners but expressive enough to fulfill the requirements of the chosen language domain.

2.2 Tutorial via extended metaphor

There are many aspects to programming and therefore many concepts to learn, either implicitly or explicitly. The goal is to help first-time programmers to internalize the necessary concepts with the least amount of struggling. There are a few options to consider: rote memorization, anecdotes, metaphors, and extended metaphor. There are certainly other ways to teach and learn, but this paper assumes that the first-time programmer is learning by his or herself via a tutorial.

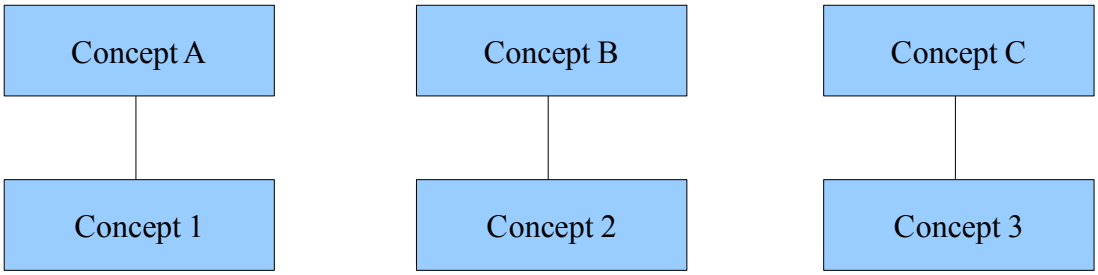
In rote memorization, or lower order learning, one memorizes the exact definitions of the concepts (Paul, 1993). For example, “A variable is something with a value that may change” (Python Programming/Variables and Strings). The issue here is

that the definition is that in order to understand what this means the author needs to clarify what a value is and how to change it. For this reason the author will often clarify with a metaphor and then some examples: “In simplest terms, a variable is just a box that you can put stuff in.”

```
changing = 3
print (changing)
changing = 9
print (changing)
```

For true rote concept memorization, the author would need to define value and the change process also. The reason that author presents metaphors and examples is that they provide a better link between the necessary concepts. It is a lot easier to remember the above metaphor and example that demonstrates the process than the initial definition and “A value is any primitive or object. Changing the value of a variable involves replacing the current value with another.” Notice also that the author would need to define a primitive and an object, creating a long chain of related concepts that the first-time programmer would also need to remember in order to remember the simple definition of a variable.

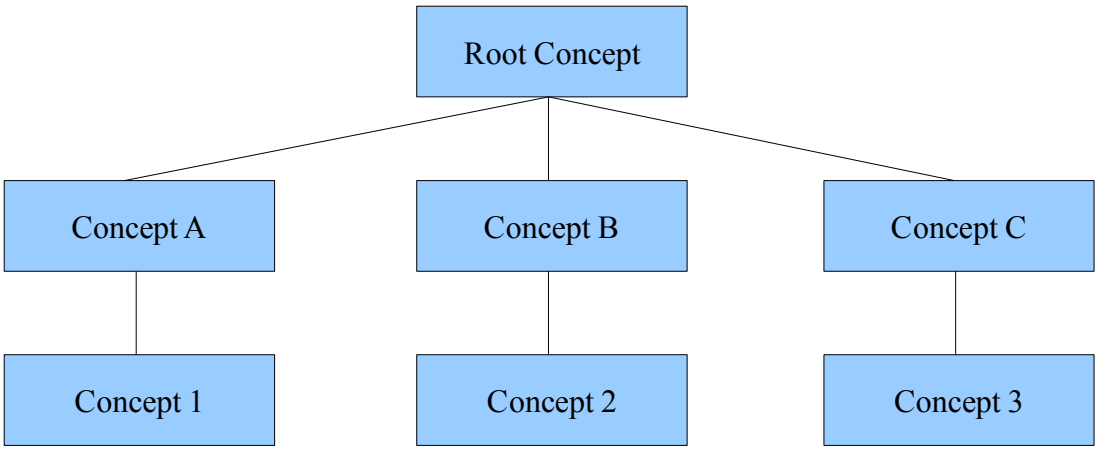
Metaphors enable the author to connect a new concept with a concept with wich the first-time programmer is more familiar (REF).



Drawing 2.1: Disconnected metaphor clusters

This allows the more familiar concept to fill in the details about the specifics of the first concept. It is also easier to remember the first concept, because it only requires the programmer to remember its connection to the second. However, as the metaphors have no relationships to each other, the programmer must remember each individually.

The key to covering a whole topic is to connect the individual metaphors with a single overarching metaphor. This creates an strong controlling metaphor; concepts connect to each other through the top-level metaphor (metaphor).



Drawing 2.2: controlling metaphor

This kind of metaphor is related to a concept know as holistic learning. There are several versions of holistic learning, but the one identified in this paper states that learning works as a whole, more connections to the metaphor increases its utility, and that with enough connections, it is possible to remember some information from the sub-concepts even if one forgets one connection (Young 2008). The overarching metaphor in this paper will be the narrowed domain of the language.

2.3 Solving familiar problems

Narrowing the domain of the language allows the author and the first-time

programmers to focus on solving one type of problem. Making this problem familiar has the added benefit of the programmer already knowing a good deal about the problem. This means that the metaphors from programming concepts to problem-specific concepts will be more natural, and thus easier to learn and remember. The question is which problem lends itself well to algorithmic interpretation and is familiar enough to create meaningful metaphors.

The answer to this is the board game. Children are introduced to simple board games from a very young age and to iteratively more difficult games as they mature. Games such as ***Memory™*** are recommended children as young as three years old (Hasbro Memory). It deals with simple concepts such as turn taking, matching, and winning conditions. ***Monopoly™*** is recommended for children age eight and up, and it deals with more complex concepts like game states, risk assessment, randomness, ownership, value, and trading (Hasbro Monopoly). Hasbro recommends ***Risk™*** for children ten and up, which teaches probabilities, complex strategy, and alliances (Hasbro Risk). It is clearly possible to represent these games algorithmically, because the rule books for the games are algorithms, and each has been made into a computer game in one form or another.

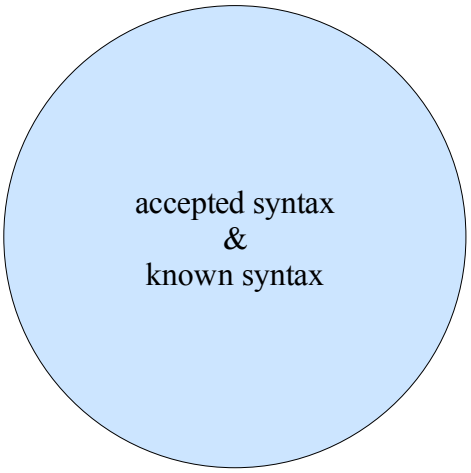
It is possible to explain programming through board games, because it is possible to create mappings from board game concepts to programming concepts. At a basic level, board games represent flow-of-control, functions, objects, and variables. Flow-of-control is the main game loop, as seen in the instruction manual, functions are the actions that the players and game take, objects are the cards, boards, game pieces, currency, etc.

with which the players interact, and variables are the attributes of the objects. This are very basic mappings; more specific mapping are introduced later in the paper.

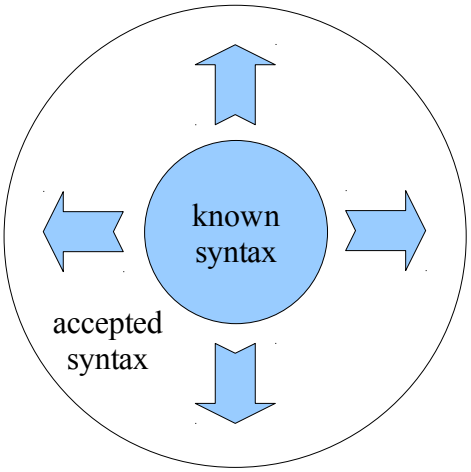
The effect of narrowing the domain to board games, is that it is possible to present board games as a metaphor for programming. The next task is specifying how to reduce the learning curve by simplifying the syntax.

2.4 Imitating English Syntax

It has long been a dream of computer scientists and programmers to be able to program by writing in natural languages (Sammet, 1966). It is easy to see why this would be favorable. For example, most native English speakers also know English syntax, and all native English speakers implicitly know English semantics (not formal semantics). Ideally this would mean that first-time programmers would need almost no instruction in syntax or semantics. In traditional programming languages, the first time programmer would start from no understanding of syntax or semantics and work toward complete understanding.

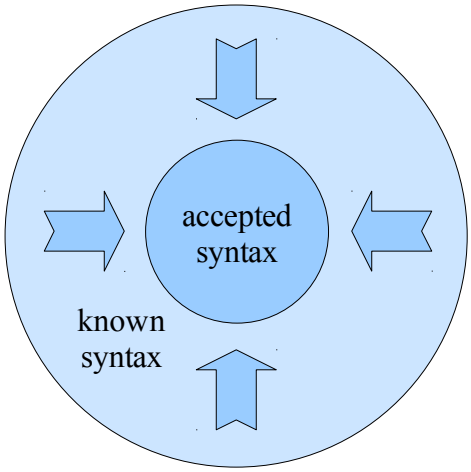


Drawing 2.3: The ideal natural language programming language accepts all grammatical English sentences.



Drawing 2.4: Programmers learn traditional programming languages by learning the syntax.

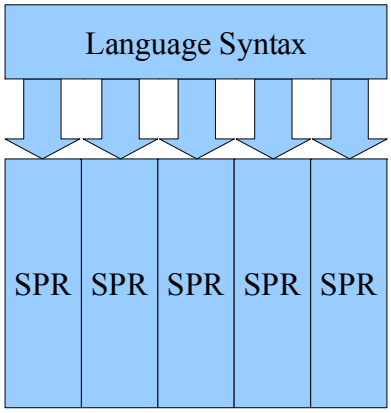
Several researchers have attempted to create large vocabulary, narrow domain natural language programming languages, and they have met various degrees of success. This paper explores these studies in the next part. In reality, the accepted and known syntaxes are not equal. In practice, a programmer learning natural language programming language will start with the entire English language and narrow it down to syntax accepted by the grammar (Halpern, 1966).



Drawing 2.5: Programmers learn actual natural language programming languages by narrowing the syntax they already know.

In other words, the programmer comes to the task of learning a traditional programming language with no knowledge of its syntax. By learning the language, the programmer builds a mental model of the accepted syntactic patterns of the language. Ideally, since there are no examples in the tutorial where incorrect syntax is presented as correct syntax, the programmer does not create faulty syntax models. In reality, the programmer forgets or misreads the syntax and creates faulty models. However, the

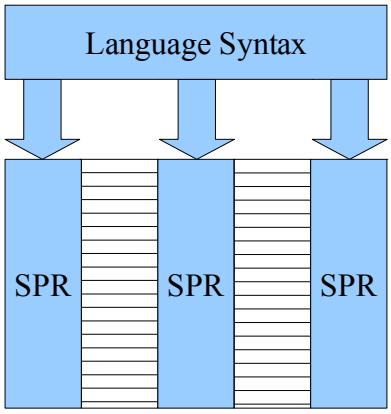
compiler or evaluator soon alerts him or her to the mental model mismatch, and the programmer rectifies the error, learning the correct syntax in the process. *Drawing 2.6*, below, shows the one-to-one connection between the actual syntax of a traditional programming language (Language Syntax) and the programmer's mental model (SPR).



Drawing 2.6: Programmers learn traditional programming languages by building mental models of syntax pattern recognizers (SPR).

With natural language, on the other hand, it is currently impossible to capture the syntax and semantics of every statement. Therefore, it is also impossible to implement the entire language as a programming language. This creates important learnability issues for a programmer attempting to learn a natural language programming language, because he or she will already have a mental model of the language, assuming that it is the programmer's primary language. Since it is not possible to create a natural language programming language that accepts all grammatical statements of the original language, there will be gaps in the syntax of the resulting programming language. The upshot of this it that the mental models for the language cannot be the same, and the programmer will likely struggle determining where they differ. *Drawing 2.7*, below, shows the gaps

in the mental model for the resulting natural language programming language.



Drawing 2.7: Programmers learn natural language programming languages by closing syntax pattern recognizers (SPR) in their mental models of the language's syntax.

There is a way to reduce the amount of confusion for the programmer attempting to learn a natural language programming language, though it might seem contrary to the goals of natural language programming. One of the goals of natural language programming is for the programmer to be able to express his or her ideas in the way most natural to him or her (Sammet, 1966). While this is certainly an admirable goal, it is not essential, and abandoning it brings natural language programming more inline with programming philosophy.

Programming languages ought to have one and only one way of doing things (McIver & Conway, 1996). Having more than one complicates implementation and understanding of the language. Natural languages have a way of saying everything they need to say, and often more than one way. Each way says something slightly different, but the root meaning is the same. The root meaning (semantic meaning) is what is

important in programming. The language does not need to understand sarcasm, fronting, implicatures, or any other pragmatic information, so letting these distinctions into the language does not help in the programming task. There are other cases where people say something differently in different situations, even though it has the exact same meaning. Note the examples below:

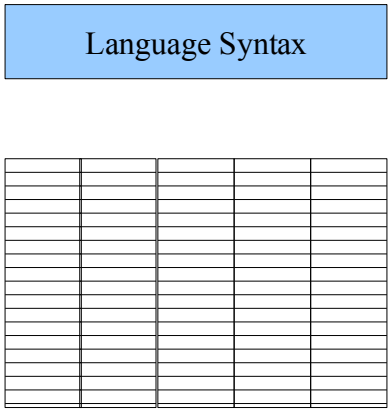
2.8	Please find me the charger of my laptop.	GOOD
2.9	Please find me my laptop's charger.	GOOD
2.10	Ryan broke the thumb of Tom	MARKED
2.11	Ryan broke Tom's thumb.	GOOD

2.8 and 2.9 are both fine, while 2.10 is grammatical but sounds strange to native ears. It is the sort of statement that would mark someone as a non-native speaker of English. 2.11, however, is what most native speakers would say. What is important for natural language programming here is that the semantics within the sentence pairs (2.8/2.9, 2.10/2.11) are the same. The reason why 2.10 is marked is unimportant for our purposes. The upshot is that natural language conventions dictate that someone say something one way in one situation and something that is semantically equivalent in a different situation.

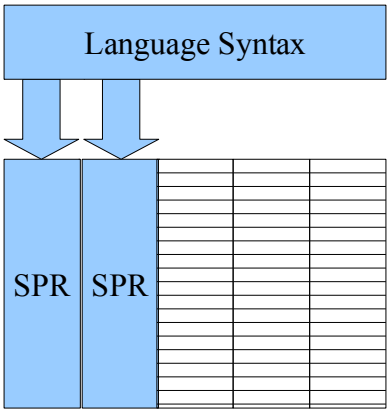
If a language designer were to design a natural language programming language that emulated the entire English grammar, he or she would need to account for both types of statement. However, since they are semantically equivalent, a language designer can implement only one. The end result does not sound 100% natural; it will sound like a non-native English speaker. Making the sacrifice of being able to say something in one and only one way has one important benefit for teaching and learning the resulting natural language programming language.

It is now possible to learn the natural language programming language like a

traditional programming language. By assuming an empty syntactic model for the language, but having a complete model for the original natural language, it is possible to learn the programming language much less effort. The programmer begins with a false empty language model as seen in *Drawing 2.8*, where each of the syntax pattern recognizers are closed. When the programmer encounters syntax blocked off in the model, he or she unblocks the associated recognizer, as shown in *Drawing 2.9*.



Drawing 2.8: A programmer can start to learn a natural language programming languages by closing each SPR in their mental models of the language's syntax.



Drawing 2.9: A programmer can learn the language by unblocking an SPR as he or she encounters its match.

2.5 Make code readable by non-programmers

The beauty of creating a programming language for which both programmer and non-programmers understand the syntax is that collaboration becomes much easier. Anyone with a logic background can troubleshoot the code, finding both syntax and logic errors. This gives the programmer many more potential pairs of eyes to find bugs in his or her code, and is one of the prime arguments favored by proponents of natural language programming (Halpern, 1966)(Gawlik, 1963).

The programmer also benefits from the ease of explaining the code to those who

are interested or need to understand it (ie. manager or colleague). Someone who does not understand traditional programming languages probably cannot understand a statement like "if add==true: items.append(item)". Instead, it is necessary to translate this statement into natural language the explainee understands with semantics that match the semantics of the programming statement. It may also be necessary to summarize, if the programmer is explaining a large section of code. With natural language however, the programmer only need summarize the code, which is a process with which he or she should be familiar from giving or receiving instructions in real life. This dramatically simplifies the process of transferring the mental model of an algorithm from one person to another.

2.6 Design for Learning Efficiency

By "learning efficiency", I mean the ease of learning the concepts behind the language as opposed to the actual code writing. Languages like Java and Python allow programmers to evaluate compound commands like the following.

```
2.12    objInst1.method1(args1).method2(args2).method3(args3);
2.13    objInst1.method1(obj2.method2(obj3.method3(args3)));
2.14    if(((1==1 && 2==2 || false) && (7<2 && 11>4 && true)) || (((true || false) &&
        5>3) || (true))) {...}
```

Consider what is necessary to understand these statements. In 2.12 the programmer must know the order in which the statement is evaluated.

2.15 Find objInst1
2.16 Find and evaluate method1 in objInst1 with args args1
2.17 Find and evaluate method2 in the result of 2.16 with args args2
2.18 Find and evaluate method3 in the result of 2.17 with args args3

In order to write 2.12, the programmer must know which methods are available at each step and what types they return. This is a process that can be made easier by methods

with revealing names or by IDEs, which show that information as the programmer is typing. However, even if this kind of chaining does not pose learnability concerns while initially writing the statement, it can be very difficult to understand when reviewing the program, which is a very important part of adding to or explaining a program. The reason is that this kind of chaining encapsulates all of the data from the first to the last step, so the only data type available is that of the last method.

To understand 2.13, the programmer must understand how the order of evaluation differs from 2.12. In particular, he or she must know that in order to evaluate a method, the arguments must first be evaluated. Thus the steps would be.

2.19 Find objInst1
2.20 Find method1 in objInst1
2.21 Find method2 in objInst2
2.22 Find and evaluate method3 in objInst3 with args3
2.23 Evaluate method2
2.24 Evaluate method1

The problem here is the same as in 2.12, namely that, upon review, it is unclear what each function is doing. Once again, proper method naming can alleviate some of this issue. However, because the entire construct appears on one line, the novice programmer might see it as one step. Clearly, from 2.15-2.18 and 2.19-2.24, this is not the case.

It would be far better for the first-time programmer not to need to deal with these problems. One could certainly make the argument, that in both Java and Python, the programmer could assign the returned value of each method to a variable and use that variable in the next step. This is certainly a valid point, but in online tutorials, forums, and other learning resources, he or she will encounter constructs like 2.12 and 2.13. These chaining constructs are valuable for those already proficient in Java or Python,

because they are unlikely to see the statement as one step and unlikely to be confused upon reviewing the statement later. However, it would be far better to force the novice to write out each step on its own line and give a meaningful name to the variable associated with it. This will make the program much easier to read, and he or she will be able to understand each step better upon encountering constructs like 2.12 and 2.13 at a later time.

2.14 is a much different construct from the other two, but broaches related issues. 2.14 is a very complex boolean test, reprinted here for convenience:

```
2.14    if(((x==y && y==y || false) && (7<2 && 11>4 && true)) || (((true || false) &&
        5>3) || (true))) {…}
```

Upon looking at it, it will take the experience logician very little time to see that it can be broken down into if((A and B) or true), which evaluates to true regardless of the unevaluated contents of A and B. However, the programmer's job is to understand every part of a statement, because each part is important and necessary.

Supposing that the programmer knows that the arguments at the lowest level must be evaluated before those at the upper levels, he or she must still find that level and understand why it is necessary. This is no easy task in its current format. It is of course possible in Java or Python to break up the line via parentheses to show each level on its own line. However, this still does not associate each level with a meaning. The best way to give meaning to a boolean is to name it, in this case as a variable. This might, on first glance, seem a silly idea. Clearly each part of each level does not need a name. This would be extremely tedious to write out and make the length of the file unbearably long. One would not be able to locate anything among the sea of variables and logic

statements. That is not exactly what I am claiming to do.

Instead of the unrestricted constructs in Python and Java or the fully constrained constructs just mentioned, I suggest a depth-limited construct. In this construct a programmer can combine any number of statements on the same level (ex. x and y and z and c), but the statement can contain at most a certain number of levels. In particular, I suggest limiting the depth to three levels. The reason for this comes from how humans express booleans when speaking in Standard Formal English. Consider the following sentences and their accompanying shallow logical segmentations. Intuitively, it is possible to change unit grouping by changing the prosody. Words in ***bold-italic*** indicate that that word is spoken with an atypical upward inflection at its start.

2.25 One Level

2.25.1. If I go to the store or go home, then …

2.25.2. If((I go to the store) or (I go home)) …

2.26 Two Levels

2.26.1. I could go to the food store and go to the fitness center ***OR*** go to class and go to the mall.

2.26.2. I could (((go to the food store) and (go to the fitness center)) or ((go to class) and (go to the mall)))

2.27 Three Levels

2.27.1. I could go to the food store or go to the fitness center ***AND*** go to class or go to the mall ***AND*** do the following: help at the food pantry and go to work ***OR*** go to the pet store and go to the furniture store.

2.27.2. I could (

(

(go to the food store) or (go to the fitness center)

) and (

(go to class) or (go to the mall)

) and (

(

(help at the food pantry) and (go to work)

) or (

(go to the pet store) and (go to the furniture store)
)
)
).

2.25 is fairly simple, because there is only one scope. 2.26 is where it becomes interesting. The instance of ***OR*** creates an outer scope for the current statement, and "go to class and go to the mall" creates inner scope separate from "go to the food store and go to the fitness center". It is the inflection of ***OR*** that causes this change. With normal prosody no outer scope is created.

2.27 shows how to create a third level with ***AND***. For there to be two more levels, one must end the statement in a way that indicates that the next statement is a continuation of the current scope (ie. "the following"). This frees the upwards inflection for the next statement. Thus "help at the food pantry and go to work" begins the second level. The ***OR*** creates an outer scope (the new second level, previous statement pushed to the third level), and "go to the pet store and go to the furniture store" creates another inner scope on level three.

It is possible to create a third level with prosody, though only with OR. Consider the following sentence and its accompanying shallow logical segmentation. ***ORR*** indicates the "or" is spoken with an upward inflection on the first syllable, and the the "R" phoneme is held longer than normal.

2.28 Three Levels (*ORR*)

2.28.1. I could go to the park or go to the library ***AND*** go to the market or throw a party ***ORR*** make my bed.

2.28.2. I could (
(
(
(go to the park) or (go to the library)
) AND (

(go to the market) or (throw a party)
)
) ORR (
(make my bed)
)
).

This kind of "or" directly follows a series of "and"s and breaks out of the ands to create an outer outer scope.

If one were to create a three-level system patterned after Standard Formal English speech, each of the arguments in a single level would need to be joined by the same logical connective (ie. (x or y or z) and (a or b or c), (x and y and z) or (a and b and c)).

It should be noted that when people describe situations in English, it is sometimes possible to create new scopes in different ways. Consider the following sentence with its accompanying shallow logical segmentation.

2.29 Alternative Scope Creation

2.29.1. I will go somewhere and buy thing1 and eat it or give it away or help my landlady carry out her garbage.

2.29.2. I will (
(
(go somewhere)
and
(
(buy thing1)
and
(eat it)
or
(give it away)
)
)
or
(help my landlady carry out her garbage)
).

Note, however, that this "and" is different from the logical connective. It means "and

then", creating a sequence of events instead of forming a logical expression.

Instead, this depth-limited approach requires novices to break down complex booleans into easy-to-manage sub-conditions assigned to variables. These variables, when properly named, serve as labels to remind the programmer (or inform a new one) of the purpose of the statement. The sub-conditions can then be recombined into a simple boolean that is equivalent to the initial complex boolean.

I chose three as the number of levels, because this is the number of levels that people can create in everyday speech in one statement. Since people create and communicate these levels, we must be able to keep them in memory long enough to reason about them, and if necessary, evaluate their truth-conditions. This is not to say that people can reason about a large number of three-level booleans chained together, but the English language does not place these constraints on the speaker, so it would be unnatural to enforce such a condition with a programming language.

2.7 Increase the precision of Method Signatures via Templates

Templates make a direct connection between verbs and functions by representing functions as complete subcategorization frames, allowing the programmer to call a function exactly like speaking a command to a person who is controlling the game.

66. RUN [Person] ***person*** from the [Location] ***start*** to the [Location] ***end***: …
67. RUN ***roy*** from the ***start_location*** to the ***end_location***.

66 is the function declaration. Like Java it restricts types on the function's arguments. The types are specified inside [], and the corresponding argument directly follows. The rest of the text serves two purposes. First it creates a grammatical English sentence. Second it serves to identify the functions arguments like English does. In other

words, the non-argument text serves to identify the argument. This means that it is possible to rearrange the arguments even if they are the same type.

68. RUN [Person] ***person*** to the [Location] ***end*** from the [Location] ***start*** : …

68 is exactly equivalent to 66, and both may specify the same function in the same class. Java on the other hand would have a signature of run(Person,Location,Location) for both, meaning that both could not be specified in the same file. More importantly this flexibility allows a language to accurately represent the subtle semantic differences between [run TO location] and [run TOWARD location]:

69. RUN [Person] ***person*** to [Location] ***location***: …
70. RUN [Person] ***person*** toward [Location] ***location***: …

Like in English there is only one word difference in the function declaration, but this allows the language to provide separate semantics for both functions. Java, on the other hand, would represent both functions as run(Person,Location). As is evident from examples 66 – 70, one can think of a template signature as a simple extension of Java's function signature. While Java's signature includes only the function name and argument types, the template's argument identification text (AIT) allows for a more specific signature:

71. run(Person,Location)	Java – to/toward
72. run(Person,"to" Location)	BGL – to
73. run(Person,"toward" Location)	BGL – toward

This template signature definition allows it to avoid the semantic closeness representation issues specified previously. AIT is the kind of unique function identifier found in the English language. As seen above, AIT solves the issue of function with the same name, arity, argument order, and argument types. It can represent Python's default

arguments with Java-style method overloading. It is possible to create a function of as lower arity that calls a function of higher arity with default arguments (Yap). Ideally default arguments should be explained in comments without confusing the purpose of the argument inside the function. Requiring this extra level separates a function from its default arguments allowing for less confusion.

Section 3. The Case for Natural Language in Programming

3.1 Introduction

For as long as there have been programming languages, there have been those who desire to borrow the syntax of natural languages to program. These are the proponents of the concept of so-called natural language programming, or NLP. There are clearly opponents of NLP, because most current programming languages do not borrow entire constructs from natural language. It is unlikely that the two sides will ever come to an agreement, because their arguments comprise the core of programming language design.

Interestingly, these debates were not continuous or mainstream in Computer Science in the early 1960s. Instead, the two sides would occasionally write articles on the subject or engage in minor debates at conferences (Halpern, 1966). The mode of debate has changed substantially since then, because proponents were not content to debate the theoretical advantages or disadvantages of NLP. The arguments between the two sides of the NLP war are no longer completely theoretical, because proponents of NLP have created implementations of natural language programming languages (NLPL), which serve as concrete battlegrounds. I explore several well-studied implementations in section 3.5.

Please note that NLP can also refer to natural language processing, but for the remainder of this paper, it will refer to natural language programming.

3.2 Goals of Natural Language Programming (NLP)

The main purpose of NLP is to make programming easy enough that a person who can already successfully complete the problem solving and planning portion of programming can represent his or her algorithm formally. The presupposition here is that the largest barrier to programming language acquisition is the syntax and semantics of the language (Halpern, 1966). A secondary, but related, goal is to provide easier explanation of the code to someone without formal programming experience (Gawlik, 1963). Explainability is certainly a desirable attribute for any programming language, because understanding the code written by someone else is the gateway to understanding the solution for the problem it solves. Further, it allows non-programmers to assist in designing the program or find high-level bugs.

NLP proponents do not agree on what extent an NLPL must represent its parent natural language. There are two main positions. The ***passive*** position is that an NLPL must only read like its parent language. The ***active*** position is that an NLPL must implement an entire subset of its parent language (Halpern, 1966). Therefore the goals for these groups are extremely different. A passive NLPL attempts to obfuscate the programming concepts from the readers of the code, not the programmer. The vocabulary of such a language is very limited. An active NLPL, on the other hand fully implements a subset of the natural language, which results in a more open vocabulary, allowing the programmer to formalize the problem however he or she sees fit. While proponents disagree on this point, they do agree among themselves and with NLP opponents on one point.

Most NLP proponents do not believe that every part of the code must be in the parent natural language. They agree with the calculus school that the mathematical notation should be maintained for arithmetic, because it is much more concise (Halpern, 1966).

3.3 Arguments for NLP

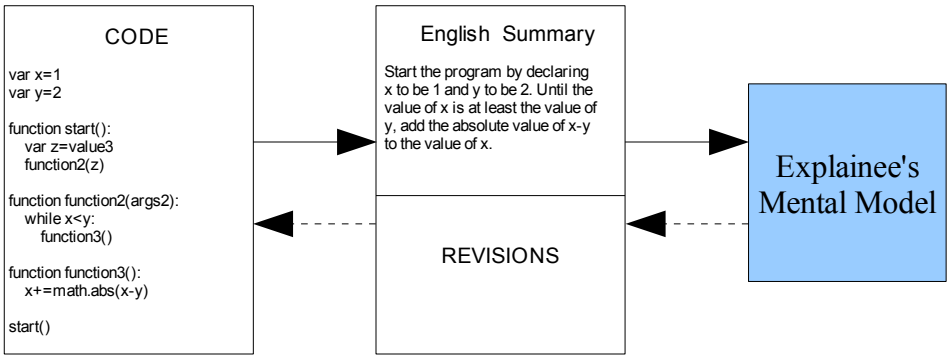
The central belief for NLP proponents is that the form of a programming language need not be determined entirely by the fact that it is addressed to a machine; they are called the ***natural-language school*** (Halpern, 1966). In particular, they believe that natural language strikes the right balance between expressive power and resource complexity (van Bentham, 2000).

The critical argument for NLP is that the primary barrier to people who would like to learn programming is not the formulating a plan, but the form of the expression. There is evidence to this effect. In 1983, Kathleen Galotti and William Gangong III carried out an experiment to test what non-programmers know about the programming process. The experiment yielded several interesting results. Previous experiments had concluded that non-programmers left out control statements, statements which govern order of evaluation, from instructions because they were unnatural. Galotti and Gangong found that by indicating to the test subjects that the instructions were for a being that does not follow the maxims governing cooperative conversation (Grice, 1975), and found that the percentage of control statements was higher than previously reported results. They concluded that with low but acceptable reliability, non-programmers possess and understanding of control statements. They also understand actions, modifiers, and

declarations, three important components of programming (Galotti & Gangong, 1985).

In 2001, John Pane, Chotirat Ratanamahatana, and Brad Myers carried out two experiments to identify how non-programmers describe solutions to programming problems. The participants in the first study were a mixture of boys and girls in the fifth grade. The experimenters gave each child a set of nine visual scenarios from the game PacMan and asked him or her to provide diagrams and or write out instructions so that a computer could accomplish the scenario. The participants in the second study included adults and fifth graders. The scenarios in this study involved creating and manipulating a database of names and numeric values. The results indicate that non-programmers possess the concepts of looping, set construction, conditionals, ranges, states, objects, object operations, and randomness. The participants did not always show perfect understanding of these concepts (Pane et al, 2001). The ramifications of this imperfect understanding are explored later in the paper.

The other main reason for NLPL is to increase the explainablity of code. NLP proponents claim that, by nature, programs written in NLPLs are more explainable (Halpern, 1966). The reasoning behind this claim involves not the number of steps in the translation, but the capabilities required at each step.



Drawing 3.1: The programmer has written code, and explains it to another person via a summary of the algorithm. The explainee then forms a mental model of the algorithm and, if necessary, summarizes any changes that should be made to the high-level algorithm. The lines returning to the code are dashed to indicate optionality.

There are several issues with this model as it stands. First, only the programmer can read the code, so if the second person forgets part of the summary, he or she must ask the programmer to explain the section again. Alternatively, the programmer could write the summary, though this takes more time. The explainee might need to ask for clarification regardless of the medium. There is also the risk that the explainee will misunderstand the summary and transfer them to his or her mental model. It is very difficult for the programmer to check whether the explainee understood the summary, because a further summary would result in loss of information. The explainee must create a high-level mental model of the algorithm from the summary and reason about it to find any design flaws. If the explainee finds an issue, he or she becomes the explainer, and summarizes the flaw in English and communicates it to the programmer. The programmer will then try to incorporate the fix into his or her code.

With an NLPL, on the other hand, the explainee from the previous scenario can read the code and construct a mental model, provided that he or she can follow the program's flow of control. The programmer can check the explainee's understanding by asking for a summary and comparing it to his or her own. This approach does demand more ability from the person trying to understand the code, but it results in fewer, more direct steps.

Further, despite opponents' claims, proponents assert that careful design can

account for problems raised by ambiguity. NLPL implementers confront the issue of ambiguity from two main directions. First, in active NLPLs, it is possible to detect many forms of ambiguity. Upon discovering the ambiguity, the program can disambiguate manually or automatically. The manual solution is to present the different options to the user an allow him or her to choose the option that matches his or her intent (Lieberman & Liu, 2007). When results are changeable and displayed immediately, it is possible to automatically choose the most likely option (Biermann & Ballard, 1980). In passive NLPLs, it is possible to carefully constrain the syntax so that no ambiguity is very unlikely and the user can be alerted. This is related to methodology to make the language learnable.

To make an NLPL learnable by non-programmers, the set of inputs can be restricted to a useful subset of a natural language (Biermann & Ballard, 1980). Biermann & Ballard describe a generic system for creating a learnable NLPL. First, find a small number of rules, which a person can easily understand and follow, to restrict the number of inputs. Then stretch the language processing technology to the point that it can reasonably cover the subset of statements to which the rules restricted natural language. In their NLPL project NLC, Biermann & Ballard claim that the acts of displaying the data structures the programmer is modifying and restricting references to only those objects reduces the subset of possible phrases. Further, the programmer begins each command or action with an imperative verb. H. J. Gawlik also restricted sentences to starting with imperative verbs in his MIRFAC prototype almost twenty years prior (Gawlik, 1963).

It is natural to convert natural language into a programming language, because

most of the parts-of-speech of natural language correspond to programming structures (Liu & Lieberman, 2004A). In particular, there is a direct correlation between verbs and functions, noun phrases and classes, adjectival modifiers and class properties, and adverbial modifiers and auxiliary function modifiers. In similar fashion, verb phrases with noun phrase arguments tend to lend themselves to an object-oriented programming interpretation (Liu & Lieberman, 2004B). Further, natural language has a concept of an object-oriented system of inheritance. The finding that non-programmers tend not to explicitly mention loops, which are vital to object-oriented programming, seem to threaten this model. In answer to Pane et al's findings, Liu & Lieberman claim that non-programmers might evaluate loops lazily to preserve short-term memory.

Some arguments for NLP are less relevant since the rise of direct manipulation interfaces, graphical interfaces in particular. One argument for NLP was to introduce new, less technical users to the computer (Petrick, 1976). As most consumer computers have been equipped with GUI environments for the past twenty years, and these interfaces provided for more intuitive direct manipulation of data, this argument has lost much of it potency.

3.4 Arguments Against NLP

The central belief for NLP opponents is that the form of a programming language must be entirely determined by the fact that it is addressed to a machine; they are called the ***calculus school*** (Halpern, 1966). The calculus school argues that NLPLs tax the programmer more than an unfamiliar syntax and semantics (Biermann & Ballard, 1980).

At the heart of this argument is the claim that Symbolic logic is superior to natural

language for formal logic (Dijkstra, 1978). In particular, Dijkstra makes reference to the fact mathematics did not progress for a long time, because of its reliance on natural language. Only when civilization adopted symbols instead of natural language did mathematics once again begin to progress. He then claims that people ought to regard formal symbols as privilege instead of a burden. The natural-language school's objection here is not that mathematical should be abandoned. Indeed, as previously stated, members agree that mathematical notations should stay, because they simplify the work of arithmetic. Instead, they take issue with the comparison of mathematics to programming; they are of course related. However, they are not exactly the same (Halpern, 1966). In particular, there are two main differences. First, mathematical objects are not real, while programming objects are, in a sense, real. Second, the goals of mathematics are flexible, as failure in one track still leads to an interesting result. In programming, however, there is a single goal, which is accomplished or not.

The most difficult aspect of programming is planning and formulating the solution precisely (Petrick, 1976). From this perspective, it does not matter whether the programming language is natural language-based or not. The result is summed up nicely by the adage “a problem well-stated is half solved” (Sammet, 1976); clearly if a programmer cannot do the first half, the second half is irrelevant. Still, the studies by Galotti & Gangong and Pane et al challenge these results by showing that non-programmers can still answer programming questions.

One of the largest and most valid concerns of the calculus school is that natural language is too ambiguous and vague to be useful as a programming language (Dijkstra,

1978) (Liu & Lieberman 2004B) (Biermann & Ballard, 1980) (Sammet, 1976). The issue here is not natural languages' inherent ambiguity, as humans can obviously communicate with little effort despite the ambiguity. Instead, the problem is trying to formalize something that is by nature informal. This argument is best summed up by the epigram "One can't proceed from the informal to the formal by formal means" (Perlis). They are certainly correct to point out ambiguity and vagueness as potential worries, as they are the mortal enemies of formalism, and natural language is incontrovertibly full of ambiguity and vagueness of every variety. The question is whether they can be constrained and controlled adequately. Several of the implementations in section 3.5. address ambiguity and vagueness directly, but none solve the problem in any provable way. This issue continues to be one of the strongest for the calculus school.

No one would want to program in an NLPL, because the syntax would be too verbose (Biermann & Ballard, 1980). It is undeniable that an NLPL would be more verbose than a purely symbolic language covering the same area, as mathematical notation is simply a shorthand for concepts in natural language (Halpern, 1966). However, this is precisely the point of an NLPL, and, more saliently, this paper. For novice programmers, mathematical notation lacks some of the relevant information that allows him or her to acquire the language. The root objection here is that writing in such a way in sub-optimal for programmer efficiency. For an experienced programmer, this is, of course, a valid concern. He or she has already acquired the concepts and techniques that allow him or her to solve problems. This person should obviously be allowed to program in a language that allows him or her to program in a more efficient manner.

It would be impossible to narrow down natural language to a useful subset (Dijkstra, 1978) (Petrick, 1976). The argument here is that the creation of the language itself would be impossible. The worry is certainly rational, that one cannot reduce a natural language to a single useful, unambiguous subset. However, this worry is most valid for general-purpose programming languages. Creators of special-purpose languages, however, can anticipate the needs of programmers, because he or she knows the types of problems ahead of time. Thus, he or she can limit the subset of the natural language to a subset that can accomplish a large majority of the goals in the desired area.

3.5 NLP Implementations

3.5.1 MIRFAC (Mathematics in Recognizable Form Automatically Compiled)

A research team, led by H. J. Gawlik, developed MIRFAC in 1963 (Gawlik, 1963). The goal of MIRFAC was not only to simplify programming, but abolish it. The subgoals of MIRFAC were to eliminate the need for a person to learn another language to program and for non-programmers to be able to check the correctness of the code. The system attempted to address the following issues with programming: equations are not written in mathematical notation, the language itself is unintelligible to the normal person, and that it is difficult to discover the problem statement by looking through the code.

MIRFAC's alphabet and type-setting system distinguish it from contemporary and modern systems. In particular, it natively supported superscripts and subscripts, eliminating the need for a special symbol to mark subscripts. Further, its alphabet

includes lower-case English and 21 lower-case Greek letters, the digits 0 through 9, and 26 mathematical symbols. MIRFAC represents variables as Greek letters.

MIRFAC allows the programmer to enter mathematical formulas or English instructions like Gawlik's following examples:

3.2 Formula: $\varphi = \sin (\psi^2 + \theta^2)$
3.3 Instruction find the smallest positive root of the equation

It is important that instructions always begin with an imperative verb and must include one and only one operand. Instructions may include multiple qualifiers.

The announcement of the system in 1963 did not fall on deaf ears. Professor Dijkstra, a devout member of the calculus school, objected vehemently to several aspects of the system (Dijkstra, 1964). Dijkstra found fault with the general premises of the language. In particular, he pointed out that even if a non-programmer could understand the code, he or she will not be able to add to it, because he or she will not be able to formulate code correctly in the language. He further claims that the language in MIRFAC has only superficial similarities to English, and that the semantic differences cause more confusion than they solve. Lastly, Dijkstra finds Gawlik's goal to have non-programmers check the code for mistakes to be pointless. He claims that a non-programmer might be able to read the code, but that he or she would not understand it and would not catch errors due to intention accommodation. In a response to Dijkstra's comments, Gawlik countered that non-programming mathematicians will indeed find errors, because the errors programmers makes are mostly errors of formula form (Gawlik, 1964).

The MIRFAC system was, in general, a great leap forward in the usability of

programming languages. In particular, its inclusion of a large number of mathematical symbols and ability to natively display superscripts and subscripts reduced the amount of unnecessary syntax, increasing readability. Further, its simple instructions in plain English, though increasing the amount of syntax, increased readability for the non-programmer. However, there are a few issues that must be addressed.

First, the type of non-programmer that Gawlik intends is clearly not similar to the target of modern programming languages, which have a much broader appeal. Also, the group's decision to limit variables names to single letter is not a technique that will help in the creation of current systems for two reasons. First, it is difficult to remember the meaning of a one-letter variable. Second, as there are a finite number of characters, there must be a theoretical upper bound on the number of variables, and thus the utility of a program written in that language.

The more significant issues, however, deal with MIRFAC's ties to English. Dijkstra raised some valid points concerning the vagueness and ambiguity of language that require attention. Gawlik did not mention how the group determined how they populated the dictionary with word senses, so there is no way to determine whether they considered the semantic ramifications of the chosen senses. Further, Gawlik did not mention ambiguity, which is a problem all language designers must face. It is not possible to determine, whether the language is constrained enough to avoid potential complications.

3.5.2 NLC (Natural Language Computer)

NLC was an interactive NLPL developed by Bruce Ballard and Alan Biermann in 1979 to cover the domain of matrices (Ballard & Biermann, 1979). The system allowed a

programmer to manipulate the matrices presented to him or her visually via natural language commands and programmer-defined word definitions. Portions of the following sentence, presented by Ballard & Biermann as representative input, will also serve as reasons to critique this system.

3.4 Add five te the 2nd positive entry in col 2.

The system processed input in four stages: morphological, syntactic, semantic, and computational.

The first stage was token identification and classification. Each token could be an integer (ex. 2, two), ordinal (1st, first), misspelled words, names, floating-point constants, and abbreviations. The dictionary included about 450 words, including imperative verbs, domain nouns (ex. column), functional nouns (ex. square root), and comparative and superlative adjectives.

Output from the morphological level was sent to the syntax analyzer. NLC's syntax recognizer was far more flexible than that of MIRFAC some fifteen years prior. It constructed noun, verb, and prepositional phrases into a tree structure to pass to the semantics processor. Besides the mono-transitive imperative verbs in MIRFAC, NLC allowed di-transitive imperative verbs and particles (what they call verbicles) such as “up” in “add up”. As in MIRFAC, though, each sentence needed to start with an imperative verb. NLC also allowed noun phrases or verb phrases to be created via conjunction.

3.5 Add the 1st and 2nd rows to row 3.
3.6 Add row 1 to and subtract row 2 from row 3.

This also means that NLC supported some forms of verb phrase ellipsis.

The semantic analyzer was fairly standard, but with some notable exceptions. First, in situations, like “te” in 3.4, where there are unrecognized unnamed words, NLC attempted to correct it to a similar word in the dictionary. Second, NLC will attempt to resolve anaphora through syntactic and semantic constraints.

Unlike Gawlik, Ballard & Biermann dealt explicitly with ambiguity in their paper. They claimed that NLC could easily detect semantic ambiguity and reported it to the user. NLC dealt with conjunction ambiguity by encoding the preferred parse in the order of the rules. NLC dealt with syntactic ambiguity through nesting (prepositional phrase attachment ambiguity) though semantic preference rules. Ballard & Biermann did not claim to have solved the issue, only to have reduced its impact in NLC.

Ballard & Biermann tested their system with an experiment designed to test whether people unfamiliar with the system but familiar with the domain could write write programs with it. They recruited 12 subjects from an introductory programming course, trained them for 50 minutes, and asked them to solve two intermediate difficulty linear equations. Only 1 of the 12 subjects could not complete the task, and the students entered 81% of the sentences were processable.

NLC is clearly a successful system from a technical standpoint. NLC shows clear progress from MIRFAC in terms of natural language processing technology, because it allows more syntactic structures verb types, and sentence types. NLC also displays changes to matrices immediately, so the programmer does not become lost in the computation. However, a few of its techniques are questionable.

The semantic processing techniques for spelling correction and ambiguity

resolution might cause problems. Consider again the sentence 3.5. There would be a significant problem if the programmer intended to type “and” instead of “to”, but forgot the rest of the statement. Clearly NLC should alert the programmer to an error. Instead, the machine would have carried out an unwanted computation. This scenario is considerably less likely, but is a potential problem nonetheless.

Also, NLC's implementation of syntactic ambiguity resolution, though better than nothing, could have created sentence with an unintended syntactic structure, changing the meaning of the statement. This could have also resulted in an incorrect computation if NLC could understand the semantic structure. Certainly the programmer familiar with the domain would have recognized NLC's mistake immediately, because NLC displayed the result, but this means that the programmer had to rearrange his or her syntax until it was something the resulted in the correct calculations.

3.5.3 METAFOR

Hugo Liu and Henry Lieberman developed a visualization tool for programming that allows people to describe their programs in a natural language narrative (Liu & Lieberman, 2004A). As a person describes his or her code to the system, it updates and displays its approximation of the code in a known programming language. The goals of the system are to help novice programmers to gain intuitions about programming and to help intermediate programmers plan and brainstorm.

METAFOR allows a programmer to have a simple conversation with a bot about the program he or she would like to make. The programmer can either make declarative statements or provide example interactions. Liu & Lieberman provide the following

examples:

3.7 When a customer orders a drink, the bartender tries to make it. When the bartender is asked to make a drink, he makes it and gives it to the customer only if the drink is on the men's (declarative)
3.8 Miranda gives you a hug (example interaction)
Mouse says “I'm here to hug you!”
Mouse hugs Miranda
Mouse says, “I made a mistake”

METAFOR then parses these statements and refines the ***scaffolding code*** of the application. Scaffolding code, in this case, is a skeleton valid programming language representation of what the program should look like. This does not mean, however, that METAFOR implements everything. For instance, the statements in 3.7 refer to several actions which can be represented by functions: order(drink), make(drink), give(drink,to_customer), say(quote,to_customer). METAFOR has enough information to implement a simple version of order(drink), because “the bartender tries to make it” is evaluated when “a customer orders a drink”. METAFOR can implement a simple version of make(drink) by the same logic. However, 3.7 only supplies the arguments for give(drink,to_customer) and say(quote,to_customer), so METAFOR does not have enough information to implement the function. It supplies valid placeholder code instead. The fact that METAFOR does not implement the internal mechanics of some functions is what makes the code scaffolding. METAFOR revises its approximation of the code the programmer continues the conversation.

Liu & Lieberman view ambiguity much differently than NLP opponents and even proponents. They see it as an advantage rather than a problem. Their reasoning for this is that programming languages force programmers to make decisions about how to

represent portions of their code. Through natural language conversations, METAFOR is able to hypothesize representations and change them if new information demands it. Liu & Lieberman present the following examples as evidence that natural language allows this kind of representational dynamism:

3.9 There is a bar. (atom)
3.10 The bar contains two customers (unimorphic list)
3.11 It also contains a waiter. (unimorphic wrt. Persons)
3.12 It also contains some stools (polymorphic list)

In this way, natural language represents an object as the most general type that still fulfills all its constraints. A programmer must do this manually in a conventional programming language. METAFOR works like natural language, automatically determining if the current representation fulfills the current constraints, and changes it if it does not.

METAFOR is implemented atop the MontyLingua natural language understanding system, which parses an input sentence into verb subject object object form. The interpreter then finds objects, properties, control structures, and other special structures. It compares objects and properties to its store of background knowledge to determine representations. The interpreter then stores the statement, current scope, and interpretive context for later retrieval. The system then updates its approximation of the representation of its programming structures and renders the code for the programmer to analyze. The system then tells the programmer, in natural language, what changes it has made and alerts the programmer to any ambiguity.

Liu & Lieberman carried out an experiment to determine the utility of METAFOR for novice and intermediate programmers. The experiment involved 13 MIT

undergraduates: 7 intermediate programmers and 6 novice programmers. In particular, the experiment's goal was to determine how brainstorming code with METAFOR affects the programmer's view of the difficulty of the task. The experimenters gave each subject some high-level programming tasks from the game Pacman and asked how long the programmer expected it would take him or herself to implement it. Then each programmer had two minutes to write a story about the task and reported how long he or she expected it would take him or herself to implement it. The programmer then recounted the story to an experimenter, who interpreted the story into semantically equivalent syntax that METAFOR can understand. The programmers then answered the same question and how likely he or she would be to adopt paper brainstorming versus METAFOR brainstorming. The results were that METAFOR made more of a positive difference with novices than intermediate programmers, but both overwhelmingly preferred METAFOR to paper brainstorming.

METAFOR is a radical departure from the other two NLPL systems discussed earlier in the section, because it is really a conversation about code with the focus being on understanding the mapping between natural language and a programming language. The other systems simply execute the code, and the programmer only learns the connection indirectly by observing the command's effect. Liu & Lieberman did not explicitly mention how the system responds to syntactic ambiguity, but they did provide a mechanism for reporting general ambiguity. It is interesting to note that the participants in the experiment largely favored METAFOR for a way of drafting a project and that several decided that they would have written the story differently if they had known the

effect on the code beforehand. This seems to indicate that the participants understood, on some level, the mapping between the languages.

The experiment might have been a little biased against brainstorming by hand, as the time to do this was limited to two minutes. Two minutes is certainly not a lot of time to think. Also, because an experimenter served as a buffer between the participant and METAFOR, the experiment was more a test of an idealized METAFOR system, where it always understood the participant's statements. Had the participants interacted with the system themselves, they would have experienced its idiosyncrasies and limitations and might have been less inclined to draft projects with it in the future. These issues do not invalidate the general results of the experiment, but they do alter what the experiment actually measured.

Section 4. Board Game Language (BGL)

4.1 Introduction

4.1.1 General Goals

Board Game Language, hereafter known as BGL (pronounced "bagel"), is an attempt to integrate the learning solutions of Section 2 into an introductory programming language. In particular, BGL allows new programmers to solve familiar problems by creating board games, the syntax of BGL imitates that of English to a large extent, which should make code readable by non-programmers, and the syntax of BGL supports learning efficiency instead of programmer efficiency.

4.1.2 Target Age Group

BGL targets people of age fourteen or older who are interested in learning to program. Some teachers incorporate other introductory programming languages, like LOGO, to those in elementary school (Papert, 1997). While those teachers achieve some success (REF), formal logic is not typically taught until later in the academic career. Those of at least age fourteen are advanced middle-school student and should have at least a passing knowledge of logic and the other concepts required to actually learn BGL. Since the domain of board games is still relevant to adults, there is no reason to cap the age of the target age group. There might be, however, an age that people would consider learning the skill of programming to be inconsequential to their careers or personal lives.

4.1.3 Implementation Language

This version of BGL is implemented in Java for a few reasons. First and foremost, Java is platform-independent, which means that language needs only to be

implemented once, and it should work across each OS that Java supports (The Java Language: An Overview). This also means that should a bug be found, fixing it once should suffice for all platforms. It also makes an interactive web-based compiler possible, which would remove the need for BGL to be installed and updated on the programmer's computer. Java is also a proven language implementation platform. Jython, JScheme, JRuby, and a host of other languages gain the benefits of running on the JVM (Open Source Scripting Languages in Java). Further, Java shares many of the semantic properties of BGL, which are explained throughout the rest of this section.

4.1.4 Paradigm

BGL takes an object-oriented approach to programming, which extends the procedural approach to programming by defining the communication objects in the problem domain and providing communication interfaces for those objects (Reid, 1993). In other words, the objects are the entities from the problem domain.(Korson & McGregor, 1990). Korson & McGregor also define the other basic concepts of object-oriented programming. First, classes define the sets of objects and should only allow outside access to a few accessor and transformation (state transition) procedures. Second, the programmer can base one class on another through class inheritance. This allows the new class to start with the functions and variables already associated with its parent class. They also define polymorphic reference to be the ability to refer to more than one class over time. For instance, if there is a function that accepts an instance of an animal class as an argument, one could pass an instance of a sheep class or an instance of a dog class, because both classes implement the required animal class methods and variables.

The object-oriented paradigm best fits the problem of board game creation. It is

very natural to see the parts of a board game as objects. Consider the types typically associated with board games. There are surfaces (like boards), cards, decks, place markers (armies in Risk, player tokens in Monopoly), decks of cards, hands, dice, spinners, and other chance objects. These are all examples of physical objects. Clearly viewing these physical objects as programming objects is not a large conceptual leap for someone already familiar with programming. Since that is the case, the idea is to explain to the first-time programmer the concept of object-oriented programming through objects with which he or she is already well-familiar.

4.1.5 Typing

BGL is a strongly-typed language, because strong typing reduces confusion over variable types. In weakly-typed languages such as Python, a variable can be associated with any type (Strong Typing). While this does not force the programmer to make a decision about the type of a variable when he or she declares it, its ability to change type can make it very difficult to track what type a variable has throughout the application. Restricting it to one type should reduce the problem of variable re-tasking, where the programmer uses the same variable for unrelated tasks, thereby giving the variable different meanings throughout the program. While perhaps less intuitive, the program itself will be much easier to understand, so the trade-off is worthwhile.

4.1.6 Built-in Types

BGL defines the basic objects of Object, Player, Card, Deck, Hand, Surface, and Marker. Game objects for a specific game are defined by extending these basic objects in accordance with the principles of object-oriented programming, so any variables or functions of the parent type are available to the child type. The programmer creates rules

for the game by defining actions over these objects.

Object is an abstract class that is never instantiated. Every other object in the game extends the Object class. Object contains only one variable, owner, which defines which Object instance is currently in possession of the instance of the current object. For instance, a Player will own a Hand.

Player is the object that represents the physical players of the game. One Player is instantiated for every physical or AI player in the game. The programmer will need to extend this Player class to create variables for the Player's game possessions (Hand(s), Deck(s), Marker(s), etc.) and functions to manage the its possessions and get a move when it is the Player's turn.

Card is the object that represents cards in real life. For the sake of simplicity, they are assumed to be 2D objects that have a front and a back. They can have two states, visible (visible to everyone), hidden (visible to no one). The owner can always see the card unless otherwise specified. The card has built-in functions to change between the two states.

Deck is the object that represents a queue of type Card. Only the top card can be seen (but can still be hidden if face-down). It has built-in functions that allow for a Card to removed from the top or added to the bottom. It also has a built-in shuffle function.

Hand is the object that represents a list of type Card. The Hand is conceptually distinct from the Deck, despite the fact that they are both essentially lists of cards. The access to the Deck is restricted to only the ends, and only the top Card is visible. In a Hand, all cards are visible and can be accessed. There are built-in functions for adding

Card(s) and removing a random Card. All other access can be done by accessing the list of Card(s) directly (or by extending the functionality of the class).

Surface is the Object that represents physical surfaces. There are no properties or functions associated with this Object. The purpose of the object is to provide a semantic base for all physical locations, as opposed to having the programmer extend Object directly. This will typically be a game board or parts of the game board.

4.1.7 Simple and Complex Types

BGL provides several types that programmers can instantiate without the class name. These include String, Integer, Float, Boolean, and List. The first four are BGLs simple types. "List" is a complex data type similar to lists in Python. Limiting the number of basic types to five simplifies the process of learning the basics of the language. Java, on the other hand, defines its primitive types to be byte, short, int, long, float, double, char, and boolean (Primitive Data Types), Array is a container (Arrays), and String is a sequence of chars (String). There is some overlap between the two. Java does have Integer, Float, Boolean, String, and List, but with exception of String, the programmer needs to instantiate a class with a primitive instead of simply allowing the primitive representation to instantiate the class (Class Integer)(Class Float)(Class Boolean)(Class String)(Class List).

Type	BGL	Java
integer	1	Integer(1)
float	1	Float(1.0)
string	"string"	"string"
boolean	TRUE	Boolean(true)
list	A List of 1, 2, 3, 4, 5	List list=new List(); list.add(1);

		list.add(2);
		list.add(3);
		list.add(4);
		list.add(5)

Table 4.1: Comparison of types of BGL and Java

Java does define an Array, which a programmer can instantiate with syntax like {1,2,3,4,5}, but only in a variable declaration (Arrays). The programmer cannot create lists in this way, and while arrays and lists are in some ways similar, the array has an important property, which makes it more difficult to learn. Since the elements of an array occupy contiguous memory locations, the maximum size of the array must be known at instantiation (Java Arrays). To create a larger array, the programmer must copy the contents of that array to a larger one. The list does not have this difficulty. While the array does typically provide better memory access time (Declaring Variables), first-time programmers should be concerned with learning high-level programming concepts, not efficiency.

Java does provide primitives for integer, float, and boolean, as shown above, but the programmer cannot access the class functions without first casting the primitive to its associated class. BGL allows the programmer to access each basic type's functions from a variable associated with that type.

```
4.3 LET x be 3.
    ADD 1 to x.
```

Java defines some primitives that a first-time programmer is unlikely to understand. What is the difference between a char and a String? What is the difference between short, long, int, double, and float? For the simple programs of a first-time programmer, the distinction between most of these is irrelevant. The only necessary

numeric types correspond to an integer, and a decimal. The programmer should have learned the difference between these two in grade school. BGL defines float instead of decimal to provide a point of reference for the programmer when he or she moves to a language that supports those distinctions. A Java float should provide enough precision for simple programs where number manipulation is not the primary task.

4.1.8 Variable and Function Accessibility

In some programming languages, for instance Java, it is possible to set access restrictions on variables and functions (Controlling Access to Members of a Class). This is desirable in deployed applications, because it allows the developer closely control data structures. If the developer only wants his or her data structure modified in certain ways, it is possible to make it unavailable to other objects directly. The object can access the data structure indirectly through methods in its enclosing class. Helper functions should only be accessed from inside their enclosing classes in order to prevent inappropriate modification of class structures. BGL does not implement variable and function privacy, because it complicates learning inter-class variable access . It is unintuitive to know an object (or function) exists and its location but not be able to access it. Also, there is no real reason to block access to an object (or function) in a language whose main purpose is to teach people the basics of programming. Privacy is a more advanced topic, which can be introduced in the next language the programmer learns.

Languages like Java also provide the ability to create variables and functions that can be accessed without instantiating an object (Understanding Instance and Class Members). These are so-called static variables and functions, and can be useful when there are functions semantically related to the class, but unnecessary to the function of a

class instance. For instance, a function named “createList”, which creates and returns a List, need not be called from an instance of a List. Surely it is related to the List class, but one should be able to create a List with this function without needing to instantiate a List first. Once again, this is an important language attribute for experienced programmers, but understanding the difference between static context and an instance would likely be more difficult for the first-time programmer than the convenience is worth. Errors involving trying to access instance-dependent functions or variables from the static context are particularly difficult to understand.

The creation and referencing of global variables is another topic that requires addressing. Some languages, such as Python and Java, have two types of scopes, namely the global and local scopes (Classes). Java has a system of object fields, which acts similar to a global scope (Using the this keyword). Every function creates a new local context, which is separate from the local context. The programmer can access any of the global variables from the local context, but because the contexts are separate, the programmer can declare a variable with the same name in the local context. Naturally, this leads to a conflict and potential programmer confusion.

Diagram 4.1: Example of possible variable confusion between global and local scopes

Which version of x is involved in the calculation of variable Z? Clearly some disambiguation is required. Java requires the prefix of "this." (this.x) to refer to the global variable x if a local version of x is present, but does not require it if there is no conflict (Using the this keyword).

This seems a decent solution, as it strikes a nice balance between programmer productivity and understandability. However, this solution breaks down with very long functions, in which the programmer decides not to include the "this." prefix. At the time of writing, it is clear to the programmer that the variable reference is to the global variable, but if the programmer returns to the function later, when he or she receives an unexpected result, he or she must remember that it references a global variable. It can be confusing. Is this variable defined somewhere in the function, or as a global variable? The global variable is defined at the top of the file, but that does not rule out a local variable with the same name in the local scope. Also, consider what happens when a programmer decides, without realizing the issue, to define a variable with the same name and type at the top of the function and reference it below the previously-written code.

The reference to the global variable will change to that of the local variable, yielding unanticipated consequences. This programmer can mitigate this problem by being attentive to variable names and by always prefixing references to global variables with "this.", but that is a lot for a programmer to learn early-on, and there is no guarantee that others will follow this rule. Clearly this makes the troubleshooting process more complex.

Python removes this problem by making the "self." prefix mandatory when accessing global variables (classes), but a problem still remains. It is still not intuitive to prefix a variable with "this.", "self.", or any other string. The beginner programmer will see the variable and expect, justifiably, to be able to reference the variable simply with its name. With Java, the first-time programmer can do this, but with the problems mentioned previously, of which he or she is potentially unaware.

The only way to provide the intuitiveness of simple variable referencing without the possibility of conflict is to eliminate the cause of the conflict. Removing the ability to have duplicate variable names between the global and local scope ensures that there will be no conflict and removes the need for a disambiguation element. This allows the programmer to reference any defined variable, regardless of global or local scope, with simply its name.

Of course, having duplicate variable names between global and local scopes can be desirable, otherwise no language designer would have expended the time and effort to implement disambiguation mechanisms. In particular, if the name most semantically relevant to a variable is already taken by the global scope, it seems obvious that declaring

it in the local scope should not conflict with the variable in the global scope. However, it is less demanding for the first-time language learner to understand that there is one scope, and that he or she cannot create two variable with the same name in the same scope than to explain disambiguation mechanisms and troubleshooting procedures.

Another important issue affecting global variables is where and when they are declared. Like Java, BGL requires that all global variables/fields be defined at the top of the class Declaring Classes). BGL places them in the "VARIABLES" section of the document. That is the only place a new global variable can be defined. Allowing the programmer to define global variables elsewhere creates a large debugging problem: Which global variables are defined now? This now depends on the order in which functions were called. Calling them in the wrong order can result in confusing errors concerning non-existent variables. Declaring them only in the "VARIABLES" section might involve more work at the beginning of writing the class, but it is completely obvious which global variables are defined throughout the class.

4.1.9 Initial Directory Setup

To create a game in BGL, the programmer first creates a game directory and one sub-directory, "objects". He or she then defines the objects of the game in text files with the "bgl" file extension stored within the "objects" directory. The rules for the game are also specified in a text file named "Rules.bgl" in the game directory. The log files generated by compiling the game have a "log" file extension and are placed in the "logs" directory, which is automatically created. The compilation creates Java source files for each object and "rules.bgl". These files are placed in "java/src/objects" and "java/src/rules" directories respectively. Each of those directories is created as it is

required. The log files from the compilation of the java source files are also placed in the "logs" directory. After BGL generates the source files, it runs the Java compiler, which will place the binary files into "java/bin/objects" "java/bin/rules". The programmer can run the .class file in rules to play the game. He or she can also archive the src and bin folders in .jar files, which are placed in the "java/jar/src" and "java/jar/bin" directories respectively. The resulting directory structure follows:

- game_name/
  - objects/
  - Rules.bgl
  - logs/
  - java/
    - src/
      - objects/
      - rules/
    - bin/
      - objects/
      - rules/
    - jar/
      - src/
      - bin/

4.1.10 Rules Layout

Each game must have one and only one set of rules that defines the structure and flow of the game. The rules file, "Rules.bgl", must be located in the game directory. "Rules.bgl" must have the following structure:

```
VARIABLES:
LET players be a new List of type Player.
global variable definition
…
global variable definition


FUNCTIONS:
function definition
…
function definition
```

```
SETUP:
statement
…
statement

GAMEPLAY:
statement
…
statement
```

The reason the document is broken down into sections is to make it obvious where to put different elements. When faced with a blank document, it can be challenging to know or decide where to begin. This structure forces the programmer to decide what is going in each place.

The "VARIABLES" section defines the global variables for the document. Programmers may define local variables in functions, but this is the only location for global variables. The programmer must declare a variable named "players" of type "Player". Only variable declarations are allowed in this space. Later variable modifications must be located in the "SETUP" or "GAMEPLAY" sections or in a function in the "FUNCTIONS" section.

The "FUNCTIONS" section allows the programmer to define functions, which he or she can call from the "SETUP" or "GAMEPLAY" sections. The programmer cannot create statements outside the function declarations.

The "SETUP" section contains statements which setup the environment for game-play. These statements may include variable declarations (local), variable updates, conditionals, loops, and function calls. The programmer may reference a variable

anywhere the type of the variable or an ancestor type of the variable matches the expected type. This is explained further in Section 4.2.

The "GAMEPLAY" section contains statements related to the actual playing of the game after the setup is complete. The "GAMEPLAY" section is itself an implicit loop, which iterates through the "players" list and evaluates the statements in the "GAMEPLAY" section. The programmer must decide where to check for a winner, and when a winner is found, the programmer calls the EXIT command, which breaks out of the loop and exits the application.

4.1.11 Class Layout

Each class must have a <class_name>".bgl" file located in the "objects" directory of the game directory. The layout for a class is as follows:

```
type declaration

VARIABLES:
global variable declaration
…
global variable declaration

FUNCTIONS:
class function declaration
…
class function declaration
```

The class must begin with a type declaration, which adds it to the type hierarchy and makes it available to other classes and the rules file to reference. The "VARIABLES" and "FUNCTIONS" sections are exactly the same as their counterparts in "Rules.bgl"

4.2 Syntactic Properties

4.2.1 Goals for Syntax

The general goal for the syntax of BGL is to have it read significantly like English at the sentence level by completing partial templates. As explained in Section 3, this is a passive implementation of a subset of English. While Halpern argued that passive implementations have all of the downsides associated with NLPLs and non-NLPLs (Halpern, 1966), there are a few benefits that he failed to see.

First, by defining a language through templates and not words, the base semantic unit in the language becomes a phrase as opposed to a word. This allows the language designer to implement phrase-level semantics without restricting his or herself to one sense for a particular word. The sense of the word comes from the surrounding context of the template. Thus, a programmer could have an imperative verb "JUMP" and a noun "jump" in the same or multiple templates. BGL simply defines the semantics over the entire phrase. This allows the programmer to parse the phrase with his or her own internal semantic parser, supplying the appropriate word senses and syntactic tree at a subconscious level. Thus a well-written template should communicate its basic semantics, without needing to read the template definition.

Second, by carefully controlling the built-in templates, the language designer can be certain that if a phrase is recognized, the system can accomplish it. Further, this limits the effect of syntactic ambiguity. Given that the built-in templates can be unambiguously recursively combined, the only possible ambiguity comes from the templates created by the programmer. This is explored in Section 4.2.x.

The end result it a language that seems contrived in its NL implementation. This,

however, is not a flaw, but an advantage of the language. The reason is that this is not a general-purpose NLPL. First-time programmers write programs in BGL to discover programming concepts and ready themselves for general-purpose non-NLPLs. Therefore the programmer should always be aware that he or she is writing code, not an essay or paper. Restricting the language to templates should make the language just unnatural enough that the programmer should be able to constrain his or herself to in-language templates.

In BGL, programmers are not forced to implement programs in natural language. BGL provides the framework to do so, but the burden is on the individual programmer to create meaningful templates. BGL does not have a word dictionary from which to check statements, so the programmer is free to create new words or define templates of completely random characters.

4.2.2 Templates

A template, for the purpose of this language, is a listing of a pattern, names, types, and roles of arguments, and an emitting type. The following is the general layout:

```
pattern: “string_1”arg_1”string_2”arg_2”…”string_n”arg _n
result:  emitting type
role_1:  arg_1 (arg_1_type)
role_2:  arg_2 (arg_2_type)
…
arg_n:   arg_n (arg_n_type)
```

The “pattern” is the string that a phrase must match. An “arg” is a placeholder for another string. The “type” of the “arg” is the string representation of the compiler type that that “arg” must be to satisfy the template. The “role” is the semantic role of the argument. It serves no purpose in the actual unification process, but it does provide those curious about the language an intuitive idea as to what each “arg” means. The “emitting

type" is the type that the pattern returns if a statement completely matches the template. A more concrete example follows:

```
#class declaration
pattern:   "a "x" is a type of "y"."
result:    ClassDeclaration
type:      y (Type)
subtype: x (Type)
```

This template defines the statement a programmer must include in a class to define its parent type. The pattern matches strings like the following:4.3

4.3 a Playing_Card is a type of Card.
4.4 a Board is a type of Surface.
4.5 a Chicken is a type of Bird.
4.6 a Octopus is a type of Sea_Creature.

Each of the referenced types must be defined in order for the template to match the statement.

4.2.3 Classes

Names begin in upper-case, and multiple words are joined with "_" (ie. Playing_Card). Declaring objects/classes is done by declaring which object it inherits properties from (ie. Board is a type of Surface). Classes are broken down into two sections: VARIABLES and FUNCTIONS. Global variables are stored in the VARIABLES section. Class-specific functions are stored in the FUNCTIONS section.

4.2.4 Variables

Names are completely lower-case (ie. playing_card). Programmers must define a variable with "LET" statement (ex. "LET the number be 0.","LET the card be a new Card."). Type is inferred by the value of the variable. BGL is strongly typed, so the type cannot be changed once declared. The programmer may define a variable and set its value to Nothing through the "type of" variable declaration template (ex. "LET the

number be a type of Integer").

To make clear the difference between defining a variable and updating one, there is a separate template for updating variables (ex. "UPDATE the number to be 2."). There is no reason to include "type of" for UPDATE, because the programmer cannot update the type of a variable.

When defining or referencing a variable, it is possible but optional to prefix the variable name with the determiner "the". This allows variables with more generic names to sound more natural while not forcing determiners onto variables with more specific names.

4.7 LET the card be a new Card.
4.8 LET card be a new Card. (marked)
4.9 LET julie be a new Player.
4.10 LET the julie be a new Player. (ungrammatical)

When referencing a variable (x) inside a certain class (class) from outside that class, BGL allows for an "x of the class" construction. This approach was more verbose, but more acceptable than the "class's x" construction, because most non-NLPLs do not modify the name of a variable, this is something the programmer would need to unlearn. If this were not a concern, the latter construction would be preferable.

4.2.5 Functions

There are many built-in functions in BGL. For a full listing, see Appendix A. Function names are completely upper-case (ex. FUNCTION) and are declared by giving the function name followed by the template used to identify the arguments. These should be imperative verbs, but as BGL contains no dictionary, there is no way to enforce this. The programmer may define his or her own functions in the "FUNCTIONS" sections of

"Rules.bgl" or any of his or her classes. Patterns for functions in classes must end in "this <curr_class_name>", which identifies it to BGL as a class function. Functions defined in "Rues.bgl" may not end in "this <class_name>". Types of arguments are defined inside "[…]", which immediately precede the name of the argument. Take the following example of a function declaration:

Declaration
TRANSFER [Object] object from [Location] loc1 to [Location] loc2:

BGL then pulls the arguments from the following statement:
TRANSFER the boat from the river to the sea.

object = boat
loc1= river
loc2 = sea

The programmer cannot retrieve the object returned by a function call directly like he or she could in Java or Python (REF). There are two reasons for this. First, American English does not support this kind of access from present tense imperatives. The programmer would need to write in the progressive.

```
4.11    LET the returned_object be MOVE the card to the deck.
4.12    LET the returned_object be the object created by MOVING the card to the
        deck.
```

This violates BGLs view of treating functions as imperative verbs and would create an implicit function call. Second, the programmer should not view a function call and the object returned by the function call as the same thing. Therefore, it makes sense to separate the function call and the object:

```
MOVE the card to the deck.
LET the returned_object be the result of MOVE.
```

4.2.6 Statements

Statements begin with a function name (unless it's a conditional), and end with a

“.”. There should not be a problem with decimals, because digits after the “.” will disambiguate the statement. “,” indicates a new level in the scope of the program (ex. if mars is “red”, DO something). BGL recursively unifies statements against templates to identify functions and arguments. Take the following example:

Declaration (taken from Playing_Card)
```
FLIP this Card:
   if the state is "face-up", let the state be "face-down".
   otherwise, if the state is "face-down", let the state be "face-up".
   otherwise, FAIL.
   SUCCEED.
```

Application
```
LET the card be a new Playing_Card.
FLIP the card.
```

Each function must declare a return type, even if the return type is Nothing (ex. “LET the return type be Nothing”). Also, to make the first-time programmer aware that each function must either succeed or fail in its task, he or she must declare “SUCCEED.” or “FAIL.” state directly before the return statement. The programmer can access this state directly after the call to the function.

```
TRANSFER the troops from territory_1 to territory_2.
If TRANSFER succeeded, END TURN.
```

4.2.7 Scope Changes

Like Python, BGL marks scope changes by changes in indentation (Defining Functions). In particular, the scope change is exactly for spaces. No tabs are allowed for indentation, because combinations of tabs and spaces do not work well in some text editors. Forcing scoping in this way forces the programmer to learn good indentation, which will make his or her code more readable and easier to debug.

In order to go down one scope level, the previous statement must end with a “,”.

These types of statements include loops and conditionals. If the entire new scope is only one line, the programmer can choose whether to keep it on the same line or move it to a new line, indenting by four spaces. Only one new scope is allowed per statement. The following are examples of proper scope introductions:

```
4.10    if 1 is 1, SUCCEED.
4.11    if 1 is 1,
            SUCCEED.
```

To go back to an outer scope, the programmer simply needs to decrement spaces in the next line by the number of scopes he or she wishes to eliminate multiplied by four. If a new scope was introduced on the previous line, and there was a statement following the ",", the programmer does not need to change the indentation to close that scope. The following are examples of proper scope eliminations:

```
4.12    if 1 is 1, SUCCEED.
        GIVE the card to the deck.

4.13    if 1 is 1,
           SUCCEED.
        GIVE the card to the deck.
```

4.2.8 Conditionals

BGL provides the types of conditional statements found in Java and Python, but without the parentheses. BGL implements the "and", "AND", "or", and "OR" logical operators from Section 2.6.

The base form of the conditional is the "if" statement, which takes a boolean condition and tests whether it is true. If the condition is true, then it executes the code after the ",", which indicates a new scope. Examples 4.x and 4.y from Section 4.2.7 demonstrate the base conditional.

The alternative conditional, "otherwise, if", may only be placed directly after a

base conditional or another alternative conditional. It evaluates its condition only if the condition of the previous conditional evaluated to false.

```
if 1 is 1, SUCCEED.
otherwise, if 1 is 2, SUCCEED.
otherwise, if 1 is 3, SUCCEED.
```

The reason for the "," between "otherwise" and "if", is that to give the "otherwise" more emphasis, native English speakers sometimes pause. BGL interprets this pause as a new scope.

The default conditional "otherwise" (with no "if"), may only be placed directly after a base or alternative conditional. It evaluates its condition only if the condition of the previous conditional evaluated to false.

```
if 1 is 1, SUCCEED.
otherwise, FAIL.
```

4.2.9 Loops

BGL provides both general looping mechanisms and mechanisms for iterating through specific structures. The general looping structure is called "UNTIL", which loops until a boolean condition is true (ex. "UNTIL x is > 3, …"). BGL also provides mechanisms for exiting the loop and returning to the beginning of a loop without evaluating any of the later statements. "EXIT loop." and "GO to beginning of loop." provide these functionalities.

BGL also provides a "FOR each" statement, which will iterate through either a List or an Integer range.

```
4.8 FOR each entry in the list, …
4.9 FOR each number between 1 and 6, …
```

BGL further allows for with/without filters on iterations through lists:

```
4.10        FOR each card in the deck with state "hidden" and without value 2, …
```

The first part of the filter defines the variable to check, and the second part defines the value the variable must have ("with") or must not have ("without"). The above is syntactic sugar for the following:

```
4.11        FOR each card in the deck,
                if the state of the card is "hidden" and the value of the card is not 2, ...
```

4.2.10 Comments

Comments are an essential aspect of programming, as they help the programmer who wrote the code to remember what it does, and they help someone reading the code understand it. BGL implements the shell-style comment, where a "#" not apart of a string signals that the rest of that line should be ignored by the compiler (Shells and Shell Scripts). The reason that BGL implements "#" instead of the convention of "/* … */" for multi-line comments is that, while multi-line comments are convenient, it is easy to lose track of either end of it (King, 1997). If only one end is removed, there will be a syntax error, which can sometimes be very difficult to find.

4.2.11 Anaphora

To promote a more natural, conversational, flow of commands, BGL allows the programmer to reference the most recently-mentioned instance of an object by prepending the name of the type of the instance with "the ".

```
LET first_card_to_draw be a new Card.
LET the discard_pile be a new Deck.
LET x be 1.
MOVE the Card to the Deck.
```

4.2.12 Agreement

While agreement is an important aspect of grammaticality, BGL does not have built-in support for number, gender, or any other type of agreement. This would require a substantial dictionary and automatic recognition of plurality and gender, which would

add significantly to the complexity of implementing the language.

4.2.13 Ambiguity

In designing BGL the templates have been carefully created in such a way that before the programmer begins writing code, there is no ambiguity. However, a programmer can introduce syntactic ambiguity under a very specific set of circumstances. He or she must create two functions with the templates that only vary by replacing one word of one template with a variable in the second, and that one word must be the name of a variable defined in the program.

```
LET x be a type of Integer. (variable declaration)
DO something to [Integer] integer: … (function declaration 1)
DO something to x: … (function declaration 2)
DO something to x.
```

Clearly this is not an example of good function naming or argument placement, but it is possible. This kind of ambiguity is very easy to find, and it is then possible to alert the programmer to the problem.

4.3 Compilation

4.3.1 Basic Methodology

The compiler for BGL is implemented in Java, which, as mentioned previously, provides BGL with cross-platform capabilities. The basic compilation process is as follows:

1. Load the built-in BGL types
2. Add the built-in BGL types to the type hierarchy

First Pass (Shallow)

3. Find the types and names of the global variables defined in the user-defined types
4. Find the names, patterns, arguments, and argument types defined in the user-defined types
5. Add the user-defined types to the type hierarchy

Second Pass (Deep)

6. Evaluate each line of code in Rules.bgl and the user-defined types
    1. Split the line into phrases
    2. Unify each part of the line with a template
    3. Compile the filled template into an intermediate code representation
7. Semantically evaluate each phrase of code
8. Compile each phrase of code into its Java source representation

Packaging

9. Place all Java source files into the game directory's "src" directory
10. Compile the Java source files
11. Move the binary files to the game directory's "bin" directory
12. (optionally) Create an executable jar file of the "src" and "bin" directories

4.3.2 Automatic Type-Shifting

The inclusion of a type hierarchy in BGL allows for automatic type-shifting during compilation. The explanation is fairly simple. BGL is an object-oriented language, which implements inheritance. When BGL encounters a certain type during compilation, but it requires another type, BGL checks the type hierarchy for a direct path upward from the current type to the required type. If BGL finds such a path, it automatically shifts the current type to the required type. Otherwise, it prunes that branch from the derivation tree.

4.3.3 Unification

When BGL requires a unification of phrase of source code, it attempts to match the top-level against each template in the grammar. At the top and intermediate layers, there can be more than one match.

Phrase:	1 is not 2		
Template 1:	x" is "y	Match 1:	[x=1,y=2]
Template 2:	x" is not "y	Match 2:	[x=1,y=not 2]

The initial incorrect matches, like template 2, are eliminated when more information becomes available. BGL recursively attempts to unify each argument (x and y above) with either another template, a variable, a type name (ex. Card), or a type representation

(ex. “text”). If none of these fit one of the arguments, BGL prunes that branch of the derivation tree. If BGL reaches a point where all arguments are properly specified, it has reached the bottom of the derivation tree.

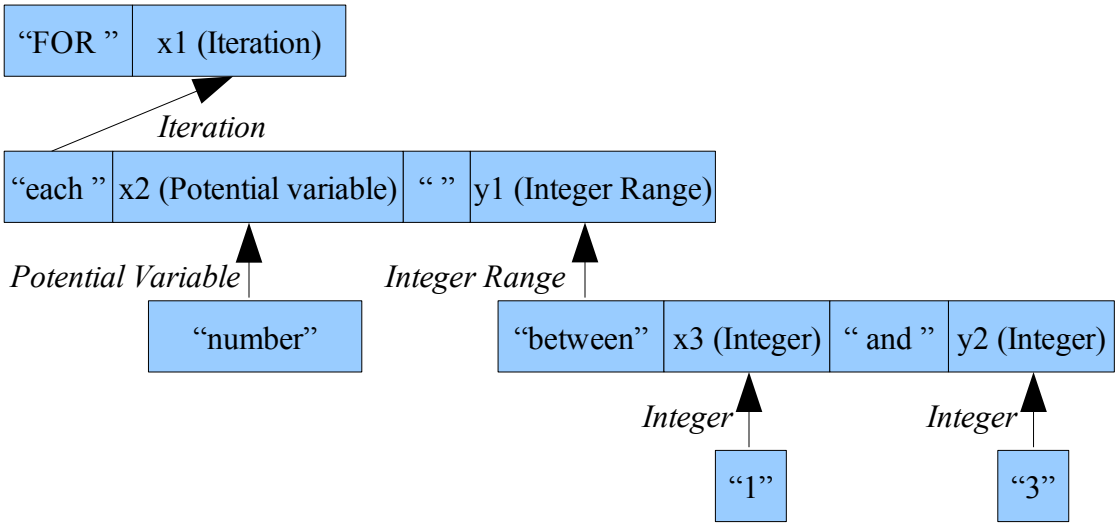
At this point, there can still be multiple matches. Consider the following,

Phrase 1: SUBTRACT “F” from “FAT”.
Phrase 2: SUBTRACT 2 from 3.
Template: “SUBTRACT “x” from “y”.”

Both Phrase 1 and Phrase 2 match the current template, because both String(s) and Integer(s) are simple types with direct representations. However, templates also supply the types for their arguments. In the example above, arguments x and y must be Integer(s), so BGL knows to prune the current derivation tree for Phrase 1, and ultimately reject the statement. BGL determines that the types of the arguments for Phrase 2 fit the template.

If the current level is not the top of the parse tree, and it is a valid type, the level above must ensure that the current level is of the proper type. For the previous example, this is quite easy, as the leaves of the tree are simple types. However, consider the following simplified statement and derivation:

Phrase: FOR each number between 1 and 3, …



In each step of the derivation, the tree requires its arguments to have certain types. The leaves are clearly Integer(s), but there are intermediate phrases also, which must have a type to prevent unwanted derivations. Therefore, each template has an “emitting type”, which is the type of statement that the template describes. Each level in the derivation tree checks the emitting type of its arguments. For simple types, it is the simple type itself.

At every step of the derivation, BGL builds up the semantics of the statement while tracking the variable names, values, and types of each of its arguments. For special types, like “Potential Variable”, which don't actually exist in a meaningful way, BGL makes additional checks to ensure that it fits the criteria of that type. The arguments of the “Iteration”, above would be “IntegerRange(Integer(1),Integer(3))”, which captures all of the information required for the next level, dropping the variable names.

Grammars like these are typically called “Context-free Grammars”. However, the tracking of the type of the argument is not within the capabilities of the CFG, so this

grammar is instead an attribute grammar, which extends the CFG by allowing it to track attributes and pass them both up and down the derivation tree (Knuth, 1968).

4.3.4 Semantic Evaluation

To ensure that the BGL source will compile correctly into Java, BGL checks the order and position of statements. For instance, both Java and BGL require that an alternative conditional follow a basic conditional and that a default conditional follow all alternative conditionals and basic conditionals. BGL checks that the ordering of these statements is correct and stops compilation and alerts the programmer if they are not.

4.3.5 Error Recognition and Recovery

An important aspect of learning a language is the language's ability to alert the programmer to mistakes he or she has made. BGL currently informs the programmer of the type of error, the line number, and the file name. However, for first-time programmers, this amount of information will not be enough.

Rules.bgl: Line 21: “PLAEC the card into the goal of the board” is not a valid statement.

Future versions of BGL will run an edit distance algorithm on all the failed derivations and return the n closest syntactic changes that will make the the statement valid.

Rules.bgl: Line 21: “PLAEC the card into the goal of the board” is not a valid statement.
Consider changing “PLAEC” to “PLACE”, which is a function that is defined in Rules.bgl.

Section 5. Experiment

5.1 Goals

I designed and carried out an experiment whose goals were to determine whether natural language and a familiar problem are enough for non-programmers to understand the high-level algorithm of a program and to compare non-programmers understandings of the syntax and semantics of BGL and Python.

5.2 Setup

5.2.1 Participant Selection

The participants for the experiment were limited to men and women currently attending a university or having graduated in the past five years. They were chosen with a preference for the humanities, because they are less likely to have experience with formal logic. There were eight participants in all, six women and two men. It was easier to find women who had never programmed before than it was to find men. Similarly, it was easier to find people in the humanities who had never programmed before, so six of the eight participants were from the humanities.

It was not possible to oversee all of the test subjects, as they were spread out geographically. All participants were told to select a learning environment comfortable to them. They were also told to address any questions to me, not to ask someone else or find the answer on the internet. I received only a few requests for clarification, and only one I had to refuse to answer, because it would skew the results of the experiment.

5.2.2 Game Implementations

I choose solitaire and card pair matching as the games to implement for this experiment. Both games are fairly simple to understand and play, require the same physical objects to play in real life: cards and a surface, and are fairly popular in the United States. I implemented the general algorithm for solitaire in BGL. However, since BGL is not fully implemented at this time, I could not check whether the game would compile and play exactly as anticipated. I was able to implement the entire card matching game in Python, along with two players that select one random card and tries to match it with another random card. Card matching requires fewer low-level functions, so I was able to successfully test this implementation. I did not place any comments in the code, except in the case of the of the player implementations, which are primitive AI.

5.2.3 Test Questions Revision

I gave the test to one person first, to ensure that none of the questions were unreasonably confusing, and I made one change to the experiment as a result of that. Question 10 in Game 1 initially asked the participant to write two lines of code (the lines should have been the same), but I changed that to requiring one line of code. I made that change to all the other sets of questions before sending the experiment to the rest of the subjects.

5.2.4 Test Layout

The test packet that each participant received included with it a data release form, instructions, an entry survey, an exit survey, and the source code for both games with their accompanying questions.

The instructions explained the goals of the experiments and assured the participants several times that it was not they who were being evaluated, but the

learnability of the two languages involved. The instructions also provided metaphorical explanations for the terms of object, variable and function. The participants were instructed to read the source code and to try to identify variable declarations, function declarations, function calls, and object creations. Then they were instructed to fill out the questions for that game, take a break, and repeat the process for the other game. The instructions provided a list of tips for reading and navigating through the code.

The surveys collected relevant demographic information as well as the participants comfort level with the concepts of object, variable, and function. In particular, the entry survey asked:

1. Have you ever programmed before? (yes/no)
2. What is your age?
3. What year did you graduate college (or still attending)?
4. What were your majors and/or tracks in college?
5. How well do you feel you understand the concepts? (Likert scale for each)
6. With which of the following games are you familiar?
    1. Solitaire
    2. Freecell
    3. Card Pair Matching
    4. Checkers
    5. Chess
    6. 4 in a Row
    7. Othello

The exit survey asked:

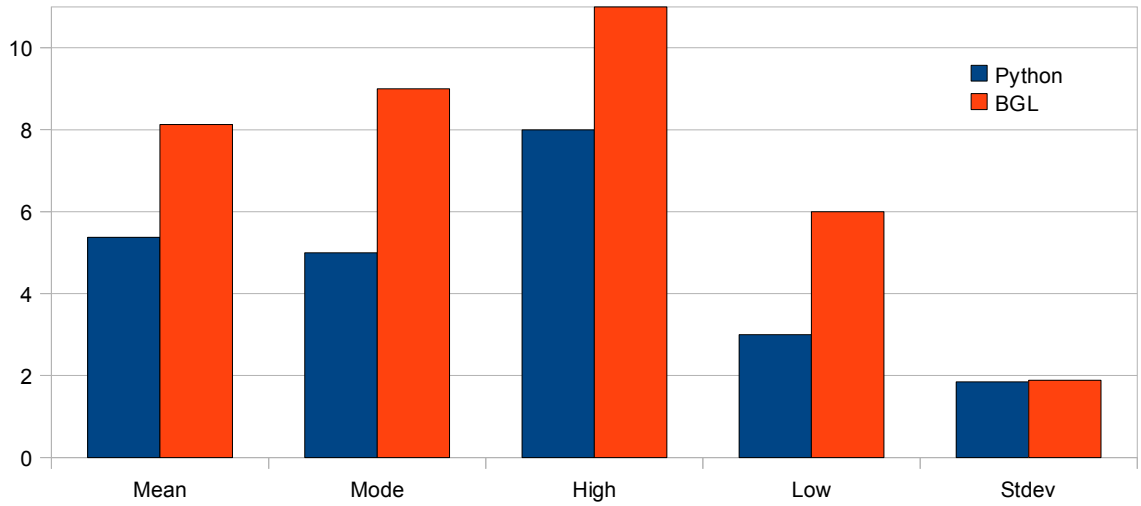
1. How well do you feel you understand the concepts now that you have been exposed to them (same Likert scales)
2. Which game did you look at first?
3. Comments

The test questions in Game 1 and Game 2 had the same numbers and types of questions, but the questions had a different order. The questions were designed to determine what level of the algorithm the participants understood. At the highest level,

the questions ask what game the code implements. The questions also ask in which file certain functions and variables are implemented. There are questions about what types of things (variables/functions) are being defined at a certain line and where to place certain declarations in the document. There are also more in-depth knowledge tests which ask the participant to explain the purpose of a function at a high level and ask the participant to write single lines of code. The purpose of having these later difficult questions was to determine the maximum depth of knowledge that the participant attained during his or her reading of the code.

5.3 Results

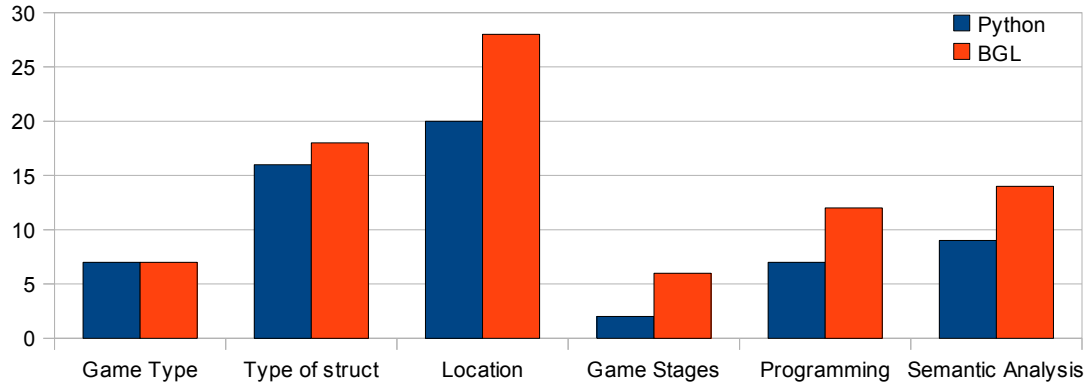
The overall outcome was very favorable for BGL, as participants answered, on average, 74% of the questions correctly while only answering 49% of answers correctly for Python.



Graph 5.1: Summary of statistics for overall test scores

The total number of correct answers ranged from 3 to 8 for Python and 6 and 11 (all of them) for BGL. The standard deviations for the answers of both languages are almost identical.

Each question was associated with one or more question type. Answering a question incorrectly does not necessarily mean that the participant answered all aspects of the question incorrectly. If the participant answered a question partially correctly, he or she received no credit in the overall test score for the question, but received credit for type of the part of the question that he or she did answer correctly.



Graph 5.2: Totals of each type of question answered correctly

5.4 Discussion

Clearly the participants benefited from the game implementation in BGL over Python. The participants answered 2.75/11 more questions correctly for BGL than for Python. There were no question types for which participants answered more correctly for Python than for BGL. The question types with the clearest differences were Game Stages, Programming, Semantic Analysis, and Location.

The reason for the significant difference in game stage identification is that BGL marks each of the stages explicitly in the structure of the code, while Python does not. BGL's GAMEPLAY section defines an implicit while loop, in which the winning conditions are tested at the beginning of each iteration. This understanding likely affected the other question types as well, because a programmer should expect certain

types of programming constructs in certain stages of the game.

The programming statistic is the most important for this study, as it shows a clear advantage for BGL over Python on the merits of natural language programming alone. Not only did significantly more people get the answers right, those who did not were typically closer than those who answered incorrectly for Python. For example, one person wrote the following answers for question 9 for Game 1 and Game 2 respectively:

Given: if 1 > counter(): return GOOD
Correct: if 1 > counter: print “GOOD”

Given: if 1 > the value of “counter”, DISPLAY “GOOD”
Correct: if 1 is > the counter, DISPLAY “GOOD”.

In both, the participant seems to understand the general syntax, but in the former, the participant confused a variable name with a function call or object declaration. In the latter, the participant clearly understands the concept of a variable but incorrectly assumes that one must access the value of the variable through an attribute called value.

Another participant made a significant improvement in BGL for question 8.

Given: def counter: (counter(value))=0
Correct: counter=0

Given: Let the counter number be 0
Correct: LET the counter be 0.

For Python, the participant seems to have confused the syntax for declaring a function with the syntax for declaring a variable. The participant also seems to be trying to declare the value of a counter object to 0. In BGL, however, the participant has the general syntax down perfectly, but it is unclear whether he or she meant to tie “counter” and “number” with “_” or whether “number” is supposed to be the type of the number. It is clear, though, that the participant understands the concept of a variable from the BGL

perspective, even though the variable declaration itself is significantly longer in BGL than in Python.

Another participant answered as follows for question 8

```
Given:      def counter(0)
Correct:    counter=0

Given:      let the counter_be 0
Correct:    LET the counter be 0.
```

In the Python version of the question, the participant is clearly confusing the syntax for defining a variable with that of defining a function. Save for the errant "_", the statement is correctly formatted.

Several of the participants answered as follows for question 10 on BGL:

```
Given:      DEAL the top card to the deck from the other_deck of the board.
Correct:    DEAL the top card to the other_deck from the main_deck of the board.
```

There is a distinct possibility that having the "to" before the "from" confused several people during the test. One participant mentioned in the comments section of the exit survey that it was an unnatural construction.

Participants generally did well on the semantic analysis questions, with the exception of question 11 for Python. Question 11 for both Python and BGL asks where the game checks for whether there is a winner. BGL calls the function that checks for winning conditions at the first line in the GAMEPLAY section. The Python game implementation checks in the while loop, but most of the participants chose the function call after the game loop exited that checks who the winner is. They clearly did not understand that the game already had to know that there was a winner to stop the loop. All but one person answered the same question correctly for BGL. Question 11 also counted for location, which explains why the difference between BGL and Python is so

high for that statistic despite the fact that most people answered the location only or location/object questions correctly for Python.

Several people noted in the comments or in informal interviews that BGL was much plainer and easier to understand. A few mentioned that Python's syntax was too dense and confusing. Though one person felt that BGL was too verbose, the general response to BGL can best be summarized by one of the participant's comments:

> The code for Game 2 was much easier to understand than for Game 1. The language used in the code for Game 2 is much plainer and straight-forward (even without the section labels), an is thus in my opinion much easier for someone with no programming knowledge to understand.

Summary and Conclusion

In this paper, I have argued that current programing languages are not ideal for teaching people how to program. In particular, they do not adequately help first-time programming with the confluence of topics that confront them, do not provide an ideal tutorial system, have too broad a problem domain, and do not precisely represent semantic closeness.

In the second section, I argued for a specific set of improvements that should theoretically reduce the additive learning curve for first programming language acquisition. In particular, the language designer should limit the language to a particular domain with which first-time programmers have experience, strive to make a truly unified tutorial that explains programming concepts via metaphor to the language domain, implement a subset of natural language, and design for learning efficiency instead of programmer efficiency.

In the third section, I introduced the problem of natural language programming and the ongoing debate on the subject. I weighed the advantages and disadvantages to both sides, and critiqued three NLPL systems spanning from the 1960s to the present.

In the fourth section, I introduced BGL, a language that is currently in development. BGL is a passive NLPL designed specifically for first-time programmers. It allows the programmer to implement board and card games in a natural language-like syntax and run the resulting Java code on any platform that Java supports.

In the fifth section, I evaluated BGL by implementing one board game in BGL and one board game in Python and contrasting how non-programmers understand programs written in both languages. The results overwhelmingly favored BGL for code evaluation and generation, and the participants generally favored BGL over Python.

It is clear that BGL, despite its verbosity, has an appeal to first-time programmers, and is headed down the correct path for future first programming language acquisition. The next steps for aiding those interested in learning a programming language will be to finish the implementation of BGL, to create a tutorial, and release it to the public as open source software.

Natural language programming has important roles in the world of formal languages, but perhaps not the general purpose roles that the natural language and calculus schools envisioned in the 1960s. Instead, natural language programming can be valuable project planning and pedagogical tools as shown by METAFOR and BGL. This is, in a way, a compromise between the opposing views, because it recognizes the validity of specific-purpose natural language programming while leaving the work of general-purpose programming to non-natural language programming languages.

Appendix A: BGL Key-Phrases

```
#FAIL
pattern: "FAIL."$
result:  Failure

#SUCCEED
pattern: "SUCCEED."$
result:  Success

#class declaration
pattern: ^"a "x" is a type of "y"."$
result:  ClassDeclaration
type:    y (Type)
subtype: x (Type)

#variable declaration (by type)
pattern:  ^"let "x" be a type of "y"."$
result:   VariableDeclaration
type:     y (Type)
label:    x (<<<NewVariable>>>)

#variable declaration (by value)
pattern: ^"let "x" be "y"."$
result:  VariableDeclaration
value:   y (Value)
label:   x (<<<NewVariable>>>)

#variable declaration (by List type)
pattern: ^"let "x" be a List of type "y"."$
result:  VariableDeclaration
type:   y (Type)
label:   x (<<<NewListVariableByType>>>)

#variable update
pattern: ^"update "x" to be "y"."$
result:  VariableUpdate
value:    y (Value)
label:   x (<<<OldVariable>>>)

#condition declaration
pattern: ^"condition "x":"$
result:  ConditionDeclaration

#VARIABLES section
pattern: ^"VARIABLES:"$
result:  VariablesSection
```

```
#FUNCTIONS section
pattern: ^"FUNCTIONS:"$
result:  FunctionsSection

#SETUP section
pattern: ^"SETUP:"$
result:  SetupSection

#WINNING section
pattern: ^"WINNING:"$
result:  WinningSection

#otherwise conditional (else if)
pattern: ^"otherwise,"
result:  AlternativeConditional

#first conditional
pattern:    ^"if "x","
result:     FirstConditional
type:       x (Boolean)

#UNTIL
pattern:    ^"UNTIL "x
result:     UntilIteration
type:       x (Boolean)

#conditional
pattern: ^x", "y"."$
result:  StraightConditional
left:    x (Antecedent)
right:   y (Consequent)

#outer disjunction (only after all ands)
pattern: x" OR "y
result:  OuterDisjunction
left:    x (OuterBoolean)
right:   y (InnerBoolean)

#outer disjunction
pattern: x" OR "y
result:  OuterDisjunction
left:    x (InnerBoolean)
right:   y (InnerBoolean)

#outer conjunction
pattern: x" AND "y
result:  OuterConjunction
left:    x (InnerBoolean)
right:   y (InnerBoolean)

#disjunction
pattern: x" or "y
```

```
result:  InnerDisjunction
left:    x (InnerBoolean)
right:   y (InnerBoolean)

#conjunction
pattern: x" and "y
result:  InnerConjunction
left:    x (InnerBoolean)
right:   y (InnerBoolean)

#value equality test
pattern:  ^x" is "y$
result:   ValueTest
left:     x (Value)
right:    y (Value)

#value equality test
pattern:  ^x" is "y$
result:   ValueEqualityTest
left:     x (Value)
right:    y (Value)

#negated value equality test
pattern:  ^x" is not "y$
result:   NegatedValueEqualityTest
left:     x (Value)
right:    y (Value)

#<
pattern: x" is < "y
result:  LessThanTest
left:    x (Value)
right:   y (Value)

#<=
pattern: x" is <= "y
result:  LessThanEqualToTest
left:    x (Value)
right:   y (Value)

#>
pattern: x" is > "y
result:  GreaterThanTest
left:    x (Value)
right:   y (Value)

#>=
pattern: x" is >= "y
result:  GreaterThanEqualToTest
left:    x (Value)
right:   y (Value)

#Type of
```

```
pattern: "type of "x
result:  Type
value:   x (Value)

#shallow copy
pattern: "a copy of "x
result:  <<<Type of x>>>
value:    x (Value)

#deep copy
pattern: "a deep copy of "x
result:  <<<Type of x>>>
value:    x (Value)

#for iteration (inclusionary)
pattern: "FOR "x
type:    x (InclusionaryIterationCondition)
result:  ForIteration

#inclusionary iteration
pattern: "each "x" in "y
result:  InclusionaryIterationCondition
part:    x (<<<NewVariable>>>)
whole:   y (List)

#inclusionary iteration with filter
pattern: "each "x" in "y" "z
result:  InclusionaryIterationCondition
part:    x (<<<NewVariable>>>)
whole:   y (List)
filter:  z (Filter)

#exclusionary iteration
pattern: "no "x" in "y
result:  ExclusionaryIterationCondition
part:    x (<<<NewVariable>>>)
whole:   y (List)

#exclusionary iteration condition with filter
pattern: "no "x" in "w" "z
result:  ExclusionaryIterationCondition
part:    x (<<<NewVariable>>>)
whole:   y (List)
filter:  z (Filter)

#filter
pattern: "with "x" "y
result:  Filter
label:   x (<<<PotentialVariable>>>)
value:   y (Value)

#display
pattern: "DISPLAY "x"."
```

```
value:   x (String)
result:  Display

#add (Integer)
pattern:  "ADD "x" to "y"."
result:   AddToIntegerValue
modifier: x (Integer)
modified: y (Integer)

#add (Float)
pattern:  "ADD "x" to "y"."
result:   AddToFloatValue
modifier: x (Float)
modified: y (Float)

#subtract (Integer)
pattern:  "SUBTRACT "x" from "y"."
result:   SubtractFromIntegerValue
modifier: x (Integer)
modified: y (Integer)

#multiply (Integer)
pattern:  "MULTIPLY "x" into "y"."
result:   MultiplyIntoIntegerValue
modifier: x (Integer)
modified: y (Integer)

#multiply (Float)
pattern:  "MULTIPLY "x" into "y"."
result:   MultiplyIntoFloatValue
modifier: x (Float)
modified: y (Float)

#divide (Integer)
pattern:  "DIVIDE "x" into "y"."
result:   DivideIntoIntegerValue
modifier: x (Integer)
modified: y (Integer)

#divide (Float)
pattern:  "DIVIDE "x" into "y"."
result:   DivideIntoFloatValue
modifier: x (Float)
modified: y (Float)

#random (Integer)
pattern: "GET a random Integer "x"."
result:  GetRandomInteger
value:   x (IntegerRange)

#random (Float)
pattern: "GET a random Float "x"."
result:  GetRandomFloat
```

```
value:   x (FloatRange)

#append (String)
pattern:  "APPEND "x" onto "y"."
result:   AppendOntoStringValue
modifier: x (String)
modified: y (String)

#prepend (String)
pattern:  "PREPEND "x" onto "y"."
result:   PrependOntoStringValue
modifier: x (String)
modified: y (String)

#insert (String)
pattern:  "INSERT "x" at index "y" of "z".
result:   InsertIntoStringValue
modifier: x (String)
position: y (Integer)
modified: z (String)

#get indices (String)
pattern:  "GET the indices of "x" in "y".
result:   GetStringIndices
subvalue: x (String)
object:   y (String)

#determine prefix (String)
pattern:  "DETERMINE whether "x" is the beginning of "y"."
result:   DetermineStringPrefix
subvalue: x (String)
object:   y (String)

#determine suffix (String)
pattern:  "DETERMINE whether "x" is the end of "y"."
result:   DetermineStringSuffix
subvalue: x (String)
object:   y (String)

#get substring (String)
pattern:        "GET the substring between index "x" and index "y" of
"z"."
result:         GetStringSubstring
start_position: x (Integer)
end_position:   y (Integer)
object:         z (String)

#get character (String)
pattern:  "GET the character at index "x" of "y"."
result:   GetCharacterAtStringIndex
position: x (Integer)
object:   y (String)
```

```
#split (String)
pattern:  "SPLIT by "x" on "y"."
result:   SplitString
subvalue: x (String)
object:   y (String)

#replace first instances (String)
pattern:  "REPLACE the first "w" instances of "x" with "y" in "z"."
result:   ReplaceFirstInstancesInStringValue
limit:    w (Integer)
subvalue: x (String)
modifier: y (String)
modified: z (String)

#replace last instances (String)
pattern:  "REPLACE the last "w" instances of "x" with "y" in "z"."
result:   ReplaceLastInstancesInStringValue
limit:    w (Integer)
subvalue: x (String)
modifier: y (String)
modified: z (String)

#replace (String)
pattern:  "REPLACE all instances of "x" with "y" in "z"."
result:   ReplaceAllInstancesInStringValue
subvalue: x (String)
modifier: y (String)
modified: z (String)

#ask
pattern: "ASK "x"."
result:  KeyboardInput
value:   x (String)

#random (List)
pattern: "GET a random element of "x"."
result:  GetRandomListElement
value:   x (List)

#reverse (List)
pattern: "REVERSE "x"."
result:   ReverseList
value:    x (List)

#append (List)
pattern:  "APPEND "x" onto "y"."
result:   AppendOntoListValue
modifier: x (<<<Type of List at y>>>)
modified: y (List)

#append (List)
pattern:  "PREPEND "x" onto "y"."
result:   PrependOntoListValue
```

```
modifier: x (<<<Type of List at y>>>)
modified: y (List)

#insert (List)
pattern:  "INSERT "x" at index "y" of "z".
result:   InsertIntoList
modifier: x (<<<Type of List at z>>>)
position: y (Integer)
modified: z (List)

#remove last instances (List)
pattern:  "REMOVE the first "x" instances of "y" from "z"."
result:   RemoveFirstInstancesFromList
limit:    x (Integer)
modifier: y (<<<Type of List at z>>>)
modified: z (List)

#remove last instances (List)
pattern:  "REMOVE the last "x" instances of "y" from "z"."
result:   RemoveLastInstancesFromList
limit:    x (Integer)
modifier: y (<<<Type of List at z>>>)
modified: z (List)

#remove all (List)
pattern:  "REMOVE all instances of "x" from "y"."
result:   RemoveAllInstancesFromList
modifier: x (<<<Type of List at y>>>)
modified: y (List)

#get element (List)
pattern:  "GET the element at index "x" of "y"."
result:   GetElementAtListIndex
position: x (Integer)
object:   y (List)

#remove element (List)
pattern:  "REMOVE the element at index"x" of "y"."
result:   RemoveElementAtListIndex
position: x (Integer)
object:   y (List)

#get sublist (List)
pattern: "GET the sublist between index "x" and index "y" of "z"."
result:  GetListSublist
start_position: x (Integer)
end_position:   y (Integer)
object:         z (List)

#get indices (List)
pattern:  "GET the indices of "x" in "y"."
result:   GetListIndices
subvalue: x (<<<Type of List at y>>>)
```

```
object:   y (List)

#range (Integer)
pattern:        "between "x" and "y
result:         IntegerRange
start_position: x (Integer)
end_position:   y (Integer)

#range (Float)
pattern:        "between "x" and "y
result:         FloatRange
start_position: x (Float)
end_position:   y (Float)
```

Appendix B: Experiment

Data Release Form

Please read the entirety of this form, and sign only if you agree to the rights it provides you.

Rights of the Participant

The rights below are the rights of every person who is asked to be in a study. In participating as a human subject I have the following rights:

- To be told what the study is trying to find out;
- To be allowed to ask any questions concerning the study both before agreeing to be involved and during the course of the study;
- To refuse to participate at all or to change my mind about participation after the study is started;
- To receive a copy of the signed and dated consent form;
- To be free of pressure when considering whether I wish to agree to be in the study; and
- To have your name withheld from any publication in which the results of this experiment are given;

Release of Data (Participant)

I hereby give permission for any answers collected during the course this experiment to be used in whole or as part of a statistic in publication, in which my real name will be changed to either a false name or a number to mask my identity, and that my participation in this experiment will not be disclosed.

Print Name: _____

Sign Name: _____

Date: _____

Promise to Uphold the Rights of the Participant (Experimenter)

I hereby promise to uphold the Rights of the Participant as enumerated above. I promise not to include the subject's real name in any publication of the results and will not disclose the participant's participation, personal information, or results (as tied to the participant's name) to anyone without the participant's permission.

Print Name: _____

Sign Name: _____

Date: _____

Instructions

Please read all instructions before looking at Game 1 or Game 2.

Introduction

First, thank you for participating in this experiment. By doing so, you are helping the programming language design community to understand the needs of the first-time programmer.

The experiment in which you are about to participate is designed to evaluate the learnability of two programming languages. *Please understand that you are not being evaluated.* Every person participating in the experiment has no previous programming experience, so you are not expected to know any programming theory or the syntax of either of the languages being tested. The experiment is designed to understand how much someone with no programming experience can understand simply by reading the code. You are not expected to understand all or even most of the code.

Explanation of Terms

In the next section, you will be presented with the code for two board or card games. Before you begin, there are a few terms related to these games that you should look at.

Object The objects you will be looking at today are code representations of real-life objects. Since you will be looking at board games, you will likely be familiar with these objects. There are physical objects, like dice, boards, cards, and players. There are also "abstract" objects, like a move, which doesn't physically exist. Take a moment, and think back to when you learned the definition of a noun. Likely, you learned it to be a person, place, thing, or idea. The player is the person, the things are boards and cards, and the idea is a move.

Variable A variable is simply a way to give a name to an object. Without them, we would create a bunch of objects and then never be able to find them again. Look at the room around you. It's full of objects. We can talk about them and point to them. However, if we didn't have names for anything, and we didn't have the ability to point (computers don't), how could we talk about an object? We couldn't. So a

variable is just a unique name that points out a specific object when you want to use it later. It is composed of two parts, the name and the object you want to associate with the name. When we "define"a variable, we give it both parts; when we "reference" it, we just ask for the name, and it gives us the object.

Function A function is an action. When we "define" a function, we specify the steps of the action. When we "call" the function, we actually do the steps. Actions typically involve objects. Objects can do things, and things can be done to them. In programming, even objects that are inanimate in real life, like a card, can do things. We also have the program itself, which acts as a divine being. Even though it's not an object, it can do things to objects. When we call a function on objects, the function (action) does something with them or to them. A function can also "return" a value, which is the result of doing the action.

Line Number This is exactly what it sounds like. It's the number of a line in the code. You will see them on the left-hand side of the page. They are not part of the code itself. They are just for your convenience when you answer the questions after each game.

Steps

1. Fill out the entry survey on the next page (Type your answers in a separate document if you are using the electronic version).
2. Begin with either Game 1 or Game 2 (you will do the one you don't select later).
3. If you selected Game 1, look at Rules.py (Rules.pdf in the Game 1 folder for the electronic version). If you selected Game 2, look at Rules.bgl (Rules.pdf in the Game 2 folder).
4. You will see line numbers on the left-hand side and then code after them. Read the code from top to bottom *Once again, you are not expected to understand all or even most of this*.
    1. Try to identify the following (mentally, you don't need to mark them). They look different in one language than they do in another.
        1. variable declarations
        2. function declarations
        3. function calls
        4. object creation

2. Make note of the kinds of code that you don't understand as you go through
3. Take a moment to relax, then answer the questions at the end of the Game.

5. Take a moment to relax, then repeat steps 1-4 for the other game.
6. Fill out the exit survey that is on the page after the entry survey.

Tips

- If you can't figure out part or all of a line of code, skip it, and if you come across a similar line later, try to figure out that line and then the line from earlier. If you still can't figure it out, move on.
- Not all of the functions are declared in the Rules file, so the code for all the objects that the game uses are included. It is possible for the Rules file or another object to reference a variable or call a function that is declared in the code of another object.
- Take a break before going on to the second game. It can be a lot to take in.
- Relax. You aren't being evaluated. The experiment is to determine what you have trouble with so that the language development community can make languages that are easier to learn.

Entry Survey

1. Have you ever programmed before, excluding HTML? (circle one)
   Yes No

2. What is your age? ____

3. What year did you graduate college (or still attending)? ____

4. What were your majors and/or tracks in college? ____

5. How well do you feel that you understand the following concepts? (Circle one of each)
   1. Object: Not at all A little Ok Perfectly
   2. Variable: Not at all A little Ok Perfectly
   3. Function: Not at all A little Ok Perfectly

6. With which of the following games are you familiar? (circle all the apply)
   1. Solitaire
   2. Freecell
   3. Card Pair Matching
   4. Checkers
   5. Chess
   6. 4 in a Row
   7. Othello

Exit Survey

1. How well do you feel that you understand the following concepts now that you have been exposed to them? (Circle one of each)
    1. Object: Not at all A little Ok Perfectly
    2. Variable: Not at all A little Ok Perfectly
    3. Function: Not at all A little Ok Perfectly

2. Which game did you look at first? (Circle 1)
   Game 1 Game 2

3. Write anything else you would like for the experimenters to know (optional).

Game 1

Feel free to remove the staple and spread out the pages

Rules.py

```
1 from Player import Player
2 from Pile import Pile
3 from Card import Card
45
players=[]
6 pile=Pile()
78
def game_is_won():
9 if len(pile.cards)==0: return True
10 return False
11
12 def get_winner():
13 player_1_pars_num=len(players[0].hand)
14 player_2_pars_num=len(players[1].hand)
15
16 if player_1_pars_num>player_2_pars_num: return players[0]
17 elif player_1_pars_num<player_2_pars_num: return players[1]
18 else: return None
19
20 players.append(Player("Amanda"))
21 players.append(Player("Brett"))
22
23 for value in range(25):
24 pile.cards.append(Card(value))
25 pile.cards.append(Card(value))
26
27 pile.shuffle()
28
29 while not game_is_won():
30 for player in players:
31 if len(pile.cards)==0: break
32
33 first_card_index,second_card_index=player.get_move(pile)
34 first_card=pile.get_card_at_index(first_card_index)
35 first_card.flip()
36 second_card=pile.get_card_at_index(second_card_index)
37 second_card.flip()
38
39 if first_card.value==second_card.value:
40 if first_card_index>second_card_index:
41 first_card=pile.remove_card_at_index(first_card_index)
42 second_card=pile.remove_card_at_index(second_card_index)
43 else:
44 second_card=pile.remove_card_at_index(second_card_index)
45 first_card=pile.remove_card_at_index(first_card_index)
46
47 player.receive_card_pair(first_card,second_card)
48 else:
49 first_card.flip()
50 second_card.flip()
51
52 winner=get_winner()
53 if winner==None: print "It was a tie!"
54 else:
55 print winner.name+" won!"
56 print `len(winner.hand)/2`+" pairs:\t"+`winner.hand`
```

Card.py

```
class Card(object):

def __init__(self,value,state="face-down"):
self.value=value
self.state=state

def flip(self):
if self.state=="face-down": self.state="face-up"
elif self.state=="face-up": self.state="face-down"

def __repr__(self):
return `self.value`
```

Pile.py

```
class Pile(object):
cards=[]

def shuffle(self):
pass

def get_card_at_index(self,index):
if index<0 or index>len(self.cards)-1: return None
return self.cards[index]

def remove_card_at_index(self,index):
if index<0 or index>len(self.cards)-1: return None
return self.cards.pop(index)
```

Player.py

```
import random

class Player(object):

def __init__(self,name):
self.hand=[]
self.name=name

def get_move(self,pile):

#start unimportant lines (don't read them)
first_card_index=random.randint(0,len(pile.cards)-1)

while True:
second_card_index=random.randint(0,len(pile.cards)-1)
if second_card_index!=first_card_index: break
#end unimportant lines

return (first_card_index,second_card_index)

def receive_card_pair(self,card1,card2):
self.hand.append(card1)
self.hand.append(card2)

```

Game 1 Questions

Instructions

Answer as many questions as you can. When we ask for a "type of thing", we are looking for one of the following: variable, function, object.

1. Which of the following games does it implement? If you don't know, fill out 8 with the goal of the game.
    1. Solitaire
    2. Freecell
    3. Card Pair Matching
    4. Checkers
    5. Chess
    6. 4 in a Row
    7. Othello
    8. _____
       _____
       _____
       _____

2. What is the type of thing being declared on line 6 of Rules.py (Rules.pdf)
   _____
   _____

3. Also on line 6, what is type of thing is "Pile" (not "pile"), and in which file is it defined?
   _____
   _____

4. What is the type of thing being declared on line 8 of Rules.py (Rules.pdf), and what is the purpose of lines 8-10?
   _____
   _____
   _____
   _____
   _____

5. Look at line 27 of Rules.py (Rules.pdf). In which file is shuffle defined?

_____
_____

6. There is a portion of the code in Rules.py (Rules.pdf) that sets the game up to be played and a portion that actually plays it. What is the range of line numbers where the game is actually played?
   Lines ___ to ___

7. Say we want to add a variable that we are going to use only in lines 23 through 25 of Rules.py (Rules.pdf). On which line should we define the variable?
   Line ___

8. Say that the variable from question 7 is named "counter", and we want its starting value to be 0. Write the variable declaration that does this.
   _____
   _____

9. Say that want to print the string "GOOD" if the integer 1 is greater than the value of the counter from question 8. Write the line of code that would do that.
   _____
   _____

10. Rewrite lines 24-25 from Rules.py, appends the integer 1 to players twice:
    24
    _____
    _____
    25
    _____
    _____

11. On which line of Rules.py (Rules.pdf) does the game check if there is a winner? There are 2 right answers here. Choose 1.
    Line ___

Game 2

Feel free to remove the staple and spread out the pages

Rules.bgl

```
1 VARIABLES:
2 LET the players be a new List of type Game_Player.
3 LET the board be a new Board.
4
5
6
7 FUNCTIONS:
8 CHECK for winning conditions:
9 LET the return type be a Boolean.
10
11 LET the full_goals_number be 0.
12 if the size of the cards of the spades_goal of the board is 13,
13 ADD 1 to the full_goals_number.
14 if the size of the cards of the hearts_goal of the board is 13,
15 ADD 1 to the full_goals_number.
16 if the size of the cards of the diamonds_goal of the board is 13,
17 ADD 1 to the full_goals_number.
18 if the size of the cards of the clubs_goal of the board is 13,
19 ADD 1 to the full_goals_number.
20
21 if the full_goals_number is < 4,
22 SUCCEED.
23 RETURN false.
24 otherwise,
25 SUCCEED.
26 RETURN true.
27
28
29
30 DEAL the top card to [Deck] deck_2 from [Deck] deck_1:
31 LET the return type be Nothing.
32
33 if deck_2 is Nothing or deck_1 is Nothing, FAIL.
34 if the size of the cards of deck_1 is 0, FAIL.
35
36 GET the top card from deck_1.
37 ADD the result of GET to the top of deck_2.
38 SUCCEED.
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
```

```
65
66
67
68
69
70
71 SETUP:
72 LET the suits be a List of "spades", "hearts", "clubs", and "diamonds".
73
74 FOR each suit in the suits,
75 FOR each value between 1 and 13,
76 let the card be a new Playing_Card.
77 let the suit of the card be the suit.
78 let the value of the card be the value.
79 if the suit is "spades" or "clubs", LET the color of the card be "black".
80 otherwise, if the suit is "hearts" or "diamonds", LET the color of the card be "red".
81 ADD the card to the main_deck of the board.
82
83 SHUFFLE the main_deck of the board.
84
85 LET the number_of_decks be 7.
86
87 FOR each deck_number between 1 and the number_of_decks,
88 ADD a new Deck to the decks of the board.
89
90 UNTIL the number_of_decks is 0,
91 LET the skip_number be 7 - the number_of_decks.
92 FOR each deck_number between 1 and the number_of_decks,
93 DEAL the top card to the deck from the main_deck of the board.
94 SUBTRACT 1 from the number_of_decks.
95
96 FOR each deck in decks of board, FLIP the top of the deck.
97
98 DEAL the top card to the main_deck_discard of the board from the main_deck of the
board .
99 FLIP the Card.
100
101 ADD a new Game_Player to the players.
102
103
104
105 GAMEPLAY:
106 CHECK for winning conditions.
107 if the result of CHECK is true,
108 DISPLAY "You Won!"
109 EXIT.
110
111 GET move from the player.
112 LET the move be the result of GET.
113
114 if the category of the move is "card flip",
115 FLIP the top card of the start_deck.
116
117 otherwise, if the category of the move is "main deck deal",
118 DEAL the top card to the main_deck_discard of the board from the main_deck of the
board.
119 FLIP the Card.
120
121 otherwise, if the category of the move is "card transfer",
122 LET the start_deck be the start_deck of the move.
123 LET the end_deck be the end_deck of the move.
124 LET the number_of_cards be the number_of_cards of the move.
125
126 GET the top number_of_cards cards from the start_deck.
127 LET the cards be the result of GET.
128
```

```
129 ADD the cards to the top of the to_deck.
```

Board.bgl

```
1 a Board is a type of Surface.
2
3 VARIABLES:
4 LET the decks be a new List of type Deck.
5 LET the main_deck be new Deck.
6 LET the main_deck_discard be a new Deck.
7 LET the spades_goal be a new Deck.
8 LET the clubs_goal be a new Deck.
9 LET the hearts_goal be a new Deck.
10 LET the diamonds_goal be a new Deck.
11
12 FUNCTIONS:
13 PLACE [Playing_Card] card into the goal of this Board:
14 LET the return type be Nothing.
15
16 if the suit of the card is "Spades",
17 MOVE the card to the spades_goal of this Board.
18 SUCCEED.
19 otherwise, if the suit of the card is "Hearts",
20 MOVE the card to the hearts_goal of this Board.
21 SUCCEED.
22 otherwise, if the suit of the card is "Clubs",
23 MOVE the card to the clubs_goal of this Board.
24 SUCCEED.
25 otherwise, if the suit of the card is "Diamonds",
26 MOVE the card to the diamonds_goal of this Board.
27 SUCCEED.
28 otherwise, FAIL.
```

Deck.bgl

```
1
1 a Deck is a type of Object.
2
3 VARIABLES:
4 LET the cards be a new List of type Card.
56
FUNCTIONS:
7 GET the top card from this Deck:
8 LET the return type be a Card.
9
10 if the size of the cards is 0,
11 FAIL.
12 RETURN Nothing.
13
14 GET the element at index 0 of the cards.
15 REMOVE the element at index 0 of the cards.
16 SUCCEED.
17 RETURN the result of GET.
18
19
20 GET the top [Integer] number_of_cards cards from this Deck:
21 LET the return type be a Card.
22
23 LET the return_list be a new List of type Card.
24 FOR each card_number between 1 and the number_of_cards,
25 GET the top card from this Deck.
26
27 if GET failed,
28 FAIL.
29 RETURN Nothing.
30
31 ADD the result of GET to the return_list.
32
```

```
33 SUCCEED.
34 RETURN the return_list.
35
36
37 SHUFFLE this Deck:
38 LET the return type be Nothing.
39
40 if the cards is Nothing, FAIL.
41
42 LET the cards_copy be a copy of the cards.
43 GET a random Integer between 0 and the size of the cards.
44 LET the remove_index be the result of GET.
45 GET the element at index remove_index of the cards.
46 REMOVE the element at index remove_index of the cards.
47 APPEND the result of GET onto the cards_copy.
48
49 SUCCEED.
50
51
52 ADD [Card] card to the bottom this Deck:
53 LET the return type be Nothing.
54
55 if the card is Nothing, FAIL.
56
57 APPEND the card to the cards.
58 SUCCEED.
59
60
61 ADD [List:Card] cards to the top of this Deck:
62 LET the return type be Nothing.
63
64 if the cards is Nothing, FAIL.
65
66 REVERSE cards.
67 FOR each card in the cards,
```

Deck.bgl

```
2
68 PREPEND the card onto the cards.
69
70 SUCCEED.
71
72
73 FLIP the top card of this Deck:
74 LET the return type be Nothing.
75
76 if the size of the cards is 0, FAIL.
77
78 GET the element at index 0 of the cards.
79 FLIP the result of GET.
80
81 SUCCEED.
```

Game_Player.bgl

```
1 a Game_Player is a type of Player.
2
3 FUNCTIONS:
4 GET a move from this PLayer:
5 LET the return type be a Move.
67
#start unimportant lines (don't read)
8 ...PRESENT legal moves to physical player and get input... (not actually a statement)
9 #end unimportant lines
10
11 SUCCEED.
12 RETURN move.
```

Move.bgl

```
1 a Move is a type of Object.
2
3 VARIABLES:
4 LET the category be a type of String.
5 LET the start_deck be a type of Deck.
6 LET the start_deck be a type of Deck.
7 LET the number_of_cards be type of Integer.
```

Playing_Card.bgl

```
1 a Playing_Card is a type of Card.
2
3 VARIABLES:
4 LET the state be "face-down".
5 LET the suit be a type of String.
6 LET the value be a type of Integer.
7 LET the color be a type of String.
89
FUNCTIONS:
10 FLIP this Card:
11 if the state is "face-down", update the state to be "face-up".
12 otherwise, if the state is "face-up", update the state to be "face-down".
```

References

[1] The algorithm design manual, volume 1.

[2] Arrays.

[3] Class boolean.

[4] Class oat.

[5] Class integer.

[6] Class list.

[7] Class string.

[8] Classes.

[9] Controlling access to members of a class.

[10] Declaring classes.

[11] Declaring variables.

[12] Dening functions.

[13] Java arrays.

[14] The java language: An overview.

[15] The java tutorials.

[16] Monopoly property trading game.

[17] nderstanding instance and class members.

[18] Open source scripting languages in java.

[19] Original memory game.

[20] Primitive data types.

[21] Returning a value from a method.

[22] Risk game.

[23] Shells and shell scripts.

[24] Strings.

[25] Strong typing.

[26] Using the this keyword.

[27] What? it wasn't supposed to do that! tracking down logic errors.

[28] What went wrong? nding and xing errors through debugging.

[29] metaphor, 2010.

[30] Serge Abiteboul and Victor Vianu. Expressive power of query languages. Research Report RR-1587, INRIA, 1992.

[31] Azad Ali and Frederick G. Kohun. Considerations for selecting a programming language to teach perspective teachers. In Proceedings of Alice Symposium, June 2009.

[32] Bruce W. Ballard and Alan W. Biermann. Programming in natural language:
\nlc" as a prototype. In ACM 79: Proceedings of the 1979 annual conference, pages 228{237, New York, NY, USA, 1979. ACM.

[33] Alan W. Biermann and Bruce W. Ballard. Toward natural language computation. American Journal of Computational Linguistics, 6(2):71{86, April 1980.

[34] Benedict Du Boulay. Some diculties of learning to program. Journal of
Educational Computing Research, 2(1):57{73, 1986.

[35] Benedict Du Boulay, Tim O'Shea, and John Monk. The black box inside the glass box: presenting computing concepts to novices. Int. J. Hum.-Comput. Stud., 51(2):265{277, 1999.

[36] Edsger W. Dijkstra. On the cruelty of really teaching computing science.

[37] Edsger W. Dijkstra. Some comments on the aims of mirfac. Commun. ACM, 7(3):190, 1964.

[38] Edsger W. Dijkstra. On the foolishness of "natural language programming". In Program Construction, pages 51{53, 1978.

[39] Matthias Felleisen. On the expressive power of programming languages. In Science of Computer Programming, pages 134{151. Springer-Verlag, 1990.

[40] Kathleen M. Galotti and William F. Gangon III. What non-programmers know about programming: Natural language procedure specication. International Journal of Man-Machine Studies, 22(1):1{10, January 1985.

[41] H. J. Gawlik. Mirfac: a compiler based on standard mathematical notation and plain english. Commun. ACM, 6(9):545{547, 1963.

[42] H. J. Gawlik. Mirfac: a reply to professor dijkstra. Commun. ACM, 7(10):571, 1964.

[43] H. P. Grice. Logic and Conversation, chapter 2, pages 121{133. Readings in language and mind. Wiley-Blackwell, 1996.

[44] Diwaker Gupta. What is a good rst programming language?, Summer 2004.

[45] Mark Halpern. Foundations of the case for natural-language programming. In AFIPS '66 (Fall): Proceedings of the November 7-10, 1966, fall joint computer conference, pages 639{649, New York, NY, USA, 1966. ACM.

[46] Brian Harvey. Symbolic Computing, volume 1 of Computer Science Logo Style. MIT Press, 2 edition, February 1997.

[47] C. A. R. Hoare. Essays in computing science. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[48] Neville Holmes. The case for perspicuous programming.

[49] K. N. King. The case for java as a rst language. In Proceedings of the 35th Annual ACM Southeast Conference, April 1997.

[50] P□aivi Kinnunen and Lauri Malmi. Why students drop out cs1 course? In ICER '06: Proceedings of the second international workshop on Computing education research, pages 97{108, New York, NY, USA, 2006. ACM.

[51] M. Klerer and J. May. A user oriented programming language. The Computer Journal, 8(2), 1965.

[52] Donald E. Knuth. Semantics of context-free languages. Mathematical Systems Theory, 2(2):127{145, 1968.

[53] Donald E. Knuth. The genesis of attribute grammars. In WAGA: Proceedings of the international conference on Attribute grammars and their applications, pages 1{12, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[54] Christian Koch. On the benets of interrelating computer science and the humanities: The case of metaphor. Computers and the Humanities, 25(5):289{295, 1991.

[55] Tim Korson and John D. McGregor. Understanding object-oriented: a unifying paradigm. Commun. ACM, 33(9):40{60, 1990.

[56] Zoltan K□ovecses. Metaphor: a practical introduction. Oxford University Press, 2002.

[57] George Lako and Mark Johnson. Metaphors We Live By. University of

Chicago Press, 1980.

[58] Henry Lieberman and Hugo Liu. Feasibility studies for programming in natural language. Kluwer Academic Publishers, 2007.

[59] Hugo Liu and Henry Lieberman. English: The lightest weight programming language of them all (article). Lightweight Languages, December 2004.

[60] Hugo Liu and Henry Lieberman. English: The lightest weight programming language of them all (slides), 2004.

[61] Hugo Liu and Henry Lieberman. Toward a programmatic semantics of natural language. In IEEE Computer Society, IEEE Symposium on Visual Languages and Human Centric Computing, 2004.

[62] Hugo Liu and Henry Lieberman. Metafor: Visualizing stories as code, January 2005.

[63] Hugo Liu and Henry Lieberman. Programmatic semantics for natural language interfaces. In CHI '05: CHI '05 extended abstracts on Human factors in computing systems, pages 1597{1600, New York, NY, USA, 2005. ACM.

[64] Linda McIver and Damian Conway. Seven deadly sins of introductory programming language design. In SEEP '96: Proceedings of the 1996 International Conference on Software Engineering: Education and Practice (SE:EP '96), page 309, Washington, DC, USA, 1996. IEEE Computer Society.

[65] Don Merten and Gary Schwartz. Metaphor and self: Symbolic process in everyday life. American Anthropologist, 84(4):796{810, 1982.

[66] Rada Mihalcea, Hugo Liu, and Henry Lieberman. NLP (Natural Language Processing) for NLP (Natural Language Programming), volume 3878/2006 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 978-3-540-32205-4, 2006.

[67] Iain Milne and Glenn Rowe. Diculties in Learning and Teaching Programming Views of Students and Tutors. Kluwer Academic Publishers, 2002.

[68] John C. Mitchell. Concepts in Programming Languages. Cambridge University Press, 2003.

[69] John F. Pane, Chotirat \Ann Ratanamahatana, and Brad A. Myers. Studying the language and structure in non-programmers solutions to programming problems. International Journal of Human-Computer Studies, 54(2):237{264, February 2001.

[70] Seymour Papert. Educational computing: How are we doing? T.H.E, pages 78{80, June 1997.

[71] Seymour Papert. Logo philosophy and implementation, 1999.

[72] Kevin R. Parker, Joseph T. Chao, Thomas A. Ottaway, and Jane Chang. A formal language selection process for introductory programming courses. Journal of Information Technology Education, 6, 2006.

[73] R.W. Paul. Critical thinking. Foundation for Critical Thinking Santa Rosa, CA, 1993.

[74] Alan Perlis. Epigrams on programming. SIGPLAN Notices, 17(9):7{13, September 1982.

[75] S. R. Petrick. On natural language based computer systems. IBM J. Res. Dev., 20(4):314{325, 1976.

[76] Joseph A. Raelin. A model of work-based learning. Organization Science, 8(6):563{578, 1997.

[77] Richard J. Reid. The object oriented paradigm in cs 1. In SIGCSE '93: Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education, pages 265{269, New York, NY, USA, 1993. ACM.

[78] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. Computer Science Education, 13:137{172, 2003.

[79] Jean E. Sammet. The use of english as a programming language. Commun. ACM, 9(3):228{230, 1966.

[80] G. Michael Schneider. The introductory programming course in computer
science: ten principles. In SIGCSE '78: Papers of the SIGCSE/CSA technical symposium on Computer science education, pages 107{114, New York, NY, USA, 1978. ACM.

[81] Steven S. Skiena. The algorithm design manual, volume 1. Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[82] Ken Slonneger. Attribute grammars, 2000.

[83] David Ungar and Randall B. Smith. Self: The power of simplicity. SIG-PLAN Not., 22(12):227{242, 1987.

[84] Johan van Benthem. Economics and language. In Ariel Rubinstein, editor, Economics and Language, pages 93{107. Cambridge University Press, 2000.

[85] Leon E. Winslow. Programming pedagogy|a psychological overview. SIGCSE Bull., 28(3):17{22, 1996.