Reusing Code by Reasoning About its Purpose

by

Kenneth Charles Arnold

B.S., Cornell University (2007)

Submitted to the Program in Media Arts and Sciences, School of
Architecture and Planning
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2010

Author . . . .
Program in Media Arts and Sciences, School of Architecture and
Planning
January 15, 2010

Certified by. . . . .
Henry Lieberman
Research Scientist
Thesis Supervisor

Accepted by . . . .
Deb K. Roy
Chair, Academic Program in Media Arts and Sciences

Reusing Code by Reasoning About its Purpose

by

Kenneth Charles Arnold

Abstract

When programmers face unfamiliar or challenging tasks, code written by others could give them inspiration or reusable pieces. But how can they find code appropriate for their goals? This thesis describes a programming interface, called Zones, that connects code with descriptions of purpose, encouraging annotation, sharing, and reuse of code. The backend, called ProcedureSpace, reasons jointly over both the words that people used to describe code fragments and syntactic features derived from static analysis of that code to enable searching for code given purpose descriptions or vice versa. It uses a technique called Bridge Blending to do joint inference across data of many types, including using domain-specific and commonsense background knowledge to help understand different ways of describing goals. Since Zones uses the same interface for searching as for annotating, users can leave searches around as annotations, even if the search fails, which helps the system learn from user interaction. This thesis describes the design, implementation, and evaluation of the Zones and ProcedureSpace system, showing that reasoning jointly over natural language and programming language helps programmers reuse code.

Reusing Code by Reasoning About its Purpose

by

Kenneth Charles Arnold

The following served as a reader for this thesis:

Thesis Reader . . . .

Mitchel Resnick

LEGO Papert Professor of Learning Research

MIT Media Laboratory

Reusing Code by Reasoning About its Purpose

by

Kenneth Charles Arnold

The following served as a reader for this thesis:

Thesis Reader . . . .
Robert C. Miller
Associate Professor
MIT Department of Electrical Engineering and Computer Science

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Programs start life in a very different form than the programming languages we often use to express them—as goals, intentions, or disconnected incomplete thoughts. "Programming," then, is not simply about feeding procedure[1] to computer in minimal time. It's a process of translating and refining ideas and strategies into executable artifacts that partially embody our thoughts and fulfill our goals. We gradually develop a computational "grounding" of our intentions and strategies in a computer language. As we write, debug, and extend code, we learn to map our subgoals and intentions into code, and vice versa.

When a programmer is fluent in a programming environment (a programming language plus various frameworks and libraries) that is well adapted to solve a problem that the programmer is familiar with, the resulting program can have a poetic elegance that defies translation even into natural language. But this idyllic situation is rare in practice: few programmers are so fluent with all of the environments they may have to deal with, and classes of problems develop far faster than even special-purpose programming environments can adapt to them. So as long as programmers continue to tackle new kinds of problems (and surely to stop would bore them!), even the most elegant, expressive, or clear programming language will never be the single ideal representation of procedural knowledge. And some of the same qualities that make

---

[1]Since programs are just data, they can represent knowledge in a much more flexible way than the term "procedure" implies, but that is beyond the scope of this thesis.

code elegant for fluent programmers—compactness, expressiveness, or the pervasive embrace of powerful ideas—make it impenetrable to a novice's understanding.

While we should continue to pursue ever more powerful representations of procedural knowledge, we must also seek ways of connecting those representations with more accessible (if less powerful) representations. And perhaps the most accessible representation is a description of the purpose of each part of the code in natural language.

1.1   Natural Language Code Search

We often learn from others' examples, especially if the task is difficult. Programming has historically been such a task. Some programmers like to "take apart" interesting programs to see how they work, which provides concepts and examples that become useful when they write their own programs. Others post interesting code examples and tutorials in forums and blogs. And some programmers search for projects or modules that might have code they could copy and paste.

Whether within a team or increasingly in a geographically distributed open-source community, programmers often want to find some code that someone else wrote that accomplishes some goal. In the opposite direction, they may wonder what purpose some code they're examining might serve. For both problems, programmers often turn to search engines, both general and code-specific. Sometimes they're looking for a package of functionality[21], sometimes just a few statements[4]. Often the code exists somewhere, if the programmer can just make the right query.

How can a programming environment find relevant code to reuse? How can programmers communicate their goals to the computer in the first place? Both questions admit many responses, from contracts to keywords, unit tests to interactive dialogues. Formalized semantics, such as design contracts and unit tests, can give accurate results, but require a significant investment in programmer discipline. Popular code search engines like Google Code Search[10] employ keyword searches, which usually return only exact matches. Experienced programmers know how to use related

keywords or code-specific query refinements, but novices lack this knowledge, and even experts find some queries difficult to formulate.

This thesis presents an integrated code reuse system that uses natural-language descriptions of code purpose. Such descriptions are not always possible (e.g., for math formulas or intricate algorithms), but when they do apply, they are potentially the easiest for the programmer to provide. Though natural language descriptions may be more difficult to process, they present a much lower barrier to entry, and thus a high cost-benefit ratio, to the programmer.

1.2   Blending Syntax and Semantics

To search based on a natural language description of a goal, the system must identify to what degree the goal description might apply to any given fragment of code. The question is difficult even for formal descriptions, which have restricted syntax and vocabulary. If every code fragment were annotated with one or more natural language goal descriptions, the problem would become finding which statements describe the same goal as the one searched for. Variations within these descriptions include differences in syntax, vocabulary, and level of description (e.g., "arrow keys" vs. "move"), making the matching process difficult. Worse, not all code will be annotated; in practice, less than 1% may be. So to find matching code that was not explicitly annotated, a natural language goal must be matched to a programming language implementation.

The ProcedureSpace method presented in this thesis addresses this problem for the first time by learning relationships between code structures and the words that people use to describe it. Figure 1-1 shows the basic idea: words and phrases in the natural language description of the a programmer's goal are directly associated with code that they or another programmer wrote to accomplish that goal, through a process described in the next section. But ProcedureSpace knows additional information about the code (characteristics of its structure) and about the natural language words and phrases (commonsense and domain-specific background knowledge). A process

natural language
descriptions

background knowledge

dispatch
search

You would search if you
wanted to find.

```
for rule in rules:
    if rule.matches(input):
        return rule.process(input)
```

```
for > if > return
if > matches > return
f1(input) > f2(input)
```
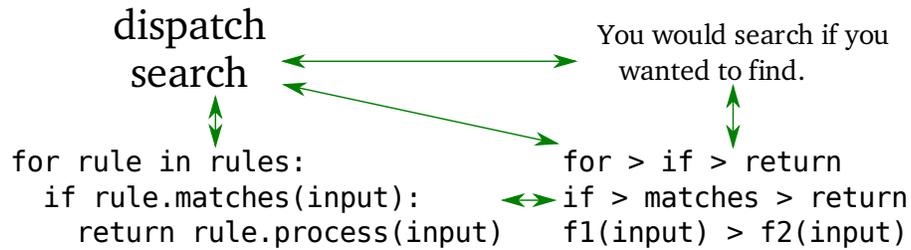
code fragments

static analysis

Figure 1-1: Diagram of a representation of "search" that incorporates both natural language and programming language knowledge

called Blending (presented in background section 3.2.4) enables ProcedureSpace to reason jointly over these different types of knowledge to learn relationships between words and code structures.

This process of connecting *what* programmers want to do with *how* they accomplish it enables programmers to find code for a goal (or goals for some code), integrate it into their program, and share the results to help future programmers. It relies on understanding both natural language and code. Code is more than the bag of symbols it contains; we need to understand how those symbols are structured. Likewise, natural language is more than a bag of words; we need to understand how words relate. This understanding often requires background knowledge about the program's domain: for the simple video games that are common in the corpus of this thesis, it helps to know that "arrow keys are used for moving." Or it may require general commonsense knowledge, such as "balls can bounce." These relationships form the background knowledge that ProcedureSpace uses to reason about code and goals together.

1.3   Organization

Chapter 2 presents Zones, an intelligent goal-sharing interface integrated into the Scratch[28, 29] programming environment that helps programmers find and integrate

code for a given purpose, or identify what unfamiliar code might be for. Then, after covering technical background about Scratch and the Blending process in chapter 3, chapter 4 describes ProcedureSpace, which powers Zones queries by reasoning jointly over code and natural language. It combines purpose annotations, keywords, code features, and natural-language background knowledge (both general and domain-specific) into a unified space organized around the relevance of code examples and characteristics to particular purposes. Chapter 5 describes users' experience with the system, showing how Zones, powered by ProcedureSpace, enables more meaningful code searches to facilitate code reuse. Chapter 6 discusses related work. Finally, chapter 7 summarizes the contributions of this thesis.

Chapter 2

Zones: Active Scoped Comments

Comments would be much more useful if they actually helped you find and reuse the code you were about to write. A Zone is a comment that links a fragment of code with a brief natural language description of its purpose. You draw a box around a scoped section of code, or create an empty box, to create a Zone.[1] You can attach a comment, usually a single line in natural language, that answers the question, "What's this for?"—that is, what goal does that section of code accomplish? Conciseness is more useful than exactness. You can edit comments or code in either order.

But unlike normal comments, Zones are active. You can simply type an English statement of purpose, then the system searches for code that accomplishes that purpose. Alternatively, you can type some code and then search, which means: find code that accomplishes the purpose that this code does. At minimum, these code or description examples provide reminders of syntactic elements, factors to consider, and common practice. At best, a snippet from another project works in this context with minimal modification, and can simply be transplanted.

---

[1]Kaelbling asserted back in 1988 that comments should be scoped[16]. Authors of documentation tools have heeded his advice, treating comments before major structural units like classes and functions as documentation for those units. Although some editors allow "code folding" based on comments at other levels of granularity, I am not aware of attempts to utilize local comment knowledge for purposes beyond navigation.
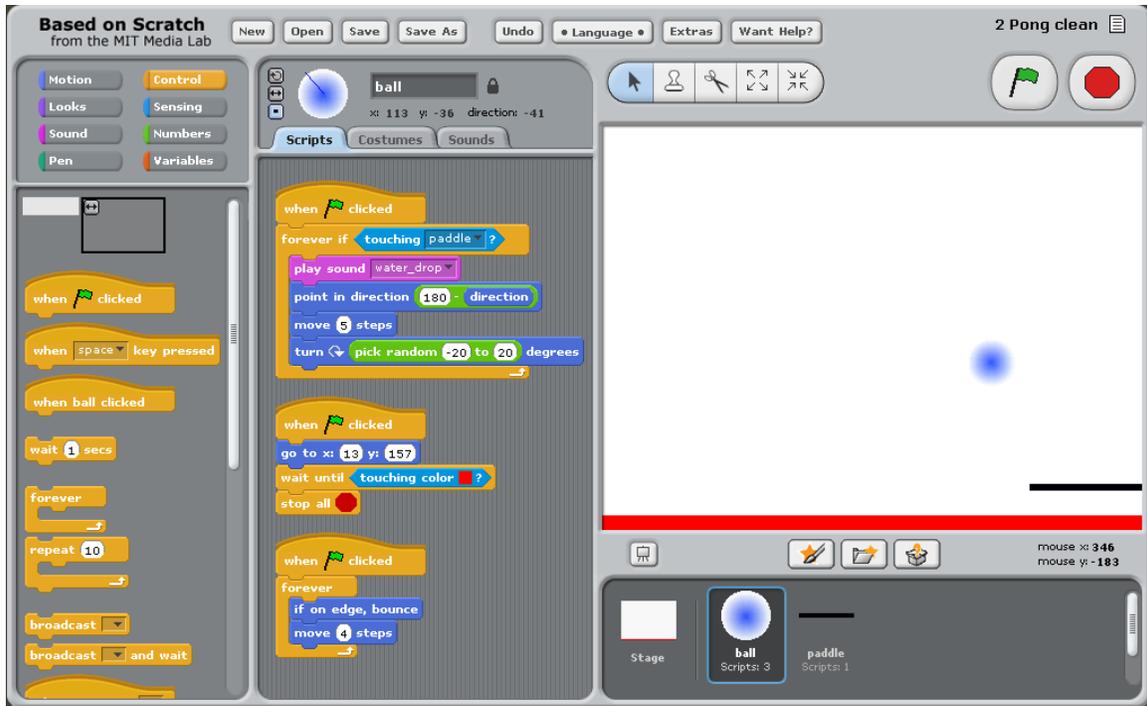
Figure 2-1: Scratch IDE showing the "Pong" example project. The middle pane shows the code for the currently selected sprite, in this case, the ball.

2.1 Scenario

Let's start with a scenario of how Zones can be used. Scratch, like many programming environments, is distributed with a collection of sample projects that a programmer can modify to create their own project. One of those projects is a simple Pong game[2]. Imagine a new programmer exploring this example project for the first time. The Scratch IDE (Figure 2-1) shows the graphics and code. Table 2-1 shows the code fragments (called "scripts" in Scratch) from both sprites.

2.1.1 "Here's what this does."

At first, the code scripts seem entirely mysterious to him. But as he watches what code is active when various things happen in the game, or double-clicks a script to make it run, he might start to get a feel for what some script is for. For example, by double-clicking the second script shown for the ball, he might notice that it puts

---

[2]http://scratch.mit.edu/projects/SampleProjectsTeam/62832

26

ball

paddle
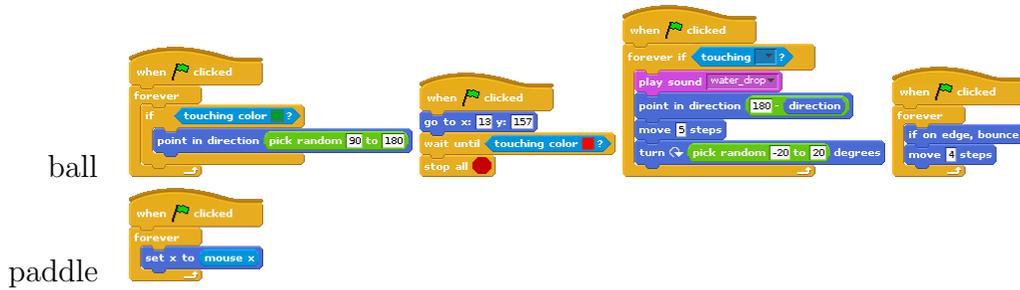
Table 2-1: Code in the sample "Pong" project, shown in Scratch's graphical code representation
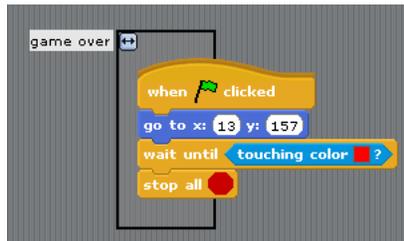
Figure 2-2: Scenario: A Zone describes the purpose of some Scratch code.

the ball in its starting location and stops the game when it hits the bottom. He can record his observation by dragging a Zone onto the script and typing a description of its purpose in his own words: "game over" (see Figure 2-2). He can describe the purpose however he thinks about it; there's no requirement that the description be a precise or comprehensive description of the script's behavior; it doesn't even have to be grammatical. There's one recommendation: by only giving one line, the Zones UI encourages descriptions to be short. Once he's done (and saves the project), the annotation is uploaded to a central server that records the text of the annotation, the code contained in it, and which project and sprite it came from. Now, the next time someone is looking for game-control logic and describes it with a phrase like "game over," they'll be able to find that code.

2.1.2 "What's this do?"

He successfully figures out what the rest of the scripts for the ball are for, but the script for the paddle remains mysterious: what does "set x" mean? He wants not only to figure out what the script does but to see how other people describe it. Figure 2-3

27

(a) The mysterious script (from the "paddle" sprite)

(b) The programmer drags a Zone onto the script

(c) The programmer clicks to open the Zone browser (left), which suggests possible annotations
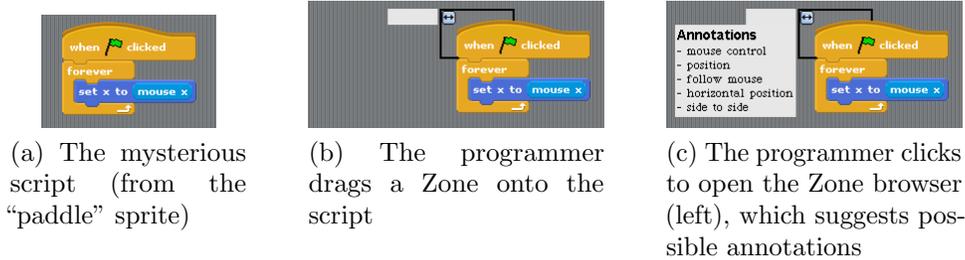
Figure 2-3: Scenario: The Zone can suggest possible annotations for a script.

shows how he can use a Zone to search for descriptions. He drags a Zone onto the script, but doesn't type an annotation. Instead, the Zone shows him annotations that other people have written for code like his mystery script.[3]

2.1.3 Adding new behaviors

After getting comfortable with the code in the example project, many novice programmers try adding and changing things to make their own version. A user named natey123 "remixed" that sample project to make his "Sonic Pong"[4], making the keyboard instead of the mouse control the paddle, adding an enemy that the ball would have to hit several times, and protecting the enemy with a platform that the ball would bounce off of. Consider if natey123 had been able to use Zones.

2.1.4 "How do I?"

He made the enemy stand on top of a platform, but the ball goes right through the platform. To fix this problem, he creates a new Zone, but doesn't put any code inside. He just types in "bounce off platform" as the annotation. When he clicks, the Zone browser opens, showing him code that others wrote that does similar things. Some of the results are irrelevant, but in Figure 2-4a he has found one that nearly works; he just needs to tweak the numbers. In fact, Zones highlights the numbers in red outline

---

[3]The reverse order—searching for annotations before adding them—is more likely in practice, but I describe the annotation process first for clarity. Also, the suggested annotations were modified from the actual ProcedureSpace annotation results for clarity; section 4.8 discusses the actual search process and its performance.

[4]http://scratch.mit.edu/projects/natey123/246133
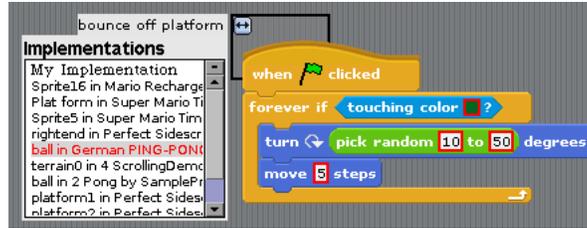
(a) Given an goal, the Zone browser shows code that might fulfill it. Selecting an implementation from the list on the left shows its code on the right. Red boxes surround values that vary among otherwise similar code, highlighting what might need to be changed.

since others also had to change those parameters when they reused similar code.[5] In the spirit of open source, the system automatically acknowledges the author(s)[6] of the original code and lets them know that their code was helpful.

In this example, the programmer did find code that accomplished his goal. However, if the system was unsuccessful at finding relevant code (e.g., because of insufficient knowledge about his goal), his use of Zones is still important. Since the empty Zone that he used for searching remains in his workspace, he'll just put the new code he writes inside the Zone. That is, an unsuccessful search becomes an annotation. Then when he saves the project, his new code—and the particular way he annotated it—gets added to the knowledge base so that other programmers can now find it—and of course, he'll get the credit.

2.2   Zones Link *What* (in English) with *How* (in code)

Through Zones, the user and computer work together to connect *what* the user wants to do with the code that tells *how* to do it. The key idea is that the computer has some understanding both of code and of natural-language descriptions of what code is for. Another defining characteristic of Zones is that this understanding comes from reasoning over a large collection of code, including some examples of the same types

---

[5]Parameter highlighting is not actually implemented yet; the red outlines are a temporary mockup.
[6]That code could have itself been modified from some other project!

of annotations. That way, the system can learn just by watching what people do, i.e., what code they write or reuse when they state a certain intent. When the system understands, it's helpful to the user; when it doesn't understand, the user's behavior is helpful to the system. I tentatively call such an interface an *intelligent goal-sharing interface*, because it facilitates sharing procedures that accomplish various goals within a user interface.

2.2.1   Interactions

The scenario showed just a few of the many possible interactions that an interface like Zones enables. They can serve as just another kind of comment, but they can also automatically suggest annotations based on the code they contain. If an annotation is used as a query, the system will show code that others have written when they had similar goals.

Another possible interaction is using code (possibly in conjunction with annotations) to find other code. If code is used as a query, the Zone would show alternative implementations, determined both by similarity of the code and purpose (captured by the annotations). The alternative implementations may make the programmer realize that they missed a corner case, or discover alternative approaches to the problem—or they may entirely replace their existing code with the foreign code and adapt it to work in their program.

2.2.2   Reuse

When the Zone presents a foreign code fragment, the programmer can reuse it directly, or at least use it as inspiration. The Zone helps the programmer integrate the foreign code by highlighting parameters that differ in different copies of code that is otherwise similar to the code fragment in question.

Creativity is expensive; a programmer (especially a novice) may take a strategy they're familiar with, even if a different approach would be easier or better. By juxtaposing different strategies that various people took (that presumably worked for

Table 2-2: User-contributed examples of different ways of describing a goal

- speed, velocity, moving
- animation, costume change, costume switch
- health bar, endurance/life points, life meter, health meter
- gravity, falling down
- fire, shoot, slash, attack
- obstacles, barriers, walls

them), the programmer can readily survey the options available and make a more informed decision, whether it be reusing, adapting, or writing code.

2.2.3   Bootstrapping

Most search queries are transient, but a Zone's link between code and purpose is persistent: unless the programmer explicitly removes the annotation, it remains as a marker in the code, automatically linking it into a repository for the next programmer's benefit.

Collecting code annotations has a bootstrapping problem: annotating code only becomes very useful to programmers after a variety of annotations have already been collected. To seed the database of annotations, I posted a message[7] on the Scratch forum that described our project and asked for examples of code and annotations. Their responses included many examples of different ways of describing the same or similar goals (Table 2-2 shows several examples.). Different users or communities (e.g., novices or experts) may have different ways of describing the same thing, or people may describe goals at different levels. The variety of annotations highlights a strength of the approach: since Zones helps collect examples of different kinds of annotations for the same code, the backend reasoning process can use commonalities in code structure to make connections between the different styles of descriptions.

The examples gained from the forum were general and hypothetical, but validated

---

[7]`http://scratch.mit.edu/forums/viewtopic.php?id=22917`, posted by a collaborator on the Scratch team

the idea of flexible code annotation. To gain actual annotation examples, I sent the Zones interface (with browsing disabled) to selected participants in the forum discussion, with instructions on how to annotate. Their contributions together with ours totaled 96 annotations. The user studies described in Chapter 5 contributed additional annotations.

2.2.4 Integrated in the Programming Environment

Programming requires a lot of context, both in the mind and in the programming interface. Searching and sharing generally require different tools than typing or browsing code. Switching between tools can exact a high cost in time, effort, and distraction as the programmer must manually transfer contextual knowledge between tools. Sharing context across tools can help reduce the stress of tool-switching. One approach, taken by Codetrail[9] is to send data back and forth between a programming environment and a web browser. Another approach is to integrate the other tool directly into the programming environment, as was taken by CodeBroker[36] and Blueprint[3]. Following their example, Zones is implemented within the programming environment and thus can examine code within the project and transplant other code directly into it.

A reason for resistance to comments and other kinds of code annotation is that programmers often cannot see the benefit that comes from attaching natural language descriptions to code. However, because the Zones search is sensitive to both code and annotation, programmers will learn over time that attaching comments to code has the effect of bringing up that code in other situations that are relevant to that purpose.

Chapter 3

Background

This chapter gives background information about the programming environment and the inference techniques used in this thesis. Section 3.1 describes the unique style and structure of Scratch code and projects, then explains the origin of the corpus of code used. Section 3.2 gives an overview of intuitive commonsense reasoning (often called "Digital Intuition"), then gives new pedagogical coverage of the AnalogySpace technique. It then presents the Blending technique for reasoning jointly over multiple kinds of data, and covers several blend layouts in a more mathematically rigorous way than previous publications.

3.1 Scratch

The first implementation of Zones is for Scratch[28], a graphical programming language designed for use primarily by children and teens, ages 8 to 16. Programming language components are represented by blocks that fit together like puzzle pieces to form expressions. One unusual aspect of Scratch is that named event handlers are often used to modularize programs, because the language lacks recursive functions or parameterized subroutines. Scratch code tends to be very concrete instructions, all near the same level of abstraction, that cause *sprites* to move around a *stage*, often in response to keyboard or mouse input. Additionally, the Scratch website[22] hosts hundreds of thousands of projects, many already reusing code from other projects,

all shared under a free software license. I chose Scratch for this first implementation because interpreting Scratch code is straightforward: the runtime environment is fixed, scoping is very simple, and functional dependencies are rare. Starting with Scratch let me focus on the core idea of connecting natural language and programming language. But I conjecture that techniques described in this thesis, suitably adapted, will also work with other collections of code in different languages.

3.1.1 Program Structure

Scratch programs are called *projects*. The UI of a Scratch project is contained within the *stage*, on which various *sprite*s are displayed. The sprites can contain code to move, change their appearance, play sounds, or manipulate a pen, which execute in response to keyboard or mouse input, broadcast messages, or a global start command. The language includes control flow, boolean, and mathematical statements, as well as sprite- or project-scoped variables. Most code in a project is contained within the sprites and has egocentric semantics, though the stage can also contain code.

Scratch code is made by snapping together *code block*s,[1] not typing text. Stacking blocks vertically causes them to execute in sequence. A connected stack of blocks is called a *script* in Scratch; in this thesis "script" is used interchangeably with the more general term "code fragment." All scripts are event handlers, so concurrency is idiomatic. A script begins with a "hat" block, which specifies what event invokes it. One commonly used event is "when green flag clicked" (called FlagHat in the technical sections of this thesis), which is typically used to initialize the program and start long-running processes. The other "hat" blocks are MouseClickEventHat, KeyEventHat, and EventHat (which starts the script in response to a user-defined event). Keyboard and mouse input can also be handled by polling.

---

[1]not to be confused with the statement blocks of traditional programming languages, which indicate sequences of statements.

3.1.2 Code Corpus

Andrés Monroy-Hernández created the Scratch community website, where programmers and users of all ages can share, interact with, and "remix" (share modified versions of) each other's projects. Rita Chen wrote a parser that extracts various information from all of the Scratch projects, including an S-expression form of the code for each sprite in the project. She provided the extracted information for 278,689 projects, which formed the corpus for these experiments.

To increase code quality and reduce computational requirements, I reduced the project set to the 6,376 Scratch projects that (a) at least 2 people "remixed" (reused as a basis for their own project) and (b) were marked as a favorite by at least one person. This yielded 127,216 scripts in total.

3.2 "Digital Intuition": Background Knowledge in Natural Language

The ProcedureSpace reasoning process uses some of the recent work of my research group, the Commonsense Computing Initiative at the MIT Media Lab. A recent summary article gave it the descriptive name "Digital Intuition"[14]. This section explains that work more pedagogically, including a new step-by-step presentation of the Singular Value Decomposition as it is used in AnalogySpace. It also introduces a layout for the Blending technique, called Bridge Blending, presents a variation on it that will be used in ProcedureSpace, and gives a mathematical analysis of its ability to perform joint inference. Readers who are already familiar with the work of the group should skip the ConceptNet section (3.2.2), skim the AnalogySpace and Blending sections (3.2.3 and 3.2.4), and read the section on Bridge Blending (3.2.5).

3.2.1 Overview

Whenever we're interacting with the world, whether the situation is familiar or unfamiliar, we understand what we experience in terms of what we already know.

Some of that knowledge may be very specific, such as what Mom's face looks like. But a large amount of that knowledge is general, and much of it is shared between people. In fact, shared knowledge is necessary for communication. If I just say the word "dog," you know that I'm probably referring to a four-legged pet that can bark, even though I never mentioned any of those facts. This shared background knowledge—what nearly everybody knows but rarely explicitly says[2]—is sometimes referred to as "commonsense knowledge." Even people who are said to "have no common sense" usually know quite a bit (though perhaps with an egregious omission), but for the most part, computers entirely lack this knowledge.

A sufficiently advanced learning system, provided with a rich set of interactions with the world, could learn commonsense knowledge semi-automatically, as humans do, though perhaps with a lot of parental guidance. On the opposite extreme, simple techniques on large corpora can identify very simple common sense facts, like the fact that the noun "dog" and the verb "bark" often occur together. But obtaining rich, accurate knowledge about how those words are related currently requires human training.

The goal of the Open Mind Common Sense (OMCS) project is to collect commonsense knowledge and develop techniques to enable intelligent systems and user interfaces to work with it. Our main focus so far has been on knowledge that can be expressed in the form of simple sentences expressing relationships that generally hold between two concepts—for example, "Cheese is made with milk" and "Something you find in your mouth is teeth." Such statements generally express the way objects relate to each other in the world, people's everyday goals, and the emotional content of events or situations. We are interested not just in this knowledge per se, but also in how people describe it in words. We have been collecting this knowledge since 1999 using the principle now known as "crowdsourcing": the best way to find out what people know, we assume, is to ask a lot of people. On our website,[3] anyone can contribute new statements or rate the accuracy of existing ones. They either do this

---

[2]One of Grice's maxims of pragmatics is that people avoid stating information that is obvious to the listener [11].

[3]`http://openmind.media.mit.edu/`
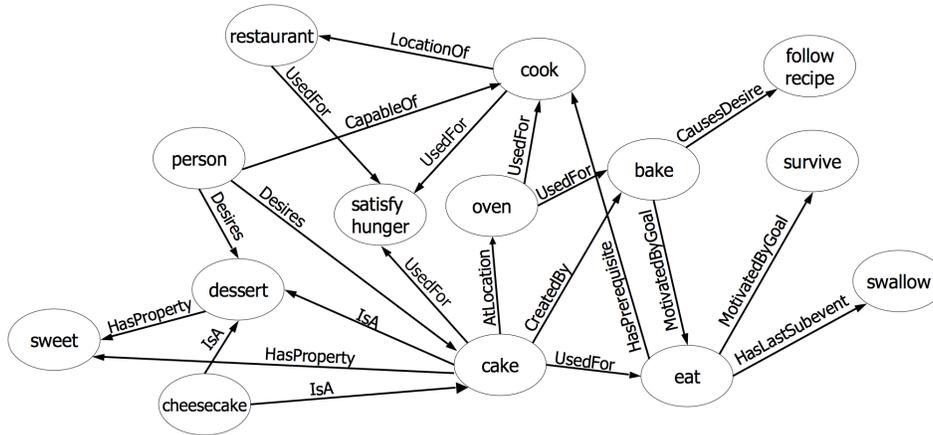
Figure 3-1: Some of the nodes and links in ConceptNet surrounding the concept "cake"

directly, by filling in blanks in sentence templates, or through a variety of interactive activities and games. Joined with contributions from collaborators around the world, our corpus now has over one million simple English statements, plus hundreds of thousands of statements in other natural languages such as Chinese, Japanese, and Brazilian Portuguese.

3.2.2   ConceptNet

To make the knowledge in the OMCS corpus accessible to AI applications and machine learning techniques, we transform it into a semantic network called ConceptNet[13, 20]. A semantic network is a directed graph with labeled edges. The nodes of ConceptNet are short phrases, called *concepts*,[4] such as "cake" or "dog." The edges represent *relations* that have been asserted to hold between those concepts. A slice of ConceptNet surrounding the word "cake" is shown in Figure 3-1.

The relations are taken from a fixed set of basic binary relationships, such as *IsA*, *HasA*, or *UsedFor*. This set of relations was initially derived from common sentence patterns we found in the OMCS corpus; a few relations have since been added for specific purposes. Anyone can contribute knowledge in any relation. Table 3-1 shows

---

[4]In its current representation, ConceptNet does not distinguish word senses or other nuances that are sometimes considered in the field of concept learning.

Relation	Example sentence frames
IsA	*NP* is a kind of *NP*.
UsedFor	*NP* is used for *VP*.
HasA	*NP* has *NP*.
CapableOf	*NP* can *VP*.
Desires	*NP* wants to *VP*.
CreatedBy	You make *NP* by *VP*.
PartOf	*NP* is part of *NP*.
HasProperty	*NP* is *AP*.
Causes	The effect of *VP* is *NP*|*VP*.
MadeOf	*NP* is made of *NP*.
AtLocation	Somewhere *NP* can be is *NP*.
DefinedAs	*NP* is defined as *NP*.
SymbolOf	*NP* represents *NP*.
ReceivesAction	*NP* can be *VP*$_{\text{passive}}$.
Causes	The effect of *NP*|*VP* is *NP*|*VP*.
MotivatedByGoal	You would *VP* because you want to *VP*.
CausesDesire	*NP* would make you want to *VP*.
HasSubevent	One of the things you do when you *VP* is *NP*|*VP*.
HasFirstSubevent	The first thing you do when you *VP* is *NP*|*VP*.
HasLastSubevent	The last thing you do when you *VP* is *NP*|*VP*.
LocatedNear	*NP* is typically near *NP*.
HasPrerequisite	*NP*|*VP* requires *NP*|*VP*.
HasA	*NP* has *NP*.
SimilarSize	*NP* is about the same size as *NP*.

Table 3-1: The 20 relations in ConceptNet 4. *NP* = noun phrase, *VP* = verb phrase, *AP* = adjective phrase

the current set of 20 relations, along with an example of a sentence frame that expresses that relation. Some of the relations are also found in other semantic knowledge bases such as WordNet[7] and the Brandeis Semantic Ontology[26], but others are unique to ConceptNet. Relations can be negated, as well, to express negative knowledge such as "*A dog* cannot *fly.*"

Concepts represent sets of closely-related natural language phrases, which could be noun phrases, verb phrases, adjective phrases, or clauses. In particular, concepts are defined as the equivalence classes of phrases after a normalization process that removes function words, pronouns and inflections.[5] The concept normalization process (not to be confused with a vector normalization process that will be described later) avoids

---

[5]As of ConceptNet 3.5, we remove inflections using a tool based on the multilingual lemmatizer MBLEM [32].
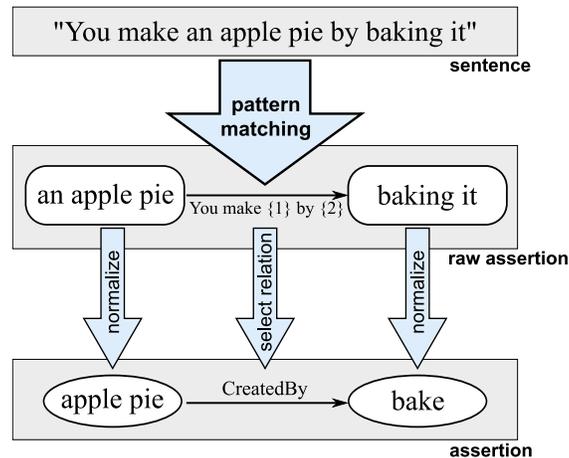
Figure 3-2: Adding an edge to ConceptNet based on a sentence. The sentence is transformed into a *raw assertion* with pattern matching, and then normalized into an *assertion* of ConceptNet.

unnecessary sparsity and duplication by making phrases equivalent when they seem to have approximately the same semantics but are expressed in different ways. Using this process, for example, the phrases "drive a car," "you drive your car," "driving cars," and "drive there in a car" all become the same concept, represented by the normalized form "drive car."

People often contribute knowledge by filling in the blanks in sentence patterns like those in Table 3-1.[6] Alternatively, a shallow parser can match a frame to a free-text sentence. Either action creates a *raw assertion*, which is a frame together with a pair of words or phrases, called *surface forms*, that fill it in. For example, if a contributor filled in the template "You make ___ by ___" with "You make *an apple pie* by *baking it*," the surface forms would be *an apple pie* and *baking it*. Then the normalization process associates these surface forms with concepts, in this case, *apple pie* and *bake*. The result is an *assertion* where a generalized relation connects two normalized concepts. Commonsense assertions are notated like *apple pie*\\`CreatedBy`/*bake*. Each assertion is an edge in the ConceptNet semantic network. An example of this parsing process is provided in Figure 3-2.

Each assertion is associated with a *frequency* value which can express whether

---

[6]The part-of-speech constraints given in the table are not enforced during fill-in-the-blank activities.

Adverb	$frequency$ value
always	10
almost always	9
usually	8
often	7
unspecified	5
sometimes	4
occasionally	2
rarely	-2
not	-5
never	-10

Table 3-2: Mapping of frequency adverbs to values in English ConceptNet

people say the relationship *sometimes*, *generally*, or *always* holds; there are also frequency values that introduce negative contexts, to assert that a relationship *rarely* or *never* holds. These frequency adverbs are associated arbitrarily with numbers from -10 to 10, given in Table 3-2. Independently of the frequency, assertions also have a *score* representing the system's confidence in that assertion. When multiple users make the same assertion independently, that increases the assertion's score. Users can also choose to increase or decrease an assertion's score by rating it on the OMCS web site. This allows collaborative filtering of deliberate or inadvertent errors. The score value is computed by subtracting the number of dissenting users from the number of assenting users.

Unlike the carefully qualified and precise assertions of formal logic, ConceptNet assertions are intended only to capture approximate, general truth. For example, someone might say "Birds can fly," which would be interpreted as $bird\backslash$`CapableOf`$/fly$. Though this statement is intuitively true in general, formal logic would require numerous qualifiers: not penguins or injured birds, flying in the physical (not metaphorical) sense, only in air within typical ranges of temperature, pressure, and composition, etc. Without such qualifiers, a logical reasoner would readily make erroneous or even contradictory conclusions.[7] Yet people tend to think about and state the general facts first, and only mention qualifiers and exceptions when they become important,

---

[7]Overly "logical" reasoning on ConceptNet can be a good supply of humor, however.

and even then, they may be hard-pressed to enumerate *all* necessary qualifiers. So while collecting data in natural language requires us to sacrifice the power of logical inference, it enables people to contribute without training and can connect to a wide variety of natural language data sources. For reasoning over this imprecise, noisy, and loosely-structured data, we have developed various approximate reasoning techniques, including one called AnalogySpace.

3.2.3 AnalogySpace

The commonsense data collected has many hidden similarities. For example, a dog is a pet and an animal; a cat is also a pet and an animal. They probably share many other characteristics also: they can be found in a house, people want them, and they want food. Instead of considering each of these characteristics independently, it would be more efficient to consider all the characteristics that pets have as a whole. This bulk consideration, which is called in general "dimensionality reduction," also helps deal with noisy and incomplete data. For example, if no one thought to tell ConceptNet that a Golden Retriever might be found in a house, we could still conclude that fact readily if we think about a Golden Retriever as a pet. Or if someone claimed that a Golden Retriever is made of metal, we would have reason to be suspicious of that statement because it is incongruous with the features we know about other pets.[8] Finally, as Section 3.2.4 discusses, dimensionality reduction allows us to reason about "eigencharacteristics" that cross the boundaries of sensory modalities.

The resulting technique is called AnalogySpace and was introduced by my colleagues in [30]. In general, this kind of technique is called Principal Component Analysis or Latent Semantic Analysis[24]. This discussion attempts to illuminate the analysis process without requiring fluency in linear-algebra.

---

[8]Hierarchy learning could also perform some of these tasks. This observation suggests that hierarchy learning can be treated as dimensionality reduction as well, where constraints such as implication or exclusion can hold between latent classes. I may explore this connection in future work. For the purposes of this thesis, we will consider only linear dimensionality reduction.

Matrix Representation

The first step is to express ConceptNet as a matrix. The rows of the matrix will be the concepts, like "dog," "cat," and "taking out the garbage." The columns will be *features*—a relation and a concept. For example, the concept "dog" may have the feature ___\IsA/*animal*, meaning that a dog is an animal. A ConceptNet assertion that a concept c_i has a feature f_j results in a positive value in $A(i, j)$, where A is the ConceptNet matrix. Similarly, an assertion that c_i lacks f_j results in a negative value. (ConceptNet has many more positive assertions than negative assertions.) However, the statement "A dog is an animal" asserts not only that something we know about "dog" is ___\IsA/*animal*, but also something we know about "animal" is *dog*\IsA/___, i.e., one kind of animal is a dog. So each natural language statement contributes two entries to the matrix. The actual numerical value is determined by the *score* and *frequency* of the assertion, as described in Section 3.2.2:

$$\frac{frequency}{10} \log_2 \max(score + 1, 1)$$

Frequency is scaled by 10 so that it ranges from -1 to 1. The logarithm of the score (clamped to a minimum of 1) is used to reduce the marginal effect of votes beyond the first.

Here's a sample of the ConceptNet matrix:

CNet =	___\IsA/*animal*	*person*\Desires/___	$\cdots$
dog	0.50	1.16	$\cdots$
cat	0.50	0.79	$\cdots$
toaster	0	0	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$

Matrix Operations

Several simple operations can be performed using the ConceptNet matrix (which we'll call A in this discussion). Mainly we'll be multiplying this matrix by one of two kinds of vectors: a vector $\vec{c}$ containing a numeric weighting for each concept, and a vector

$\vec{f}$ containing a numeric weighting for each feature. If we wanted to look up all the animals, we'd construct an f that is 1.0 for __\IsA/*animal* and 0 for every other feature. Then the animals are given by $A\vec{f}$, represented as a numeric weight for each concept of how much the __\IsA/*animal* feature applies to it—limited, of course, by the accuracy and completeness of the ConceptNet data. In this case, $A\vec{f}$ just extracted a column of A, but the matrix product notation is more general. For example, we could find animals that were not pets by adding a -1.0 in __\IsA/*pet*.[9] Generally, we would normalize the vectors such that their Euclidean magnitudes are 1.[10]

Going the other direction, if we had a collection of pets, we could find the properties they share by constructing a concept weight vector $\vec{c}$ and computing $A^T\vec{c}$. One concern immediately arises, however. Suppose a horse lover has put in a huge number of unique features about "horse." Then the resulting feature weights will be disproportionately influenced by the horse features. To keep things we know a lot about from having an undue influence, we normalize each row so that its Euclidean norm is 1. But for things we know very little about, this normalization gives the features we do know a disproportionately large weighting. So we calculate the norm as if each row had an extra entry of a constant value β; I typically use $\sqrt{5}$ so that 5 is added to the dot product.

Similarity

Another simple operation we can do with A is take the dot products between rows or columns. Each concept can be represented by its vector of feature weights: $A(i,:)$, where the : notation indicates a slice of an entire row. The dot product of vectors $\vec{a}$ and $\vec{b}$ is $\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos \theta$, where θ is the cosine of the angle between the two vectors. The $\cos \theta$ term directly measures the similarity: it ranges from 1 (for vectors pointing in exactly the same direction) to -1 (for opposing vectors). The magnitudes $|\vec{a}|$ and $|\vec{b}|$ weight the similarity by (roughly) how much is known about each concept. We can

---

[9]The pets will not come out as *exactly* 0 unless the scores and frequencies on both assertions were exactly the same.

[10]Recall that the Euclidean magnitude of a vector $\vec{v}$ is the square root of its dot product with itself, $\sqrt{\vec{v} \cdot \vec{v}}$, and that the dot product of two vectors is the sum of the products of their corresponding elements.

compute all the concept-to-concept similarities at once by forming the matrix product AA^T. The element $AA^T(i, j)$ is the dot product of the feature vector for concept i with the feature vector for concept j, i.e., their weighted similarity. So to find the concepts that are most similar to a given concept (weighted by how much is known about the other concept), you can simply look for the highest entries along a row (or column) of AA^T. Likewise, feature-to-feature similarity can be represented as $A^T A$. Both matrices are symmetric.

Eigenconcepts and Eigenfeatures

Now we'll use the ConceptNet matrix look for the bulk characteristics mentioned at the beginning of this section. We'll call these characteristics *eigenconcepts* and *eigenfeatures* for reasons that will become apparent. Suppose "pet-ness" was an eigenfeature—it turns out that it isn't, but the process will be illuminating. It would probably be strongly associated with the features ___\IsA/*pet*, ___\IsA/*animal*, *person*\Desires/___, etc. Though we don't yet know exactly what all the features are, or how strongly they should be weighted, we can guess that it has those three features weighted equally. We could write our guess at "pet-ness" as a vector $\vec{f}$:

person\Desires/___	___\IsA/*animal*	___\IsA/*pet*
0.58	0.58	0.58

We've normalized "petness" to have a Euclidean norm of 1.0. If we guessed the right "pet-ness" vector (and if "pet-ness" is indeed an eigenfeature), then multiplying by the ConceptNet matrix would give a vector of how much "pet-ness" each concept has. (We'll call the ConceptNet matrix A so that the equations are a bit more general.)

$$A\vec{f} = \vec{c} = \sigma\vec{c}$$

Here, we've split the result vector into a magnitude σ and another unit vector $\vec{c}$. The vector $\vec{c}$ looks like:

dog	cat	clothe	rabbit	own house	praise	turtle	$\cdots$
2.11	1.15	1.07	1.03	1.00	1.00	0.87	$\cdots$

"dog" and "cat" look good, but where did "clothe" and "own house" come from? They came from the $person\backslash\texttt{Desires}/\_\_\_$ feature: people want clothes (normalized to "clothe") and to own houses. So intuitively, $\vec{f}$ wasn't a good guess for "pet-ness." If it had been a good guess (and if "pet-ness" was an eigenfeature), then "pet-ness" would have been the sum of the features of all pets, that is:

$$A^T \vec{c} = \sigma \vec{f}$$

(The scaling factor σ in this equation is only the same as the σ in the previous equation if $\vec{c}$ is an eigenconcept.) If we do that multiplication to our current $\vec{c}$, we'd get what's hopefully a better approximation to "pet-ness" as our new $\vec{f}$:

$person\backslash\texttt{Desires}/\_\_$	$\backslash\texttt{IsA}/animal$	$\backslash\texttt{IsA}/mammal$	$\backslash\texttt{AtLocation}/zoo$	$\backslash\texttt{HasProperty}/animal$	$\backslash\texttt{HasProperty}/good$	$\backslash\texttt{UsedFor}/fun$	
0.93	0.14	0.07	0.05	0.04	0.04	0.04	$\cdots$

We can see that desirability has become even more prominent, and has pulled in other related notions, such as $\_\_\backslash\texttt{UsedFor}/fun$. This process will not converge on "pet-ness" after all; rather, it seems (in a sense we'll formalize momentarily) that it converges on the most prominent eigenfeature. Moreover, it seems that the most prominent eigenfeature is in fact desirability, with desirable things being the most prominent eigenconcept.

If we repeated this process, we would eventually[11] find a stable pair of vectors $\vec{c}$ and $\vec{f}$ and a corresponding scaling factor σ. Then the two equations would actually

---

[11] ...subject to numerical accuracy and eigenvalue multiplicity constraints that are far beyond the scope of this background section, and do not affect the results or discussion

hold: $A\vec{f} = \sigma\vec{c}$ and $A^T\vec{c} = \sigma\vec{f}$. In that case, $\vec{c}$ and $\vec{f}$ are called *singular vectors*, and σ is called the *singular value*. We can solve for the singular vectors by substitution. For example, to solve for $\vec{c}$, multiply on the left by A:

$$AA^T\vec{c} = A\sigma\vec{f}$$
$$= \sigma A\vec{f}$$
$$= \sigma^2\vec{c}$$

Similarly, $A^TA\vec{f} = \sigma^2\vec{f}$. Those familiar with linear algebra will recognize that $\vec{c}$ is an eigenvector of AA^T, and σ^2 is its eigenvalue. So the singular vectors are the eigenvectors of AA^T and A^TA; that's why I called them "eigenconcepts" and "eigenfeatures." There are in fact many eigenconcepts and eigenfeatures, since AA^T and A^TA have many eigenvectors. Algorithms to find eigenvectors are standard in most mathematics toolkits.[12]

A result in linear algebra called the spectral theorem states that since AA^T is symmetric, its eigenvectors $\vec{c_i}$ are all orthogonal. If we line them up as column vectors side-by-side (putting the ones with the largest eigenvalues on the left), we get a matrix C. Since the eigenvectors are orthogonal (i.e., dot products between them are zero) and unit magnitude (i.e., their dot products with themselves are 1), C is "orthonormal" and $CC^T = I$, the identity matrix (1 along the diagonal, 0 elsewhere). The spectral theorem states further that we can write $AA^T = C\Sigma^2C^T$, where Σ^2 is a matrix with the squares of the singular values on the diagonal;[13] the diagonal entries will be in decreasing order. We can do the same for A^TA, getting an orthonormal matrix F of the $\vec{f_i}$'s and the same diagonal matrix Σ.

---

[12]If you're stranded on a desert island, or just bored, here's how to compute eigenvectors of an n-by-n matrix A. (1) Choose a random vector $\vec{v} \in \mathrm{R}^n$ (e.g., a column of A). (2) Update $\vec{v} := A\vec{v}$. (3) Normalize $\vec{v}$. (4) Repeat (1) through (3) until $\vec{v}$ stops changing: it's an eigenvector, and you just divided by its eigenvalue λ. (5) Update $A := A - \lambda\vec{v}\vec{v}^T$. (6) Repeat (1) through (5).

[13]The eigenvalues are all positive because AA^T and A^TA are both positive definite.

Singular Value Decomposition

Now that we have C and F, we can write the defining equations for all the singular vectors at once:

$$AF = C\Sigma$$

(We write Σ on the right of C because each diagonal entry multiplies a *column* of C.) Now we can multiply on the right by F^T, remembering that $FF^T = I$ because F is orthonormal:

$$A = C\Sigma F^T$$

This equation is known as the *singular value decomposition* (SVD) of A and is usually written:

$$A = U\Sigma V^T$$

where, again, Σ is a diagonal matrix of the singular values of A, which are square roots of the eigenvalues of AA^T and $A^T A$, and U and V are orthonormal matrices ($UU^T = I$ and $VV^T = I$).

The result of this analysis is a vector space that we call AnalogySpace. The singular vectors—the eigenconcepts and eigenfeatures—are the *axes* of AnalogySpace. The location of a concept in AnalogySpace is given by its corresponding row in U (or C); the location of a feature in given by its row in V (or F).

Truncation

Another way to write the singular value decomposition is as a sum of outer vector products.[14] Let $\vec{u_i}$ and $\vec{v_i}$ be column i of U and V, respectively, i.e., they are the singular vectors corresponding to the singular value σ_i. Then

$$A = \sum_{i=0}^{n} \sigma_i \vec{u_i}^T \vec{v_i}$$

---

[14]The outer product of two column vectors produces a matrix where each element is just the product of the corresponding elements of each vector. The most familiar outer product is a multiplication table: if $\vec{n} = [1, 2, 3, 4]^T$, then $\vec{n}^T \vec{n}$ is the 1-through-4 multiplication table.

Each outer product $\vec{u_i}^T \vec{v_i}$ can be thought of as a simplistic view of the world. For example, for the first axis (desirability), $\vec{u_0}[c]$ gives how desirable concept c is, and $\vec{v_0}[f]$ gives how desirable it is to have feature f. So we can think of A as being a linear combination of these simplistic views of the world. And since the singular values are in decreasing order of magnitude, the axes that account for the most variation come out first. That is, each additional axis accounts for patterns within A that the previous axes didn't account for. Less-significant patterns tend to correspond to noise, both additive and subtractive. Additive noise is extraneous data added to ConceptNet by, e.g., spammers, confused people, or parsing bugs. Subtractive noise is missing data that would be consistent with the data that is present, but the entry in the matrix was forced to zero because that fact had not yet been added. So we can both filter out extraneous data and fill in missing data by only considering the top k axes:

$$A \approx A_k = \sum_{i=0}^{k} \sigma_i \vec{u_i}^T \vec{v_i}$$

which we can write in matrix form as

$$A_k = U_k \Sigma_k V_k^T$$

In fact, it turns out that the $\vec{u}$'s and $\vec{v}$'s of the SVD give the best approximation of this form, in terms of the Frobenius norm of the error—the sum of the squares of the matrix elements. We often choose k to be 100, though this choice is admittedly not systematically motivated.

Selected SVD Properties

For reference, here are a few useful properties of the truncated SVD ($A_k = U_k \Sigma_k V_k^T$):

- $A_K^T = V_k \Sigma_k U_k^T$, i.e., transposing A just flips the roles of U and V.

- $U_k U_k^T = I$, i.e., the columns of U (the left singular vectors) are orthonormal.

- $V_k V_k^T = I$, i.e., the columns of V (the right singular vectors) are also orthonormal.

- $\sigma_0 = \max|A\hat{x}|$, i.e., the largest singular value is the upper bound on the magnitude of multiplying A by a unit vector. This is sometimes used as a measure of the magnitude of A, and will be used to estimate relative weights when combining matrices in Section 3.2.4.

- $\left(\sum_{k=0}^{K} \sigma_k^2\right)^{1/2} = \left(\sum_{i=1}^{m}\sum_{j=1}^{n} A(i,j)^2\right)^{1/2}$, or loosely speaking, the (Euclidean) norm of all the singular values gives the (Frobenius) norm of the elements of the matrix.

Proofs of these properties, and of the existence and uniqueness of the SVD in general, can be found in a linear algebra textbook.

Similarity

Recall from section 3.2.3 that AA^T and A^TA give the concept-to-concept and feature-to-feature similarity. The SVD gives us an alternative way to write them:

$$
\begin{aligned}
AA^T &= \left(U\Sigma V^T\right)\left(U\Sigma V^T\right)^T \\
&= U\Sigma V^T V\Sigma^T U^T \\
&= U\Sigma\Sigma^T U^T \\
&= U\Sigma^2 U^T \\
&= (U\Sigma)(U\Sigma) \\
A^TA &= V\Sigma^T U^T U\Sigma V^T \\
&= V\Sigma^2 V^T \\
&= (V\Sigma)(V\Sigma)
\end{aligned}
$$

This means that instead of representing a concept as a vector of the weights of several thousand features $A(i,:)$, we can instead use a much smaller vector $(U\Sigma)(i,:)$, which has k (e.g., 100) numbers. More than just making the similarity computation more efficient, this more compact representation also filters out the additive and subtractive noise in the original data. And also, since both the concepts and features

are represented by the same kinds of vectors in AnalogySpace, we can easily ask what features a concept might have, or vice versa.

3.2.4 Blending

The Blending technique, developed by Havasi and Speer[14], is a technique that performs AnalogySpace-style inference over multiple data sources, taking advantage of the overlap between them. The idea is simple: just make a matrix for each data source, line up the labels (filling in zeros for missing entries), and add the matrices together. In fact, AnalogySpace itself is a blend of knowledge in different relations (`IsA`, `UsedFor`, etc.);[15] I call it the "self-blend" since, in a sense, we're blending ConceptNet with itself. The axes in AnalogySpace are then cross-domain representations; for example, axis 0, determining desirability, actually contains information about desirability (`Desires`), ability (`CapableOf`), and location (`AtLocation`). Likewise, blending ConceptNet with other data sources allows us to construct cross-domain representations between commonsense knowledge and the other data source. In our research so far, these other data sources have included other semantic datasets such as WordNet[7], domain-specific datasets, and free-text reviews of businesses and products. In this thesis, the other data sources will be data about code structure and code purpose.

An important aspect of blending is the relative weighting of each data source, which determines how much influence it has in the construction of the new semantic space. In the original AnalogySpace, the weights are constant: each relation is given a weight of 1.0. In this case, equal weighting turns out to perform reasonably well, but in many cases a more "blended" (more cross-domain) analysis can be obtained by adjusting the weights. To make the new semantic space be influenced by all of the input matrices, we weight them such that their variances are equal.[16]. Havasi et al. ([14]) discuss a "blending factor," chosen to maximize a measure called *veering*; this discussion of weights generalizes that idea. Sections 4.4.3 and 4.7 will further discuss the weighting factor in the context of the ProcedureSpace blend.

---

[15]AnalogySpace could also be viewed as a blend of left and right features.
[16]We approximate the variance of a sparse matrix by using its top n singular values: $\sqrt{\sum_{i=0}^{n} \sigma^2}$

English Features   **French Features**

English Concepts

English ConceptNet

X

French Concepts

French ConceptNet

Y

(a) No overlap = no blending

English Features   **French Concepts**

English Concepts

English ConceptNet

X

En ↔ Fr Dictionary

French Features

French ConceptNet$^{\mathsf{T}}$

Y$^{\mathsf{T}}$

(b) Transposing one of the matrices allows for a concept-to-concept bridge
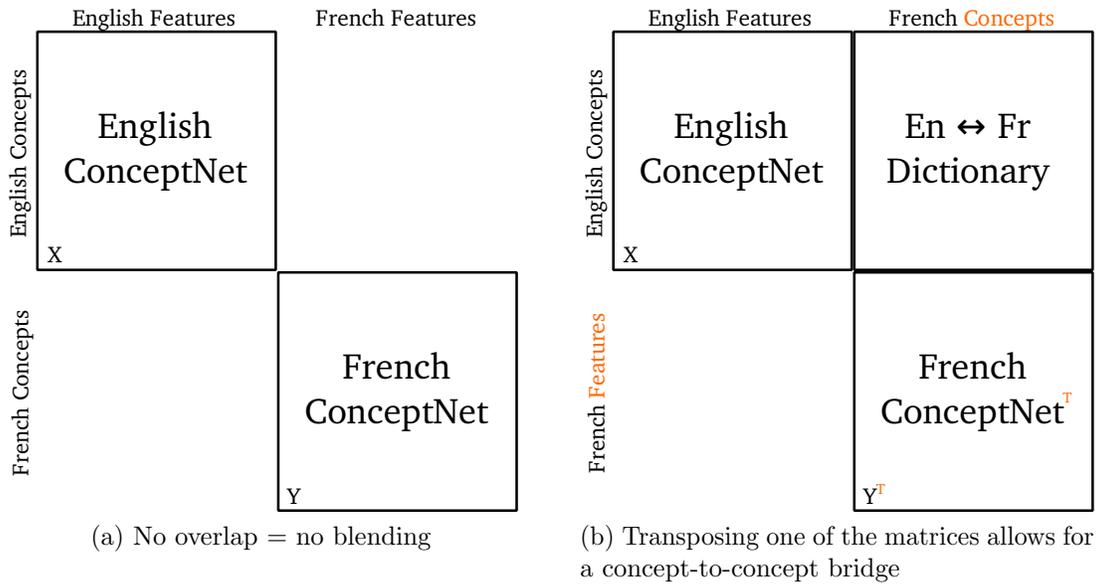
Figure 3-3: Bridge Blending: Connecting data that do not overlap

In collaboration with others in the Commonsense Computing research group, I developed an open-source software toolkit, called Divisi[5], for computing truncated SVDs of sparse labeled matrices. Divisi is also capable of performing normalization (see Sections 3.2.3 and 4.6.3), blending, and a wide variety of other operations that are often useful.

3.2.5   Bridge Blending

Blending only works when the two datasets overlap in either their rows or their columns. Consider the layout of Figure 3-3a,[17] for example: we have (hypothetically) commonsense knowledge in English and French, but without knowing which English concepts or features correspond to which French concepts or features, we have no way reasoning jointly over them. The similarity (dot product) between any English concept/feature and any French concept/feature in such a layout is exactly 0. In fact, it's readily shown that unless the two matrices share a singular value exactly, none of the axes will contain both English and French concepts. Rather, the set of singular

---

[17]Real commonsense data matrices are about 5 times as wide as tall, since most concepts participate in several features.

51

vectors of the "blend" will be simply the union of the singular values of the English and French matrices alone, padded with zeros. In highly technical terms, the result is... boring.

So how can we reason jointly over both English and French? We need to add another dataset, called a "bridge," to connect English and French. It could fill one of the missing off-diagonal entries in Figure 3-3, but with what? We would need data about either French features about English concepts, or English features about French concepts. We do not have that data directly, though we could possibly infer it from the English and French commonsense data. More readily available is a bilingual dictionary, connecting English concepts to French concepts and vice versa. We could transform that into a matrix of English concepts by French concepts. The bridge data could fit into the blend if we *transposed* one of the ConceptNet matrices, as in Figure 3-3b.

The canonical encoding of commonsense data is concepts on rows and features on columns; will the transposed arrangement still yield meaningful results? Transposing a matrix just reverses the roles of U and V in its SVD, so transposing a single matrix does no harm. But we might worry that the fact that the English and French concepts/features are on different sides of the matrix keeps them from being meaningfully related. This section gives a few steps towards a mathematical demonstration that cross-domain inference occurs in bridged blending in general and in the transposed arrangement in particular; a more complete mathematical treatment awaits a future publication. But we'll see an empirical demonstration of cross-domain reasoning with bridge blending in Section 4.5.1.

Let's call the English ConceptNet X and the French ConceptNet Y. X relates concepts x_i with features f_j; Y relates concepts y_i with features g_j. We then encode the bilingual dictionary into a matrix B, where $B(i, j)$ gives the similarity between concepts x_i and y_j. We now array the two datasets X and Y along with the bridge

dataset B in the **transposed bridge blend layout**:

$$C = \begin{bmatrix} X & B \\ & Y^T \end{bmatrix} \approx U\Sigma V^T$$

(Many bridge datasets would also allow us to make a reasonable guess at values from the bottom left corner, but we'll leave it blank for simplicity for now.) Intuitively, we suspect that the bridge dataset will cause the eigenconcepts and eigenfeatures to be composed of items from both X and Y^T. If this works, then we will be able to determine what French concepts apply to an English feature, or ask about the translation of different senses of a word based on projecting different combinations of features, all by computing matrix-by-vector products.

To see if it works, let's consider two simpler sub-problems. For both problems, we'll consider the case that the bridge data is a weighted identity matrix, i.e., every x_i corresponds to exactly one y_j with constant weight. This setup requires that the number of rows of X equal the number of rows of Y. Though realistic bridge blends break both of these rules, this setup is still a representative idealization.

Identities

First we consider the effect of just adding the bridge data. Since we're approximating the bridge data as a weighted identity matrix ($B = \alpha I$), this is equivalent to:

$$C = \begin{bmatrix} X & \alpha I \end{bmatrix}$$

To determine the effect of the bridge data on row-row similarities, we compute

$$CC^T = \begin{bmatrix} X & \alpha I \end{bmatrix} \begin{bmatrix} X^T \\ \alpha I \end{bmatrix} = XX^T + \alpha^2 I$$

That is, blending with α-weighted identities increases row-row dot products by α^2. If $XX^T\vec{v} = \lambda\vec{v}$ (i.e., $\vec{v}$ is an eigenvector of XX^T), then

$$CC^T\vec{v} = XX^T\vec{v} + \alpha^2 I\vec{v} = \lambda\vec{v} + \alpha^2\vec{v} = (\lambda + \alpha^2)\vec{v}$$

That is, blending with constant-weight identities adds α^2 to each eigenvalue without changing the eigenvectors.

Bridged Identities

Now let's consider actually adding the new dataset Y. Recall the transposed bridge layout:

$$C = \begin{bmatrix} X & \alpha I \\ 0 & Y^T \end{bmatrix}$$

We start by computing the row-row dot products:

$$CC^T = \begin{bmatrix} X & \alpha I \\ 0 & Y^T \end{bmatrix} \begin{bmatrix} X^T & 0 \\ \alpha I & Y \end{bmatrix}$$

$$= \begin{bmatrix} XX^T + \alpha^2 I & \alpha Y \\ \alpha B & Y^T Y \end{bmatrix}$$

If $XX^T\vec{u} = \lambda\vec{u}$ (i.e., $\vec{u}$ is an eigenvector of XX^T), then

$$CC^T \begin{bmatrix} \vec{u} \\ 0 \end{bmatrix} = \begin{bmatrix} XX^T + \alpha^2 I & \alpha Y \\ \alpha Y^T & Y^T Y \end{bmatrix} \begin{bmatrix} \vec{u} \\ 0 \end{bmatrix}$$

$$= \begin{bmatrix} XX^T\vec{u} + \alpha^2 I\vec{u} \\ \alpha Y^T\vec{u} \end{bmatrix} = \begin{bmatrix} (\lambda + \alpha^2)\vec{u} \\ \alpha Y^T\vec{u} \end{bmatrix}$$

So as long as $\vec{u}$ is not in the null space of Y^T, no vector with zero support in the Y^T domain could be an eigenvector. So the eigenconcepts of the bridge-blended data *must* be determined by both matrices. Exchanging the roles of X and Y^T, the same argument shows that eigenfeatures also must have cross-domain support.

Chapter 4

ProcedureSpace Reasons Jointly over English and Code

4.1 Overview

ProcedureSpace, the code-search backend for the Zones programming interface, blends background semantic knowledge about the natural language concepts that programmers use to describe their goals with static analysis of the programs that they write to accomplish those goals. The result is a representation that unifies syntactic knowledge about programs with semantic knowledge about goals. Figure 4-1 illustrates some of the relationships that such a combined representation uses.

natural language descriptions

background knowledge

follow
chase

kind of movement
opposite of *lead*

```
(forever
  (pointTowards: "mouse")
  (forward: 10))
```

forever > pointTowards:
forever > forward:
pointTowards: ~ forward:

code fragments

static analysis
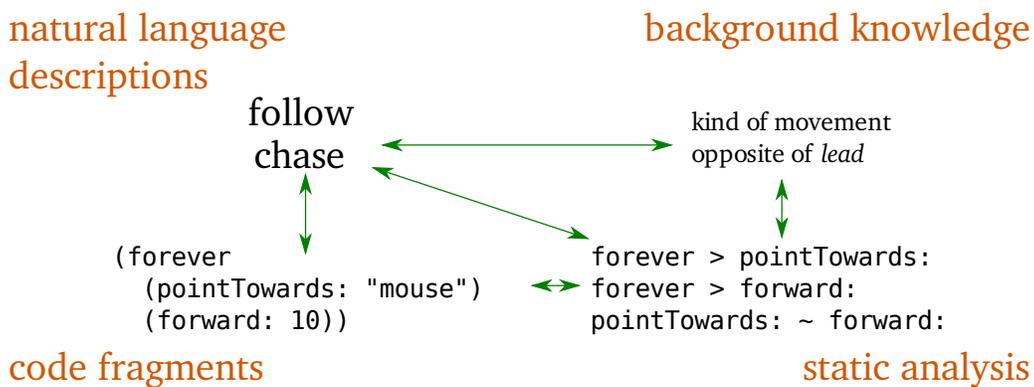
Figure 4-1: Diagram of a representation of "follow" that blends natural language and programming language

ProcedureSpace understands words like "follow" and "chase" by relating them to commonsense background knowledge (such as "follow is a kind of movement"), examples of code that people have said causes something to chase something else, and common characteristics of the structure of that code. Such a representation has many uses, but this chapter will focus on two: retrieving code using annotations, and retrieving annotations using code.

How can we find code in a corpus given a natural language description of what purpose it should accomplish? In the rare event that the desired code was described by exactly the same description as was searched for, finding it would be as simple as a dictionary lookup. But perhaps the descriptions differ in word choice. Or perhaps the desired code has no annotation at all. Nevertheless, by reasoning jointly over code and words[1] and incorporating additional information about how both the code and the words that apply to it may be related, the desired code may still be found. The ProcedureSpace technique not only improves answers to traditional code-search questions, but enables new types of questions, such as: What annotations might apply to this code? What other code is similar to this?

4.1.1 Reasoning Strategy

ProcedureSpace works with six datasets that connect five different types of data: English concepts, relations between them, English purpose descriptions, code, and characteristics about that code (called *structural features* throughout this chapter). Table 4-1 summarizes these datasets. This chapter discusses how each dataset is computed and how ProcedureSpace uses them.

ProcedureSpace uses the Blending technique, described in background section 3.2.4, to reason jointly across data of different kinds. The core technique takes a matrix, discovers its most important dimensions, and organizes entities along those dimensions in a semantic space. So the first step for any data will be to arrange it as a matrix. Blending works by arranging the individual matrices into a single matrix. An important parameter for the Blending technique is the layout of the data matrices; ProcedureSpace

---

[1]In this chapter, "words" abbreviates "words or phrases."

Table 4-1: Descriptions and sizes of data going into ProcedureSpace in matrix form

Description	Rows		Columns		# Items
	Kind	#	Kind	#	
CS: code structure	structural features	14145	scripts	127473	2721689
AD: annotations	purpose phrases	100	scripts	126	174
AW: annotations as concepts	words	143	scripts	126	429
WC: words in code	words	5639	scripts	86519	208016
DS: domain-specific knowledge	words	19	NL features	20	24
$CNet$: ConceptNet	words	12974	NL features	87844	390816

uses a *transposed bridge blend* layout, described in section 3.2.5. Figure 4-2a diagrams the basic bridge blend layout: effectively, the purpose annotations bridge the structural features derived from static analysis with the natural language background knowledge in ConceptNet. That diagram is simplified, however: as Figure 4-2b shows, the bridge is actually a sub-blend of purpose annotations with identifiers from the code, and the annotations are expressed as both complete strings and their constituent English concepts. (Even this diagram misses some details; see Figure 4-8 for the actual layout.)

4.1.2 Organization

Sections 4.2 and 4.3 walk through how to use the basic dimensionality reduction technique to identify what code is similar. Sections 4.4 through 4.7 then show how adding in other data, using the Blending technique, improves and extends the analysis. Then Section 4.8 explains how to use the full ProcedureSpace blend to ask various questions, such as what code might accomplish a given goal. Finally, section 4.9 presents and discusses results of the complete analysis.

4.2 Code Structure

Let's first consider analyzing the code itself. Many advanced techniques have been developed for static source code analysis, but I take a simplified approach to static analysis in order to more clearly show how to combine static code analysis with natural language. The result will be akin to a quick glance at the code, rather than an in-depth study. The basic goal of the code analysis is *similarity detection*: in order to find
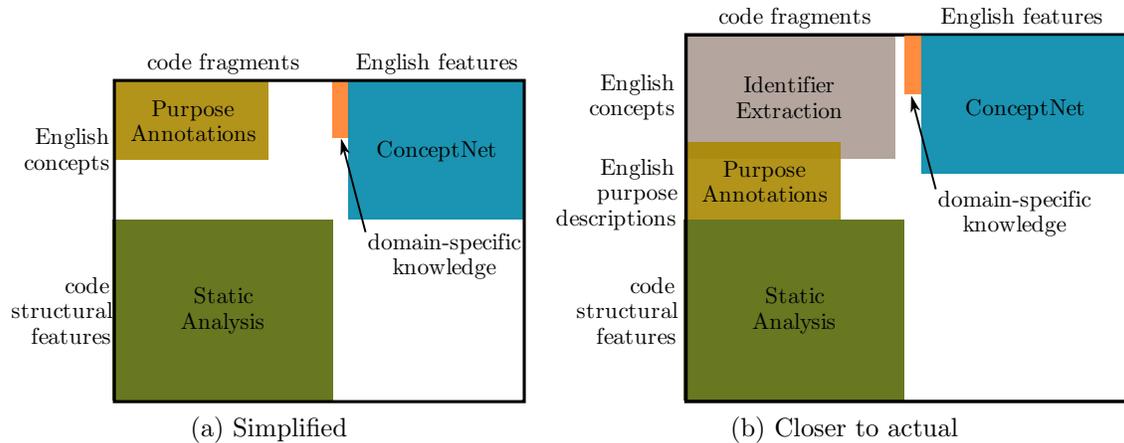
Figure 4-2: ProcedureSpace matrix, showing the data sources and their overlap in rows and columns. This diagram is somewhat simplified; see Table 4-1 and Figure 4-8 for full details.

what code is relevant to a goal, it will be helpful to understand how similar two code fragments are. However, the intermediary results of the analysis will prove useful for more than just code similarity.

4.2.1 Structural Feature Extraction

Code with similar function often have similar structural features. For example, many different examples of code that handles gravity (or "falling") all include a movement command conditioned on touching a color in the sky (or not touching a color on the ground). Those that actually simulate acceleration due to gravity will all have code that conditionally adds to a variable. In other languages, such features could also include type constraints (e.g., "returns an integer") or environmental constraints (e.g., "uses synchronization primitives").

In extracting structural features, ProcedureSpace treats the code as an untyped tree. For code units that take parameters, such as `forward:`, the parameters are treated as children. If the parameter is an expression such as `1 * 2 + 3`, the outermost function call (`+`, in this case) is a direct child, while the other elements are descendants. Conditional or looping constructs can contain an arbitrarily long sequence of children.

For each code fragment, ProcedureSpace extracts various types of simple structural
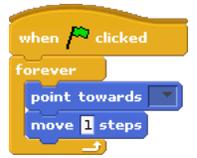
(a) as presented in the Scratch UI

(b) internal S-expression format

```
(EventHatMorph "Scratch−StartClicked"
 (doForever
  (pointTowards: "hero")
  (forward: 1)))
```

Figure 4-3: Example "chase" code fragment, taken from "Enemy AI tutorial: Chase."

features about what kinds of code elements are present and how they are related:

Presence A particular code element is present somewhere in the fragment (e.g., `doForever`)

Child A code element is the direct child of another code element (e.g., `FlagHat > doForever`[2])

Containment A code element is contained within another code element, either as a parameter or as the body of a conditional or looping construct (e.g., `FlagHat pointTowards:`)

Clump A clump of code elements occur in sequence (e.g., `[forward_ pointTowards_]`[3])

Sibling A particular code element is the sibling (ignoring order) of another code element (e.g., `forward_ ~ pointTowards_`)

Within these types of features, it is not necessary to enumerate all possible features beforehand. Rather, for each feature type, an extraction routine generates all the features of its type that apply to a particular code fragment.

Consider the code fragment in Figure 4-3, which makes a sprite chase or follow another sprite. Table 4-2 shows examples of the code structural features extracted for the example code.

---

[2]This notation is based on CSS3 Selectors.

[3]Underscores replace colons in structural features.

Table 4-2: Code structural features for the example code

Child	`FlagHat > doForever`
Child	`doForever > pointTowards_`
Child	`doForever > forward_`
Sibling	`forward_ ~ pointTowards_`
Clump	`[forward_ pointTowards_]`
Presence	`doForever`
Presence	`FlagHat`
Presence	`forward_`
Presence	`pointTowards_`
Containment	`FlagHat doForever`
Containment	`FlagHat forward_`
Containment	`doForever forward_`
Containment	`doForever pointTowards_`
Containment	`FlagHat pointTowards_`

4.2.2   Matrix Construction

From the extracted code structure features, I construct a matrix CS that relates code fragments to the structural features it contains. The rows of this matrix are the all of the 14145 distinct code structure features that were extracted; the columns are the 127473 analyzed code fragments. (The order of the rows and columns does not matter for this kind of analysis.) The entries in the matrix are the degree to which a structural feature is present in a particular code fragment.

How do we assign a number to how much a code fragment has a particular feature? One option is to count the number of times that the feature occurs. Another (the "binary" approach) is to put a 1 in an entry if the feature appears in that code fragment at all. Since a feature occurring multiple times in a code fragment isn't necessarily the most important feature about that fragment (it could be an unrolled loop, for example), I chose the binary approach.

Any technique that extracts features from documents (in this case, code fragments) is vulnerable to length artifacts: a document has a strong effect on the analysis not for being a good match for a particular query, but simply for being long. The binary approach partially mitigates the effect of long code fragments, since their features can count at most once. But sometimes a programmer will combine many different

60

functionalities into a single Scratch script. This practice is not only considered to be bad style, but also precludes easily reusing one of the subparts. Yet these scripts will still be weighted higher because they have more different kinds of features than simpler scripts. So we normalize all scripts to have unit Euclidean norm: each column in the matrix is divided by the sum of the squares of its entries. That way, long scripts have their influence "diluted," as desired. Here is the final matrix CS:
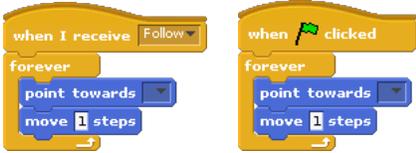
CS =			
Clump [forward_ pointTowards_]	0.27	0.27	$\cdots$
Presence EventHatMorph	0.11	0	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$

4.3   Dimensionality Reduction

Now that we have represented structural features of the code in a matrix, we can consider how to analyze that matrix to determine how code fragments and their structural features are similar.

4.3.1   Setup

This subsection repeats some of the setup of background section 3.2.3, but explains it in the context of code fragments and their structural features. It also explains the eigenvalue problem slightly differently. If you understand how the SVD applies to the matrix CS, you can skip this subsection.

The code structure matrix contains many hidden similarities. For example, since code element a being a child of element b implies that a contains b, corresponding Child and Containment rows will be very similar. (They will not be exactly the same because containment does not imply direct containment.) More importantly, code fragments that have similar features are probably themselves similar. One way to find out if two code fragments are similar is to find the dot product of the corresponding

columns. Since the columns have been normalized to unit magnitude, the dot product between any two columns will range from -1 to 1. The more similar two scripts, the closer the dot product of their columns is to 1.

The same logic applies to structural features, with one caveat. Consider a very common element, such as FlagHat or EventHat. Since these features apply to a large number of scripts, both FlagHat and EventHat will have high dot products with most other features, despite being very different in functionality. (The problem also affects the sprite similarity computation, which does not properly account for the contrast between the two features.) A straightforward solution is to subtract the mean of each row (structural feature) from that row. This operation, done after the column normalization, results in the column norms deviating from unity slightly, so the dot products will no longer be exactly within the range -1 to 1.

As we have seen, the dot products of rows with other rows, or columns with other columns, determine how similar the corresponding items are. We can write those dot products compactly as a matrix multiplication. Let A be the matrix we're analyzing (in this case, $A = \text{CS}$). Then AA^T contains the row-to-row similarity: each entry (i, j) is the dot product of row i and row j—how much having structural feature i is like having feature j. Since the dot product is commutative, the matrix is necessarily symmetric.

There's another way to think about AA^T, though. Consider a particular code fragment: the "chase" example code fragment, for example. It's represented by a column of A, that is, by how much each code structure feature applies to it. We'll represent that column of weights as a vector $\vec{v}$. But those structural features apply to many other code fragments as well. The vector $A^T\vec{v}$ gives the dot products of $\vec{v}$ with each column, that is, how closely the pattern of features matches that code fragment. In other words, $\vec{v}$ describes a code fragment in terms of its structural features, and $A^T\vec{v}$ describes those structural features in terms of what code fragments they apply to. $AA^T\vec{v}$, in turn, describes those code fragments in terms of what features apply to them: what features do code fragments like this one have? If the "chase" code fragment were a totally stereotypical code fragment of its kind, then $AA^T\vec{v}$, would

equal $\vec{v}$ (perhaps scaled by some constant): it has exactly the code features that all other code fragments like it have.

The "chase" code fragment turns out to not be so stereotypical, but what code fragments are? A stereotypical code fragment would have a set of features $\vec{v}$ such that $AA^T\vec{v} = \lambda\vec{v}$, where λ is a constant. In other words, transforming $\vec{v}$ by A^T and then A only multiplies $\vec{v}$ by a constant. Those familiar with linear algebra will recognize that $\vec{v}$ is an eigenvector of AA^T, and the constant λ is an eigenvalue.

Here we see that the problem of finding stereotypical code fragments and features is set up the same as finding the eigenconcepts and eigenfeatures of ConceptNet, as described in background section 3.2.3. The remaining derivation, and most of the interpretation, is thus the same, so it will not be repeated here. The upshot is that we can now consider the similarity of code features to each other entirely in terms of how they apply to the "stereotypical" code fragments.

4.3.2   Results

Figure 4-4 shows the top 1000 singular values of the code structure matrix (and also shows that the mean subtraction process makes no significant difference to the singular values). While the top 50 or so singular values account for much of the variance, there is no clear point after which the values become insignificant. I decided to use 200 singular values for most of the ProcedureSpace analysis, since in some cases, I found that the inclusion of some of the lower axes made a significant improvement in the similarity of code fragments that were similar in some particular respects.

Figure 4-5a shows the code fragments that are most similar to the "chase" example, that is, have the highest dot products with its vector. These similar code fragments include variations on the original code fragment. In this case, the variations have code elements added or removed; in general, code elements could also be moved. (When we later combine this matrix with other kinds of data, we will see that sometimes the "similar" code may be superficially very different, depending on purpose annotations and word use.) Since code fragments and code structure features are projected into the same space, we can also look at the nearby code features, shown in Figure 4-5b.
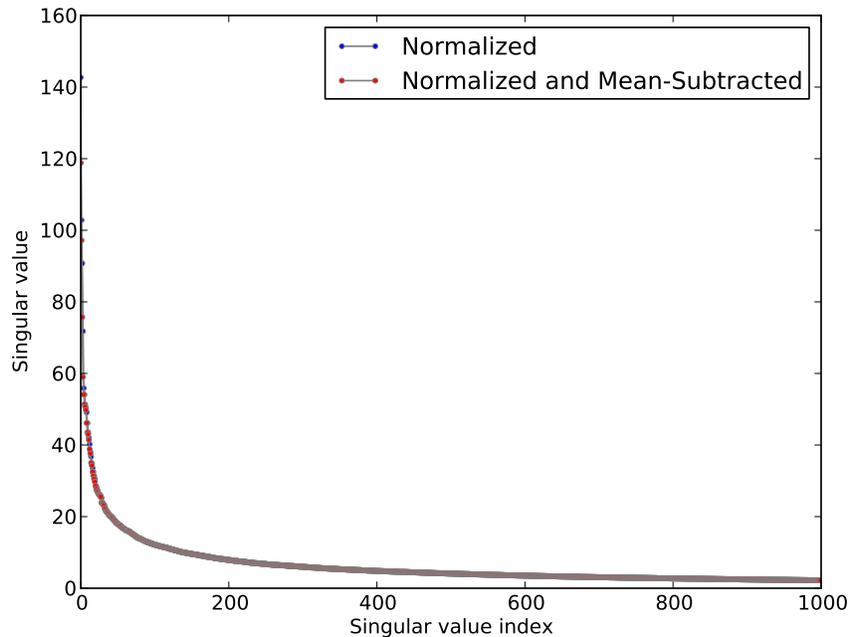
Figure 4-4: Singular values of the code structure matrix CS (normalized, with and without mean-subtraction)

We can see that the most similar features are generally those that the code element contains. If this code element were the only one analyzed, the "similar" features would be exactly its features.

We consider each eigenfragment or eigenstructure to be an axis along which actual code fragments and structural features are projected. One way of analyzing the results of the analysis, then, is to consider the items at the extremes of these axes. Figure 4-6 shows the extremes of the first two axes; since the sign of a singular vector is indeterminate, the figure does not label which end is positive or negative. The top axes reflect the *strongest* patterns, though not necessarily the most meaningful ones. The first axis is devoted to identifying the presence of two very common patterns: showing/hiding on command, and a do-nothing loop. The former is extremely common in animation; the latter is simply a very common (though mostly harmless) coding mistake. The second axis is devoted to discriminating between hiding or showing, and also between `FlagHat` and `EventHat`. Lower axes then refine that discrimination, but

0.957

0.836

0.824

0.797

0.786

0.783

0.770

0.757

0.750
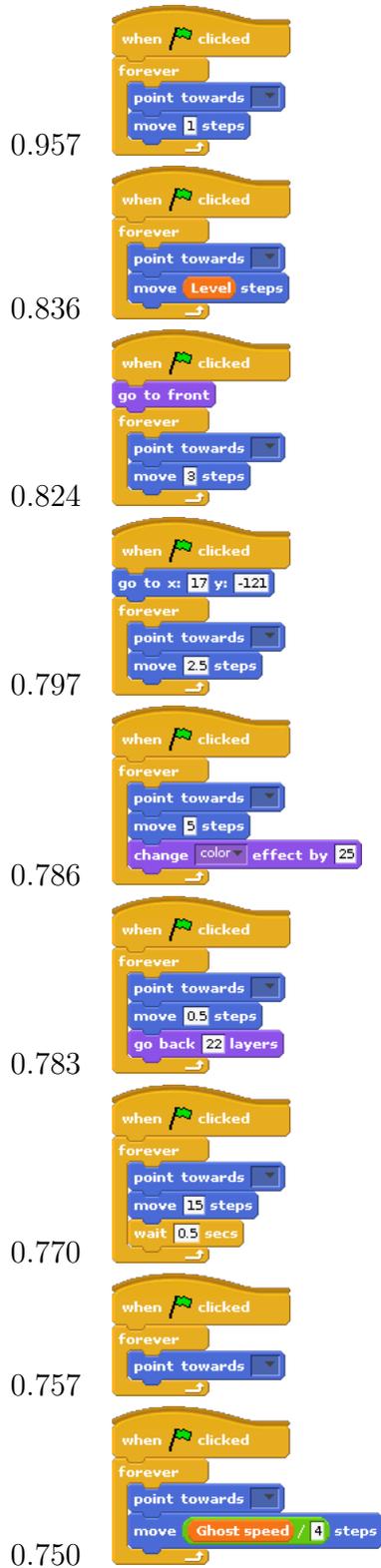
(a) Similar code fragments

13.162	**Presence** `FlagHat`
13.100	**Presence** `doForever`
12.739	**Containment** `FlagHat doForever`
12.730	**Child** `FlagHat > doForever`
6.584	**Presence** `forward_`
4.357	**Containment** `FlagHat forward_`
4.039	**Presence** `pointTowards_`
3.259	**Containment** `doForever forward_`
2.918	**Containment** `FlagHat pointTowards_`
2.431	**Containment** `doForever pointTowards_`

(b) Similar code structures

Figure 4-5: Code fragment similarity using code structure alone

65

the extremes of the axes become less clear because of the orthogonality constraint on the axes. That is, since singular vectors, and thus axes, must be orthogonal, a cluster that is not entirely orthogonal to another may have to be expressed as a linear combination of several axes, so the extremes of any one of those axes may include items from several otherwise unrelated clusters. This effect only hinders the crude visualization of looking at extremes of isolated axes; it poses no difficulty when working with complete vectors.

4.4   Blend: Incorporating Annotations

Our goal isn't really finding similar code; it's finding code relevant to a goal. Even the notion of similarity itself must be informed by a goal; otherwise, how can we know what features should be important? For example, we saw that the most similar feature to the "chase" code fragment was FlagHat. Consider Figure 4-7: in the analysis of code alone, the angle between the two code fragments is 48.7°, which is not particularly similar, because their distinction—the type of hat element—is weighted too highly. The way the behavior *gets started* (the FlagHat), though common among "chase" behaviors, is orthogonal to what the behavior *does* (point towards and move). We need some kind of extra data to make that distinction, and a technique to incorporate that data into the reasoning process.

One source of additional data is the annotations that people made about the purpose of code. In this section, I'll describe how to use Blending, a simple technique introduced and analyzed in background section 3.2.4, to reason over code structure and annotations simultaneously. This process will help identify which features are important markers for a goal, and take the first step in finding code relevant to a goal. First, I'll describe how we make a matrix from the annotations.

4.4.1   Data Extraction

I create a matrix AD of annotations by code from the annotations that programmers have given to each code fragment. For each annotation that comes from the Zones

Presence `EventHatMorph`
Containment `EventHatMorph hide`
Child `EventHatMorph > hide`
Presence `hide`
Child `EventHatMorph > show`

··· Containment `FlagHat hide`
Child `FlagHat > doForever`
Containment `FlagHat doForever`
Presence `doForever`
Presence `FlagHat`

(a) Axis 0

Presence `EventHatMorph`
Containment `EventHatMorph show`
Presence `show`
Child `EventHatMorph > show`
Containment `EventHatMorph`
`lookLike_`

··· Child `EventHatMorph > hide`
Child `FlagHat > hide`
Containment `FlagHat hide`
Presence `FlagHat`
Presence `hide`

(b) Axis 1

Figure 4-6: Items at the extremes of the principal code structure axes.

67

Figure 4-7: Two different code fragments with similar behavior. Annotating both as being for "chase" should make them more similar.

front-end, the server stores the username of the person who created the annotation, the id number of the script (i.e., some code fragment as used by some project), and the annotation text that the person gave to the code. I then construct the matrix AD by counting the number of times a script was given a certain annotation. The resulting matrix looks like:
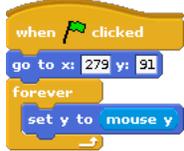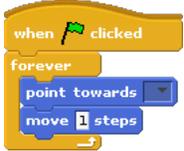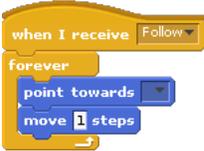
AD =				$\cdots$
Purpose `mouse control`	1.00	0	0	$\cdots$
Purpose `chase`	0	1.00	1.00	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

The purposes are actually stored as (`Purpose`, *purpose*) tuples to distinguish the full strings from extracted words, which they will get mixed with later.

I did the initial annotations myself; as development progressed, I was able to collect annotations from others. I describe those other annotations in Section 2.2.3 and Chapter 5.

4.4.2 Blending

How can we reason jointly over the two kinds of information we know about code features—code features and purpose annotations? We use a technique called Blending, described in background section 3.2.4. The setup is simple: just line up the labels and add the matrices (we'll later consider weighting the matrices before the addition). The matrices in question are the code feature matrix CS and the purpose annotation

matrix AD. In this section, let's use the shorter names $A = CS$ and $B = AD$. Then the blended matrix is:

$$C = \begin{bmatrix} A \\ B \end{bmatrix}$$

As before, the columns are code fragments; the rows are now code structure features followed by purpose descriptions.

Though simple in setup, Blending is difficult to analyze. Havasi gives some empirical analysis of the technique in [12]. Here, I take two steps towards a mathematical analysis, but a complete analysis must remain future work. Yet in this case math simply has not yet caught up to intuition, since our group regularly makes productive use of the technique for many applications.

The code fragment similarity matrices are, as expected, simply the sum of the similarities of both parts:

$$C^T C = \begin{bmatrix} A^T & B^T \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} = A^T A + B^T B$$

However, the row similarities are more interesting, now that both code structural features and purpose annotations are on the same axis:

$$CC^T = \begin{bmatrix} A \\ B \end{bmatrix} \begin{bmatrix} A^T & B^T \end{bmatrix} = \begin{bmatrix} AA^T & AB^T \\ BA^T & BB^T \end{bmatrix} = \begin{bmatrix} AA^T & AB^T \\ \left(AB^T\right)^T & BB^T \end{bmatrix}$$

No Overlap Case

To probe what's going on, let's consider the case where no code fragments have both code features and purpose annotations (i.e., no overlap in the range of the two matrices). In that case, C would actually look like (where A' and B' are the contiguous portions of A and B):

$$C' = \begin{bmatrix} A' & 0 \\ 0 & B' \end{bmatrix}$$

Then the new similarity matrices are:

$$C'^T C' = \begin{bmatrix} A'^T & 0 \\ 0 & B'^T \end{bmatrix} \begin{bmatrix} A' & 0 \\ 0 & B' \end{bmatrix} = \begin{bmatrix} A'^T A' & 0 \\ 0 & B'^T B' \end{bmatrix}$$

$$C' C'^T = \begin{bmatrix} A' & 0 \\ 0 & B' \end{bmatrix} \begin{bmatrix} A'^T & 0 \\ 0 & B'^T \end{bmatrix} = \begin{bmatrix} A' A'^T & 0 \\ 0 & B' B'^T \end{bmatrix}$$

Now suppose $\vec{v_A}$ is an eigenvector of $A'^T A'$ (i.e., $A'^T A' \vec{v_A} = \lambda_A \vec{v_A}$), and $\vec{v_B}$ an eigenvector of B'. Then we can see that $\begin{bmatrix} \vec{v_A}^T & 0 \end{bmatrix}^T$ and $\begin{bmatrix} 0 & \lambda_B^{-1} \vec{v_B}^T \end{bmatrix}^T$ are eigenvectors of $C'^T C$:

$$C'^T C' \begin{bmatrix} \vec{v_A} \\ 0 \end{bmatrix} = \begin{bmatrix} A'^T A' & 0 \\ 0 & B'^T B' \end{bmatrix} \begin{bmatrix} \vec{v_A} \\ 0 \end{bmatrix} = \begin{bmatrix} A'^T A' \vec{v_A} \\ 0 \end{bmatrix} = \lambda_A \begin{bmatrix} \vec{v_A} \\ 0 \end{bmatrix}$$

By the same reasoning, the eigenvectors of $C' C'^T$ are also just the zero-padded eigenvectors of the original matrices, with the same eigenvalues. In other words, no column of U or V will have nonzero entries from both A' and B'. So an axis will contain either structural features or purpose annotations, never both.

Overlap Case

Under what circumstances could an axis have both structural features and purpose annotations? We have seen that if there is no overlap in the input, there can be no overlap (in terms of the composition of the axes) in the output. Intuitively, we should expect that overlap in the matrix produces overlap in the axes, and indeed this is generally the case. Returning to the original augmented matrix C, remember that the row similarity matrix CC^T was $\begin{bmatrix} AA^T & AB^T \\ (AB^T)^T & BB^T \end{bmatrix}$, which includes a cross term AB^T. An element (i, j) of AB^T is the dot product of the vector of code fragments to which code feature i applies with the vector of code fragments to which purpose annotation j applies. In the no-overlap case, AB^T was 0; if A and B overlap at all, AB^T is nonzero. Though it has not yet been demonstrated, I conjecture that the

degree of overlap in axes depends on some measure of the magnitude of AB^T relative to AA^T and BB^T.

4.4.3 Example

Let's work a small example first. Consider if we annotated both of the code fragments in Figure 4-7 as being for "chase." Intuitively, we should expect that since we're adding to what they have in common, they should come out more similar in the analysis—and in fact, that's what happens. Without annotations, the angle between the two code fragments was 48.7°. Now let's make a new matrix of just the two annotations:

anno =

chase	1.0	1.0
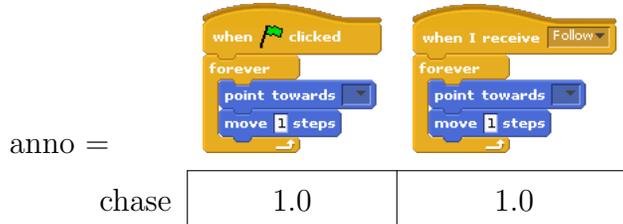
Now we blend the code structure matrix with this annotation matrix[4], and the angle goes down to 48.5°. That's a difference, but why so small? Consider the entire blend matrix, a subsection of which is shown in Table 4-3. (Recall that the mean of each row is being subtracted out.) With two entries of not quite 1.0, the "chase" annotations are already starting to get lost within the tiny subsection shown. Compared to all 14145 structural features, the two annotations have a negligible effect on the structure of the vector space.

We can cause the annotations to have a larger effect by weighting them more heavily. The weight of the second matrix in a two matrix blend is called the *blending factor*[12][5]:

$$C = (1 - f)A + fB$$

In the micro-blend we've been working with, both matrices were weighted equally,

---

[4]Blending mean-subtracted results requires careful implementation. I implemented sparse mean subtraction by computing with row and column offsets in the Lanczos matrix multiplications in Divisi. But if the blending component is not aware of the offsets of the blended matrices, it could either ignore them or use them incorrectly. In fact, if the axis of blending is the same as the axis of offset, it's impossible to express the resulting offset in terms of row and column offsets. I work around this problem by mean-subtracting *after* the blend.

[5]This equation uses labeled matrix operation notation: before performing the operation, the labels are aligned, padding missing rows and columns with zeros.

	when 🚩 clicked forever point towards move 1 steps	when I receive Follow forever point towards move 1 steps	· · ·
Child `FlagHat > doForever`	0.24	0.00	· · ·
Child `EventHatMorph > doForever`	0.00	0.26	· · ·
Child `doForever > pointTowards_`	0.27	0.27	· · ·
Child `doForever > forward_`	0.27	0.27	· · ·
Sibling `forward_ ~ pointTowards_`	0.27	0.27	· · ·
Clump `[forward_ pointTowards_]`	0.27	0.27	· · ·
Presence `doForever`	0.22	0.22	· · ·
Presence `FlagHat`	0.15	0.00	· · ·
Presence `EventHatMorph`	0.00	0.11	· · ·
Presence `forward_`	0.26	0.26	· · ·
Presence `pointTowards_`	0.26	0.26	· · ·
Containment `FlagHat doForever`	0.24	0.00	· · ·
Containment `EventHatMorph doForever`	0.00	0.26	· · ·
Containment `FlagHat forward_`	0.26	0.00	· · ·
Containment `doForever forward_`	0.26	0.26	· · ·
Containment `EventHatMorph forward_`	0.00	0.26	· · ·
Containment `EventHatMorph pointTowards_`	0.00	0.27	· · ·
Containment `doForever pointTowards_`	0.27	0.27	· · ·
Containment `FlagHat pointTowards_`	0.27	0.00	· · ·
chase	1.00	1.00	· · ·
⋮	⋮	⋮	⋱

Table 4-3: Section of the blend matrix, equal weights

so the effective blending factor was 0.5. If we instead set the blending factor to 0.9 and re-run the SVD, the angle between the two "chase" code fragments plummets to 4.3°. As we can see, the blending factor (or matrix weights in general) is an important parameter determining how much different types of data affect how the vector space is created.

4.5   Words with Background Knowledge

The example was idealized because the purpose annotations matched exactly. Realistic annotations differ in many ways, from punctuation to extra words, but the most difficult situation to handle is when people choose entirely different words. For example, perhaps the first code fragment got annotated "chase," but the second code fragment was labeled "follow." Since the annotations don't overlap, the annotations wouldn't cause the code fragments to move closer together as they did in section 4.4.3. But we have good reason to think that "chase" and "follow" are similar: they're both activities where the actor is behind the object. If we could encode background knowledge like that about words, might we be able to use the similarity between "chase" and "follow" to conclude similarity between what was *annotated* by the two words?

4.5.1   Bridge Blending

We can use the Bridge Blending layout, described in background section 3.2.5, to connect the word knowledge with the code knowledge. Suppose we knew two pieces of background knowledge about both "chase" and "follow": people can do them, and they are movements. We could encode that knowledge as *person*\CapableOf/___ and ___\IsA/*movement*. Let's set up a small bridge blend with that data:

			person\\`CapableOf`	\\`IsA`/*movement*	
Clump [`forward_ pointTowards_`]	0.03	0.03	0	0	⋯
Presence `EventHatMorph`	0.03	0	0	0	⋯
chase	0	1.00	1.00	1.00	⋯
follow	1.00	0	1.00	1.00	⋯
⋮	⋮	⋮	⋮	⋮	⋱
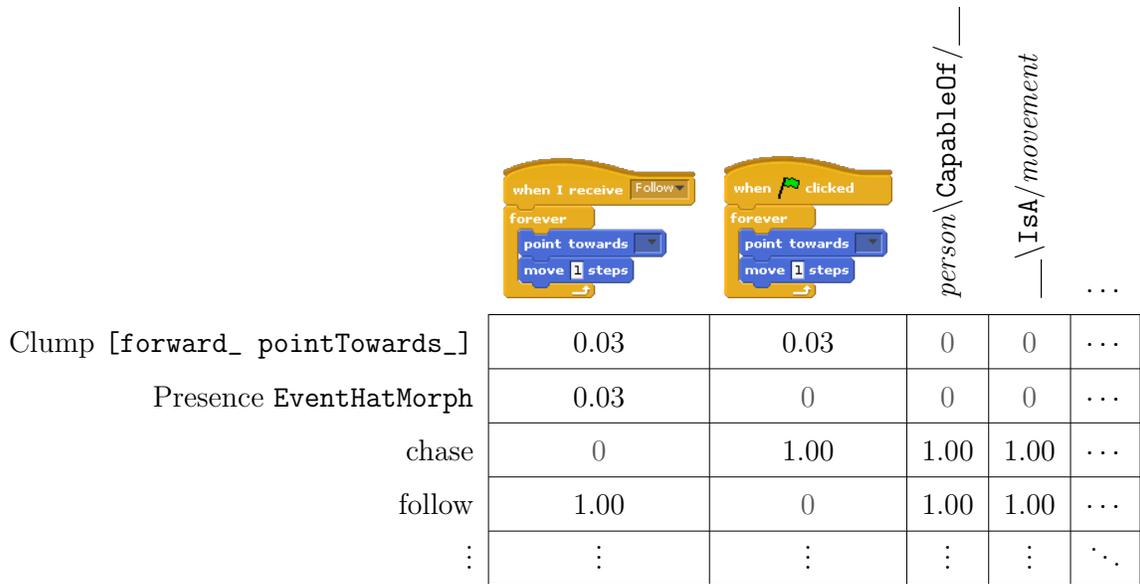
Here, I've weighted the code structure features by 0.1 and the annotation and background knowledge by 1.0. We compute the SVD as before, and the angle between the two code fragments becomes 8.0°. Recall that without any blending, the angle was 48.7°, so the background similarity information was definitely used.

4.5.2 Background Knowledge Sources

I've shown that background knowledge about the relationships between words helps connect annotations that are not superficially related and thus helps "warp" the code structure analysis so as to put code described by similar annotations close together. But what background knowledge is useful, and where can we get it?

Domain-Specific Knowledge

One kind of knowledge is domain-specific knowledge about the programs that are being built. For Scratch, many of the programs are games, so domain-specific knowledge includes facts such as "arrow keys are used for moving" and "moving changes position." Such knowledge would enable us to relate an annotation about "arrow keys" with an annotation about "position," for example. Table 4-4 shows a sample of this knowledge, which for now was manually entered.

Concept	Relation	Concept
arrow keys	`UsedFor`	move
arrow keys	`UsedFor`	control
costume	`UsedFor`	animation
costume	`UsedFor`	look
spin	`ConceptuallyRelatedTo`	around
spin	`ConceptuallyRelatedTo`	circle
fade	`HasProperty`	gradual
bounce	`HasSubevent`	hit
move	`Changes`	position
button	`ReceivesAction`	click
win	`Causes`	game over
lose	`Causes`	game over

Table 4-4: Sample of domain-specific knowledge

ConceptNet: General Knowledge

Another kind of knowledge is general world knowledge, such as "balls can bounce" and "stories have a beginning." Without such knowledge, the system may be entirely unaware that an annotation of "bounce" may be relevant to find code for "moving the ball." The ConceptNet project, discussed in background section 3.2.2, provides a large database of broad intuitive world knowledge, expressed in a semantic network representation (e.g., *ball*\\`CapableOf`/*bounce*). (Though ConceptNet includes data from many languages, I only use the English data for this work.) Rarely is a single ConceptNet relation a critical link in connecting two concepts; rather, the broad patterns in ConceptNet, such as which features typically apply to things that people desire or can do, help to structure the space of English concepts.

Matrix Encoding

Both ConceptNet and the domain-specific knowledge base are expressed as triples: *concept1*\\`relation`/*concept2*. To form a matrix out of these triple representations, we use the approach of AnalogySpace (see background section 3.2.3): for each triple, add both (*concept1*, ___\\`relation`/*concept2*) and (*concept2*, *concept1*\\`relation`/___) The columns of this matrix are called *features*. The double-encoding means that *arrow keys*\\`UsedFor`/*moving*, for example, contributes knowledge about both "arrow keys"

and "move." For ConceptNet, connections that the community rated more highly are given greater weight; for the domain-specific knowledge, I currently weight all entries equally as 1.0.

The token extraction is done with standard natural language processing techniques: word splitting, case normalization, spelling correction (using the hand-crafted ConceptNet 4 auto-corrector; see [13]), lemmatization (using the MBLEM lemmatizer[32]), and stopword removal.

The domain-specific knowledge can employ any relation, not just those that are used in ConceptNet. But the more we know about a feature, the more useful it is. So to maximize the overlap with ConceptNet, we should map any new domain-specific relation back onto an existing ConceptNet relationship where possible. For example, for the triple $move$\Changes/$position$, which uses the relation Changes that is not in ConceptNet, we might also add the triple $move$\Causes/$change\ position$.

CNet =	__\IsA/$animal$	$person$\Desires/__	$\cdots$
dog	0.50	1.16	$\cdots$
cat	0.50	0.79	$\cdots$
toaster	0	0	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$

DS =	__\UsedFor/$move$	__\UsedFor/$control$	__\HasSubevent/hit	$\cdots$
arrow key	1.00	1.00	0	$\cdots$
bounce	0	0	1.00	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

4.6 Words in Code

Ideally, every code fragment that a programmer would ever want would be annotated exactly as he/she would describe it. But in practice, only a small fraction of code may ever be annotated, and the annotations rarely match exactly. In this section, I'll discuss two other ways to glean linguistic data for code fragments.

4.6.1 Annotation Words

"Bounce off platform," "BouncePlatform," "platform bounce," ... are all different from the point of view of string equality, but you don't need much background knowledge to know that they're nonetheless highly semantically related. To help these line up, I extract *tokens* from each purpose description. Then in addition to the full purpose descriptions, I relate the code fragments with those tokens as well. Since the tokens will apply to the same code as the annotations that contain it, they'll come out similar in the analysis unless they also apply to very different code also. In either case, code that has related tokens will be pulled together.

The token extraction is exactly the same as that done for ConceptNet, so that the tokens line up as much as possible. However, I first split underscore-joined and camelCased strings. Also, I split multi-word phrases into individual words, including each word individually and any bigram (sequential pair of words) that also appears in ConceptNet or the domain-specific knowledge.

mouse	0.58	0.58	0.58	0	0	$\cdots$
chase	0	0	0	0.71	0.71	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$
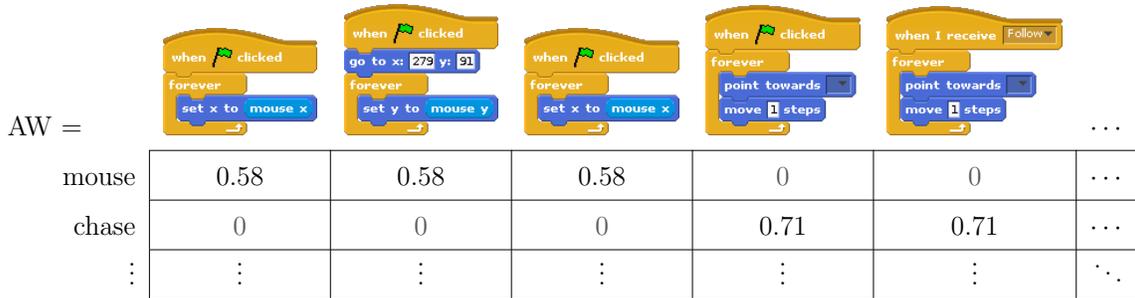
4.6.2 Identifiers

Next, ProcedureSpace also relates words to the code fragments in which they occur, much like how a traditional search engine indexes a corpus of documents. Specifically, we extract natural language tokens from identifiers in the code fragments—names of variables and events that the programmer defined—using the same procedure as in the previous section[6].

Ideally I'd also use comments and other documentation as a source of tokens; the analysis mechanism would be very similar. But Scratch comments and project

---

[6]We also name the code elements as if they were function calls, but that's currently disabled.

descriptions are associated with the entire sprite or project, not with individual scripts. Future work may incorporate them by associating them loosely with the scripts that they possibly apply to—but even so, helpful comments and meaningful project descriptions are rare in Scratch. (I actually did include this data in earlier work, but it did not prove helpful enough to maintain its use.)

I then fill in the number of occurrences of a word in a given code fragment in the corresponding element of a matrix WC (Words in Code).[7] Here is a representative sample (already normalized as described in the next section):

WC =			
score	0.35	0	$\cdots$
follow	0	7.93	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$
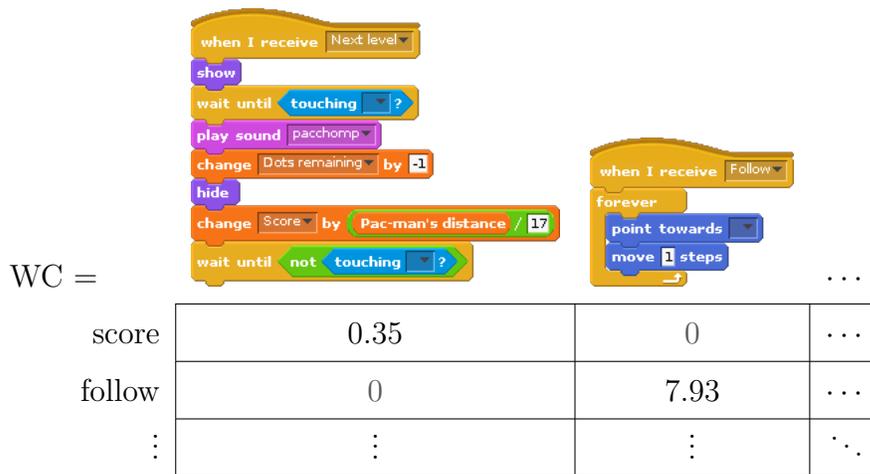
4.6.3   Term-Document Normalization

If we think of code fragments as if they were documents, then AD (annotation descriptions), AW (annotation words), and WC (words in code) report the how many times some "term" appears in that document. These raw counts are unsuitable for analysis for two reasons. First, some documents (code fragments) are simply longer than others, so more terms will occur in them, giving them disproportionately greater weight in the analysis. Also, terms that occur in nearly all documents (such as the word "start") get weighted much more heavily than terms that occur rarely, despite the fact that rare terms can be just as distinctive. So we employ the standard tf-idf (term frequency–inverse document frequency) normalization to those matrices. (This normalization is different from the vector-magnitude normalization described in Section 3.2.3.) For a matrix $A(term, doc)$ of terms and documents, an entry in the

---

[7]Readers familiar with information retrieval will recognize this as a term-document matrix.

normalized matrix $\tilde{A}$ is given by

$$\tilde{A}(term, doc) = \text{tf}(term, doc)\text{idf}(term)$$

where tf is the term frequency—the occurrence count normalized by the total number of terms in the document:

$$\text{tf}(term, doc) = \frac{A(term, doc)}{\sum_{t \in T} A(t, doc)}$$

and idf is the inverse document frequency:

$$\text{idf}(term) = \log \frac{\text{num documents}}{\text{num documents containing } term}$$

Though the set of documents (code fragments) is the same for each matrix, the document lengths have different units (number of words or annotations), so the normalization must be performed separately.

4.7   Full ProcedureSpace Blend

Now that we've covered all of the sub-parts of ProcedureSpace, the next step is to blend them together.

We visualize the combined matrix using a figure called a "coverage plot," which shows which input matrix different parts of the output matrix came from, the relative density/sparsity, and relative magnitudes. Figure 4-8 shows a coverage plot of the combined ProcedureSpace matrix. Rows (corresponding to words, annotations, and code features) are horizontal; columns (corresponding to code fragments and English features) are vertical. Colors indicate which matrix the entry came from. Darker pixels are more densely filled with entries. Labels are allocated to rows or columns one matrix at a time. So the vertical division of the ConceptNet matrix illustrates the degree of overlap in terms between ConceptNet and the terms extracted from code (WC). Some matrices, such as the annotations and domain-specific knowledge, are
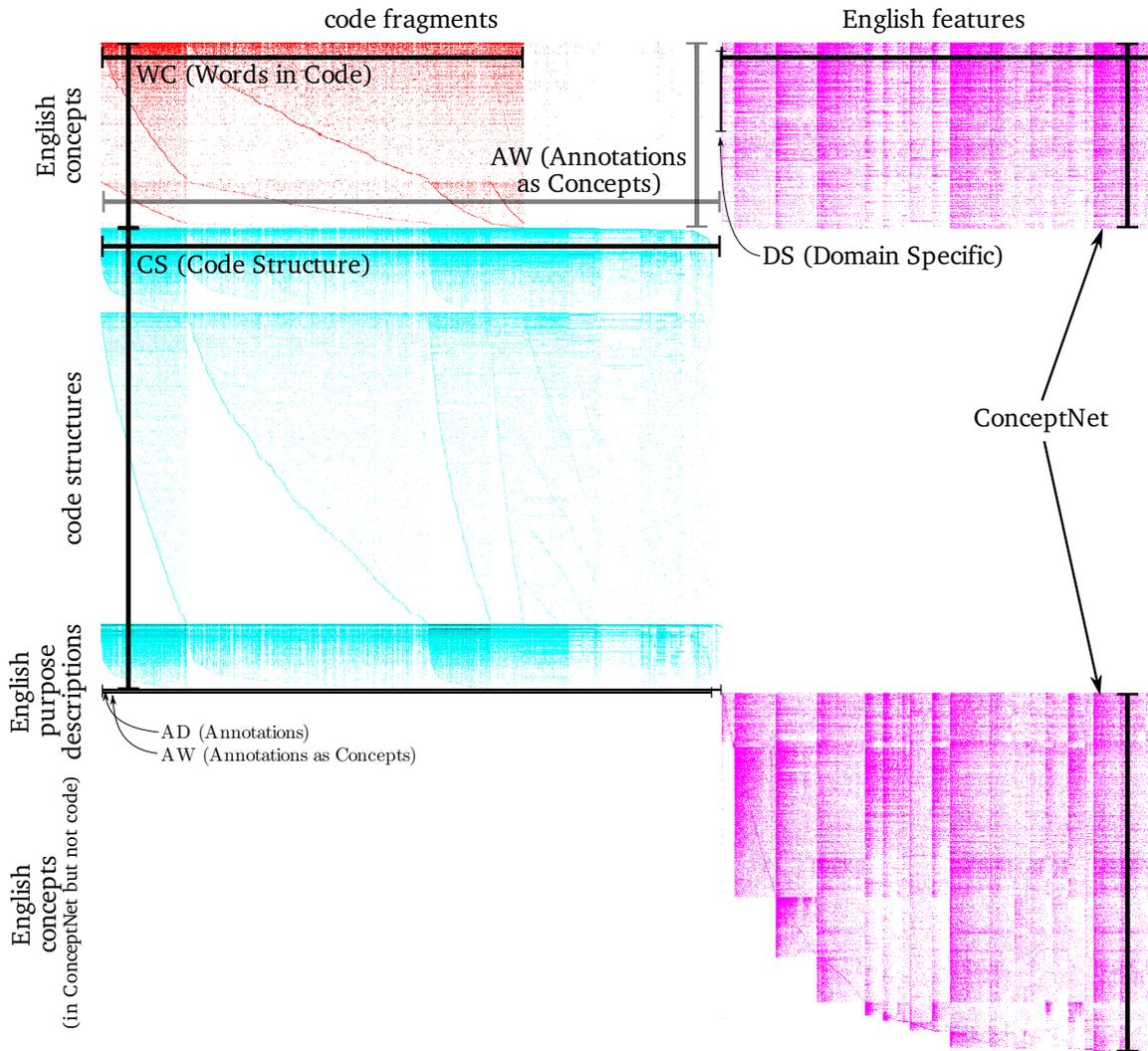
Figure 4-8: Coverage image for the combined ProcedureSpace matrix

relatively very small, so they are indicated with arrows. The internal substructure of the code structure matrix and other matrices is due to the multi-stage processing approach.

What weights should each matrix have? The optimal weights depend on the problem to be solved. Weighting CS (code structure) more would cause code characteristics to be the primary influence on the space, which would be helpful for exploring small variations on a code fragment. On the other hand, weighting AD (annotation phrases) more would cause purpose descriptions to be the primary influence on the space, which would be helpful for exploring diverse approaches to solving a problem. Weighting the

Matrix	Weight
WC (words in code)	0.100
CS (code structure)	0.100
AD (annotations)	0.900
AW (annotations as concepts)	1.800
DS (domain-specific)	0.100
$CNet$ (ConceptNet)	0.010

Table 4-5: Weights for each of the sub-matrices in the ProcedureSpace blend

background knowledge more would indirectly weight the annotations more. So the "best" weights are not yet known. For now, I set the blending weights manually and somewhat arbitrarily to the values shown in Table 4-5. Some of the results suggest that these blending factors were not ideal; the WC matrix seems to have had too much influence. Future work on the blending process will develop a better mathematical grounding for the effect of weights in order to estimate them better; future work on ProcedureSpace may develop a method for learning the best weights from the data.

Normalization and weight-setting was the hardest part of this system for me to get right. The most significant cause of this difficulty was shortage of annotation data; our group's experience with AnalogySpace shows that, like in many machine learning problems, the more data you have, the less careful you have to be about how you treat it. Another cause is that blending is still overly sensitive to relative magnitudes of the various input data. For example, a change in the normalization method of one matrix in a blend affects its overall magnitude, which alters its relative effect on the blend. Also, normalization has two linked effects: it warps the analyzed vector space, and also changes the relative magnitudes of the resulting item vectors. Separating those two effects might make the results more reliable.

Figure 4-9 shows the singular values of the blended ProcedureSpace matrix; the top 200 singular values were used for the actual analysis. The angle between the two example code fragments after all this blending is 31.9°—not as close as when we had only that one annotation, but still closer than the code-structure-only angle of 48.7°.

Figure 4-10 shows the items at the extremes of the principal axis. Unfortunately, it's difficult to get a sense for the structure of the space by looking at individual
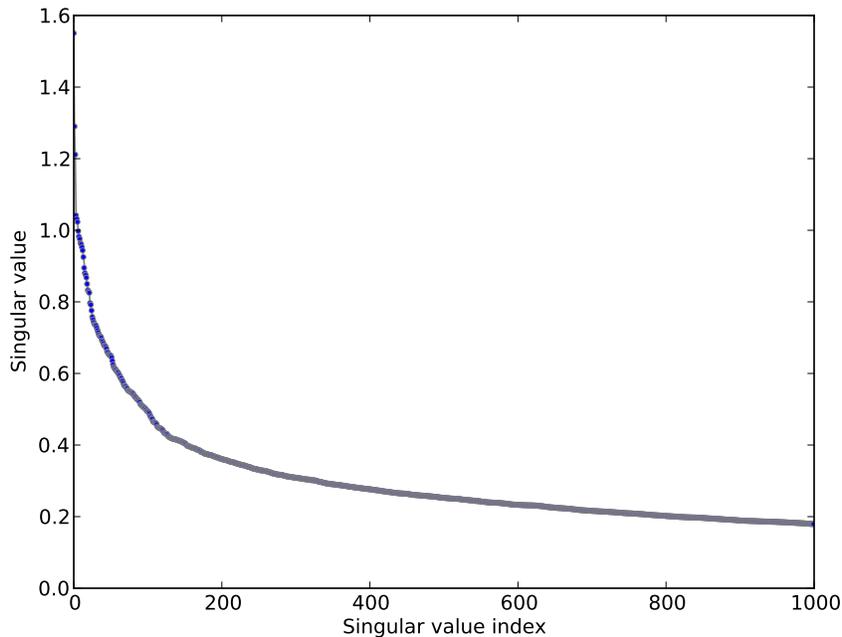
Figure 4-9: Singular values of full ProcedureSpace blend. The top 200 singular values were used for the actual analysis.

axes. Much like in the code-structure-only analysis of Figure 4-6, the principal axis shows what is most *common*, not necessarily what is most *meaningful*, and an axis viewed in isolation may confound several distinct clusters. One salient point is that the axis shown is strongly influenced by all of the blended inputs. For example, for code fragments, one extreme seems to capture whether or not the fragment was annotated (which suggests that I might want to revisit annotation normalization), whereas for structures, the same axis captures a code structure pattern—`hide` within `EventHat`—that was significant in the code structure analysis. So this one axis is composed of information from several different representations simultaneously.

4.8   Goal-Oriented Search

Once we have used blending to construct ProcedureSpace, the search tasks required to power the Zones interface become straightforward vector operations. Each entity is a vector in the k-dimensional vector space: the U matrix gives the position of each

Stacks	change visibility by -10	···	when ⚑ clicked
	volume		forever
	change costume by 1		if EGGMANS HP = 0
			broadcast You Win▾
			stop script
			when ⚑ clicked
			set ghost▾ effect to 100
			when ⚑ clicked
			forever if color ■ is touching ■ ?
			next costume

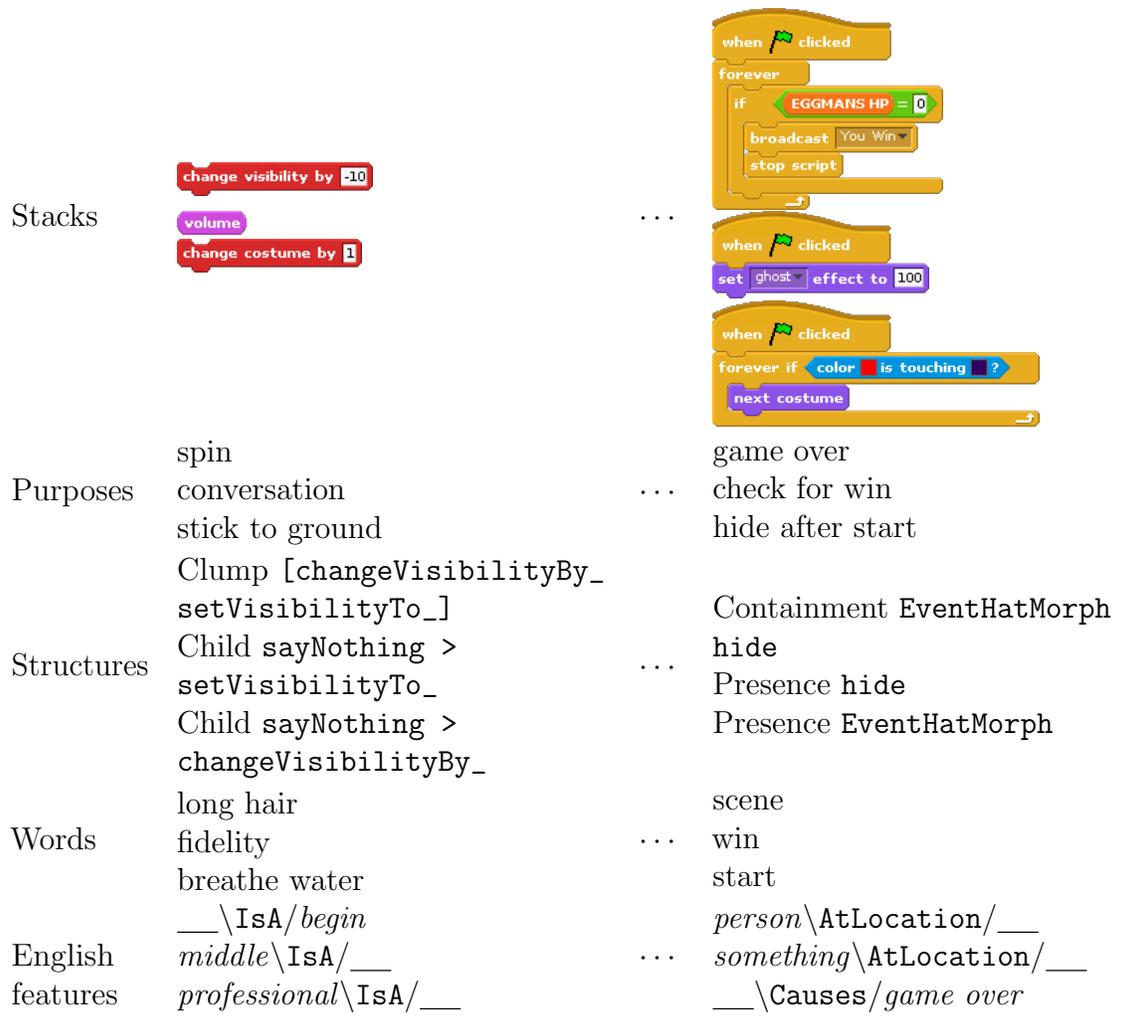|			
Purposes	spin	···	game over
	conversation		check for win
	stick to ground		hide after start
Structures	Clump [`changeVisibilityBy_` `setVisibilityTo_`]	···	Containment `EventHatMorph`
	Child `sayNothing` > `setVisibilityTo_`		`hide`
			Presence `hide`
	Child `sayNothing` > `changeVisibilityBy_`		Presence `EventHatMorph`
Words	long hair	···	scene
	fidelity		win
	breathe water		start
English features	___\\`IsA`/*begin*	···	*person*\\`AtLocation`/___
	middle\\`IsA`/___		*something*\\`AtLocation`/___
	professional\\`IsA`/___		___\\`Causes`/*game over*

Figure 4-10: Items at the extremes of axis 0 in ProcedureSpace

English word, purpose phrase, code feature; the V matrix locates scripts and English features.[8] The fact that all entities are in the same vector space means that search operations can be expressed as finding nearby vectors[9]. To find the vector $\vec{p}$ of a query string composed of English words w_i, you simply sum the corresponding vectors:

$$\vec{p} = \sum_{i=0}^{n} U\left[w_i, :\right]$$

where the : notation indicates a slice of an entire row. Then to find how well that description may apply to a particular script, you take the dot product of $\vec{p}$ and that script's vector (given by its row in V). In general, the weights for all scripts are given by $V\vec{p}$, considering only rows of V that correspond to scripts. The scripts with the highest values are returned as the annotation search results, after filtering to remove scripts that differ only in the values of constants.

Likewise, to find possible annotations given a fragment of code (goal identification), you extract its structural features f_i, form a vector $\vec{q} = \sum_{i=0}^{n} U\left[f_i, :\right]$, and find the words or annotations whose vectors have the highest dot product with $\vec{q}$.

To avoid returning results that are negligibly different, the SAMPLENEAR routine finds vectors that are near a search vector, avoiding clusters of nearly-identical things. Clusters are defined by cosine similarity. Consider a new item $\vec{p}$. Its angle to the original concept ($\vec{c}$, normalized to $\hat{c}$) is $\theta_{cp} = \cos^{-1} \hat{p} \cdot \hat{c}$. SAMPLENEAR will not include $\vec{p}$ if there exists an already-sampled item $\vec{m}$ such that $\frac{\theta_{mp}}{\theta_{cp}} > threshold$.

4.8.1 Approximating Textual Search

No code search technology currently exists for the Scratch corpus, but the words-in-code matrix WC is almost exactly the same as the standard Latent Semantic Indexing search technique would use. So to simulate a standard code search technique,

---

[8]For proper weighting, each dimension should be scaled by the corresponding singular value, so in the following discussion, U and V actually mean $U\Sigma$ and $V\Sigma$.

[9]We use "nearby" in approximately the sense of cosine similarity: two vectors are close if the angle between them is small. $\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos\theta$, so finding the $\vec{b}$ with the largest dot product finds the vector with the smallest angle, weighted by the magnitude of $\vec{b}$.

we can compute the SVD of the tf-idf-normalized term-document matrix WC. The text-to-code search procedure then proceeds identically to the ProcedureSpace search procedure, just with many fewer kinds of data.

However, while the methodology is similar, the results of this textual code search are impoverished compared to the results of such a technique on most other code corpora, simply because the Scratch corpus has relatively little semantic content in the code. For one, comments are even rarer in the Scratch corpus than in most code. Also, Scratch's event handling structures and concurrency enable complex functionality without requiring a single procedure to be named. So unfortunately a Zones annotation is often the first text associated with a code fragment. Perhaps, for a fair comparison with the full ProcedureSpace, I should have included the annotations in the words-in-code matrix, since in that respect they're like comments.

4.9   Search Results

In the user tests (detailed in Section 5), testers searched for a variety of goals and expressed them in a variety of ways. What did ProcedureSpace return? Figure 4-11 shows the top results for some of the queries that users performed. The first search, "*gravity*," returns first two code fragments that were annotated "gravity," illustrating that if an annotation matches exactly, the indirect reasoning through code structure and natural language background knowledge rarely disturbs those exact results. The later results for "gravity" match words in the code but are not generally relevant to the goal of making a sprite fall by imitating the force of gravity. This suggests that when this result set was computed, WC (the words-in-code matrix) was probably given too high an effective weighting in the blend relative to CS (the code-structure matrix), which would have given other approaches. Another cause of this problem is that with the number of annotations still very small, ProcedureSpace does not have enough data to distinguish true goal relevance from word co-occurrence. For "*follow player*"[10], again the exact match is returned, but starting with the third match, we

---

[10]The word "player" was a stopword for search queries.

85

see the effect of the blend including both AW (annotation words) and WC: "chase" is near "follow" in AW because they annotate some identical scripts, so scripts that contain the *words* "follow" and "chase" are also pulled closer together.

For a query like "score" where most programmers use identical vocabulary within the code (in this case, as a variable name), ProcedureSpace performs the same as text search. But for other queries, ProcedureSpace improves recall by including results that do not happen to include the exact search term but are nonetheless relevant.

4.10 Which-Does

In a project with many fragments of code, a programmer may want to find out which code performs a certain functionality, perhaps to fix a bug, change something, or just see how it works. With a slight modification, ProcedureSpace can help determine what code in a program performs a given functionality. As was done in goal-oriented code search (section 4.8), a ProcedureSpace vector for the purpose description is computed. But instead of being compared with all code fragments in the entire corpus, the vector is compared against only the code fragments within a project. In its simplest form, the search process returns the n code fragments that best match the query. A more refined form could treat the query as a classification problem: for each code fragment, how likely is it that it is involved in accomplishing the purpose queried?

Many existing code-search methods can be made project-local in a similar way. But the cross-modal representation of ProcedureSpace enables finding not just the code that contains the exact words of the query (e.g., as function names), but also code with similar words, and furthermore, also code with structural characteristics similar to those of code described by words like that. This capability could help reduce the entry barrier in contributing to open-source projects.

Figure 4-11: Code search results for selected user queries, top-ranked results on top.

Chapter 5

Users' Experience with the System

"Searching by goal is a really different way of programming." This was how one participant described her experience with the Zones/ProcedureSpace system. This chapter presents the process and results of a two-task experiment in which users interacted with the system. As expected, participants successfully used the Zones interface to find code that they could use in their project, and annotated both new and existing code in a variety of ways. But I was surprised by the number of different ways that people learned from their interactions with Zones. All participants understood the basic idea and were enthusiastic about it.

The experiments were aimed to answer these questions:

1. Does the Zones interface (both concept and implementation) help programmers make and use connections between natural language and programming language?

2. How do people describe the purpose of code?

3. Can ProcedureSpace find code that accomplishes a purpose they're searching for?

4. Do programmers annotate and reuse code more when interacting with Zones/ProcedureSpace?

5.1 Design

The first study was designed to address the first question: *Does the Zones interface (both concept and implementation) help programmers make and use connections between natural language and programming language?* In the process, it also provided some answers to the remaining questions. In this study, a small number of participants interacted directly with Zones in the lab. The Zones study was divided into two parts. In the first part, participants were familiarized with interacting with the Zones interface by annotating code and optionally trying to add functionality to an example project. The second part investigated how participants described behaviors that they observed by interacting, and evaluated whether they could (and would want to) use Zones searches to find code that performed those behaviors.

5.2 Procedure

In the first part, participants were instructed in the use of the Zones system in the context of the "*PacMan*" sample project. The intention was to collect the annotations they used for existing code, but they could also try adding behaviors. Part of the intention of the first part was to test the annotation-guessing functionality, but at the time of the user tests, annotation-guessing performed poorly.

The second part was a mimicry task, designed to test how people describe behaviors that they see and whether the Zones and ProcedureSpace search process could find code that matched those descriptions. Participants were shown an exemplar project[1] created especially for this task. It was designed such that many different but basically independent activities occurred simultaneously. Participants were instructed to identify and reproduce a few of those behaviors; there was no expectation that they reproduce all behaviors. They were given the identical project but with all scripts removed, so they would not need to be concerned with graphics. They were given Zones as a tool, but they were not required to use it.

---

[1]`http://scratch.mit.edu/projects/kcarnold/807416`

5.3   Results

In the first part, all participants were able to successfully use the Zones interface to annotate code, demonstrating basic usability of the interface. In the second part, all participants were able to successfully imitate at least one behavior from the exemplar project with the help of reused code from Zones, showing that their search queries were successful at finding code that is useful for reuse (though sometimes after several attempts). Finally, all participants left Zone searches as new annotations, demonstrating that the search-as-annotation paradigm can work within the Scratch programming environment. The following subsections detail the results.

5.3.1   Kinds of Annotations/Searches

In the tutorial, participants were not instructed on how code annotations should be worded. They were, in fact, informed that part of the purpose of the study was to see what annotations they used. However, they were told that one way of viewing the annotations was as if they were stage directions to the sprites. It was also suggested that when annotating existing code, they consider what other people might be thinking if they were looking for that code.

Table 5-1 shows some of the annotations that participants gave to existing code in the PacMan project; Table 5-2 shows some purpose queries that they gave in the mimicry task. Many of the annotations and queries were as expected, but some were different, e.g., "stay on path." In general, the breadth of vocabulary and ways of thinking about code purpose was surprising, which underscores the importance of incorporating a broad base of background knowledge: when presented with a word that has never before been seen in the annotation corpus, ProcedureSpace can take an educated guess about what it might mean based on commonsense background knowledge.

Some people used when-do clauses, e.g., ("when key press"), which a future version of ProcedureSpace should be able to understand. Sometimes because of these when-do clauses, and sometimes perhaps out of a desire to give a complete
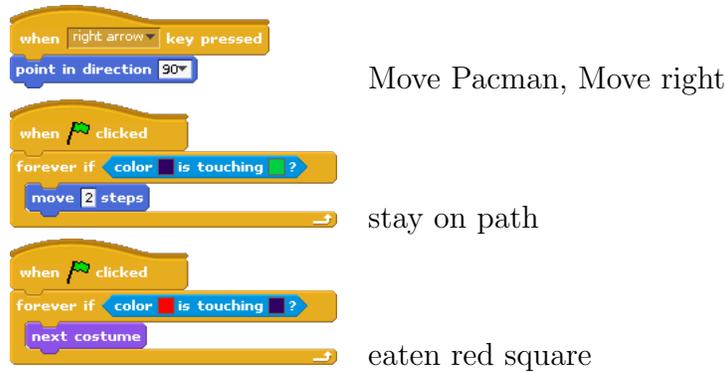
Move Pacman, Move right

stay on path

eaten red square

Table 5-1: Selected annotations for existing code in the PacMan project

Sprite	Purpose Queries
Player	move Player with arrow keys, Gravity
Ball	Bounce ball around room, bouncing, Hit edge and spins, Random fade
Shooter	track
Bat	Follow player
Platform	Bounce platform left and right
Projectile	Projectile motion, follow

Table 5-2: Selected purpose queries from mimicry task (without seeing code)

description, several people gave very long annotations, which were not as useful in the ProcedureSpace analysis as more concise descriptions. Finally, some included the name of the sprite in their annotation. In some cases, the name would also be a word that ProcedureSpace knew, which caused the search results to contain irrelevant results that merely contained that word. To mitigate this problem, some common sprite names, like "Player," were ignored for the purpose of searching.

5.3.2   Reuse Interactions with Zones

All participants used Zones extensively in the study. Some participants spent a lot of time examining the search results to find which was the most appropriate; one asked herself: "Which one requires less tweaking on my part?" One participant explored several different searches, remembering particular results and coming back to them. Some used Zones searches to find pieces of code that, while not exactly what they wanted, had pieces that were useful. When people found code, they often reused it

exactly as they found it, without even changing parameter values, though doing so would have been easy.

People expected to be able to find their own code or that of a community when they did a search. This would have been particularly helpful in cases where they wanted to do something similar to what they had done before.

In most cases, participants left searches around as annotations, even when they modified the code or wrote completely new code. Once, a participant ended up leaving a search that had found reasonable code but was decidedly not how she would have annotated it. But in general, participants seemed to understand and embrace the search-as-annotation interaction.

Though participants reused some code exactly, much more frequently the code fragments would guide their thinking or point out Scratch functionality that they could use. One participant saw a *glide* (timed movement) command in a search result, and exclaimed: "Oh, it could be gliding... I forgot [about] the glide function." Other participants found commands like "if on edge, bounce" or "go to *point*." At least one participant even started noticing larger patterns in programming structures. After seeing just a few examples of code in a project as well as search results, she started skipping quickly past unhelpful search results. When asked how she knew so immediately that they weren't useful, she said that she recognized some similar patterns to code that worked before. (The colors and shapes of blocks make some patterns more noticeable in Scratch than they might be in a textual language.) This episode suggests that Zones/ProcedureSpace might help people reuse code patterns as well as exact fragments, even without having explicit support for patterns.

5.3.3 Learning Interactions with Zones

I was surprised by the number of different ways that people learned from their interactions with Zones. Frequently, participants reported learning something from seeing another person's code, even if the code didn't directly accomplish their goal or they didn't understand all of it. In some cases, even though I had been working with Scratch and testing the system for a long time, some of the code that a Zone showed

taught *me* something also; e.g., that Scratch has a library function for querying if a sprite is touching the edge of the screen.

In the annotation task, one participant, a self-proclaimed Scratch novice, saw the "when *left arrow* pressed," etc. blocks and remarked, "I don't need to annotate that" because he thought their purpose was obvious. He annotated them anyway as "move PacMan"—but then clicked the Search button. As he paged through the ProcedureSpace search results, he was surprised to find code that looked very similar to a *different* script in the same sprite. This inspired him to take a fresh look at that other script. He soon realized how his understanding was incorrect, transferred the "move PacMan" annotation to the other script, and made a new annotation for the key-handling scripts. This vignette illustrates that seeing how other people link annotations with code helps programmers understand unfamiliar code—advantage of an interface that combines searching and annotating.

Another participant, skilled in programming languages other than Scratch, contrasted her experience with Zones with that of finding examples for a JavaScript library. In that previous experience, she had found certain sites that had interesting behavior and just copied their code without really understanding it. But she remarked that the Zones interface forced her to think from a higher-level perspective about what she actually wanted her program to do.

5.4   Summary

The user study demonstrated the basic usability and utility of the Zones/ProcedureSpace system. The annotation and mimicry tasks showed that participants were readily able to understand the system and use it as intended. Code they found using the Zones interface helped them implement a variety of behaviors. During the study, participants talked actively about what they were learning—not mainly about the system, but about programming in Scratch. And after the study, most participants took extra time to talk about how interesting the system and the underlying idea was to them.

Chapter 6

Related Work

While this work touches on broader ideas in goal-oriented human-computer interaction, the present implementation is within the realm of software engineering. Thus, this chapter begins by situating the Zones/ProcedureSpace implementation within the context of code search and reuse tools, then expands to discuss related work in goal-oriented interfaces.

6.1 Code Search and Reuse

Programmers have many options for finding code to reuse. They could base their entire work on an existing program; in the Scratch community, this "remixing" is the most popular form of reuse. They could look in a forum such as StackOverflow[31] or a snippet library like DjangoSnippets[6]. Or they could find a class or function in a library API—the preferred software engineering methodology. The term "code search" can refer to a system that retrieves code from any of these kinds of resources.

Code search systems can be distinguished by how programmers can query them. Approaches have included formal specifications[15], type systems[2], design patterns, keywords[18], and test cases[27]. Techniques for refining queries have included ontologies[35] and collaborative tagging[33]. [27] includes a good survey of code search techniques. However, these code search systems have limited ability to reason about purposes that can be accomplished in a variety of ways, and their understanding

of natural language is very limited at best. ProcedureSpace uses annotations to reason about purposes and leverages both general and domain-specific natural language background knowledge.

A task switch away from development, even to a very accurate search engine, introduces a substantial barrier to use. So Ye and Fischer[36] introduced the paradigm of reuse *within* development, linking code search into the IDE based on both keywords (from comments) and structure (from function signatures). They later surveyed facilities in the programming enviornment that can facilitate code reuse[37]. Many systems now integrate into an IDE; a state-of-the-art example is Blueprint[3].

Search-oriented systems like CodeBroker and Blueprint only directly benefit consumers of reusable software. Users of other integrated code search systems still have to publish their completed code, perhaps on a snippet library, blog, or code hosting platform. Zones "completes the cycle" by making it natural to share adapted or newly-written code.[1] We believe that integrating annotation will lower barriers to sharing and capture much more knowledge. Zones also introduces the reverse interaction—code to annotation.

6.2 Goal-Oriented Interfaces: Executing Ambiguous Instructions

This work brings work on search-based reuse together with a body of research that seeks to generate executable code given a potentially ambiguous specification of its operation in a language that is more natural for humans. This research goes back to the Programmer's Apprentice project. Its KBEmacs[34] took a high-level description of a program and generated code by combining "clichés" of procedural knowledge. It demonstrated that human-computer interaction in a programming scenario should be able to happen at multiple different levels of specificity. But its understanding of

---

[1]In an environment with heterogeneous licensing conditions, both the search and sharing components would need to be aware of license compatibility. And an integrated tool should always help the programmer credit the sources of any code they used.

language was limited to hand-coded knowledge, and its procedural knowledge limited to a very small library of clichés. ProcedureSpace, in contrast, learns about both natural language and code simultaneously.

Little et al.'s keyword programming[18] matches keywords in the input to the commands and types in a function library. Roadie[17], a goal-oriented interface for consumer electronics, goes further by using commonsense goal knowledge and a partial-order planner to understand natural language goals that do not directly correspond to procedures in its library. For ProcedureSpace, the library is a large corpus of mostly unannotated code, and the Zones interface allows the system to simultaneously search, annotate, and add to that corpus.

Metafor[19] and its successor MOOIDE[1] use sentence structure and mixed-initiative discourse to understand compound descriptions. MOOIDE further showed that general background world knowledge helps to understand natural language input. ProcedureSpace opens the possibility for these natural-language programming systems to *scale* by learning both statically from a corpus of code and dynamically through the Zones user interface.

Chapter 7

Conclusion

7.1   Contributions

This thesis makes the following main contributions:

- **Zones**, an integrated interface for connecting natural language with Scratch code fragments to make comments that help programmers find and share code, and

- **ProcedureSpace**, an analysis and reasoning method that reasons jointly over static code analysis, Zones annotations, and background knowledge to find relationships between code and the words people use to describe what it does.

The system demonstrates that reasoning jointly over natural language and programming language helps programmers reuse code. ProcedureSpace demonstrates how the Blending technique can reason jointly over very different kinds of data to find code with a requested purpose or purpose from code. The user study showed that people readily understood the Zones interface and were successfully able to use it to find code that both fulfilled their immediate programming goals and also helped them learn about programming in Scratch. Moreover, in the process of using the system, they left behind annotations that will help future users.

This work also contributes a pedagogical explanation of the AnalogySpace technique and steps towards more mathematically rigorous coverage of the Blending technique.

7.2   Applications

ProcedureSpace contributes to artificial intelligence an example of a system that can learn and use examples of mapping goals to characteristics of programs that accomplish those goals. It also demonstrates cross-domain semantic search, where data in different representations can be used in the same way for search.

To software engineering, Zones and ProcedureSpace suggests that natural language annotations can improve the use of ad-hoc libraries, that is, code that has not been formalized into a library but is nonetheless reused. As that code is reused and changed, the Zones system can monitor which parts needed to be changed. Then instead of parameterizing the library *a priori*, the modification data could be summarized to make an empirical parameterization that reflects how the code is actually used.

Finally, integrated code searching systems like Zones extend the idea of open-source software into the microscale. Instead of just collaboratively developing complete applications and packaged libraries, integrated code reuse allows contributors to share and collaborate on much smaller units, like individual code fragments. Though such collaboration was possible on an ad-hoc basis before, a system that supports that behavior can help (in a small way) to prevent the all-too-common practice in open-source communities of starting new projects from scratch and later abandoning them.

7.3   Future Directions

I built and launched a second user study, intended to collect targeted data for training and evaluating the ProcedureSpace algorithm. But in three weeks since first publicizing it, no one has yet participated in the study. If I get enough participation after re-evaluating participation incentives and publicity, then a future publication will include both those results and a quantitative evaluation of the ProcedureSpace algorithm based on them. We are also planning another user study to help one participant using knowledge it gained in interaction with another. Finally, while these user studies

incidentally collected data about how people think about programming, the variety of ways of thinking observed suggest that larger-scale studies on unconstrained natural language descriptions of programming problems could be very interesting. Some work of this type has been done by Pane et al.[23] and a few others, but experience with Zones suggests that the kinds of descriptions can vary significantly between contexts, so much room remains for further study.

I worked within the Scratch environment because of its concreteness and controlled environment. However, those qualities do not seem crucial to the success of these methods, so a next step is to try ProcedureSpace on other programming languages like Java or Python. The Processing[25] environment for Java seems like a good next step, since it has concrete visual primitives but permits the full range of representation and abstraction allowed by Java.

The flexibility of the ProcedureSpace representation offers many opportunities for improving code and language processing. For example, the Sourcerer system enables programmers to search based on the presence of a set of manually-defined "micro-patterns"[8] in the resulting code. We could learn micro-patterns of various sizes from the code+annotation corpus by identifying large clusters in ProcedureSpace and hypothesizing additional features that would summarize or separate items within that cluster. This process could be done without supervision by generating and checking possible generalizations of the larger syntactic structures within that cluster. And since the reasoning is done within ProcedureSpace, distinctions in the language used to describe code with those patterns will also inform the micro-pattern identification process, making the identified patterns more meaningful and practical.

Many of the features that others have used successfully for code search, such as type information, can be expressed as "structural features" within the ProcedureSpace framework, and could be incorporated in future work.

The current Zones interface permits search and annotation only at the level of complete scripts, which are somewhat analogous to functions in many other programming languages. Likewise, ProcedureSpace reasons only at the script level. However, some users desired to be able to interact with the system at different levels. To enable

101

that would require incorporating an understanding of *context*, especially including relationships between goals and subgoals. Such understanding would also help organize implementations into approaches to accomplishing a goal, and filter approaches by which would work in contexts like the current one. (This problem was actually my original goal for this thesis project.)

7.4   General Notes

Programming languages generally only represent a program in one way. If you're only working with one representation, any detail in that representation might be significant. But if you know about your program and the goals it accomplishes in more than one way, you can know what's flexible and what's not, and try different approaches or even different programming languages fluidly. ProcedureSpace views programs in a few different ways: the code itself, characteristics about that code, words and phrases that people use to describe what the code does or is for, and all of the interrelationships among those elements. With the incorporation of commonsense background knowledge, it could be said that ProcedureSpace knows about programs in some ways that are not within programming at all.

Whenever you're trying to solve a problem, it's helpful to see what others in similar situations did. People describe their problems differently, so background knowledge is necessary to find related descriptions. Similar procedures may be useful for solving different problems, or the situation may impose particular constraints on the procedure, so procedure analysis is also necessary. Techniques like ProcedureSpace that reason across both natural language and procedural language artifacts will leverage the distributed knowledge of communities of practice to empower users to perform tasks that they previously didn't even know how to exactly describe. Together, Zones and ProcedureSpace provide a new way for programmers to leverage the wisdom of those who have gone before them, and in the process contribute their own experience.

Bibliography

[1] Moinuddin Ahmad. MOOIDE: natural language interface for programming MOO environments. Master's thesis, Massachusetts Institute of Technology, 2008.

[2] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An internet-scale software repository. In *SUITE '09: Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 1–4, Washington, DC, USA, 2009. IEEE Computer Society.

[3] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-centric programming: Integrating web search into the development environment. Technical report, CSTR-2009-01, 2009.

[4] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 1589–1598, New York, NY, USA, 2009. ACM.

[5] Divisi. `http://divisi.media.mit.edu/`.

[6] Django snippets. `http://www.djangosnippets.org/`.

[7] Christiane Fellbaum. *WordNet: An Electronic Lexical Database*. MIT Press, Cambridge, MA, 1998.

[8] Joseph (Yossi) Gil and Itay Maman. Micro patterns in java code. *SIGPLAN Not.*, 40(10):97–116, 2005.

[9] Max Goldman and Robert C. Miller. Codetrail: Connecting source code and web resources. *Visual Languages - Human Centric Computing*, 0:65–72, 2008.

[10] Google Code Search. `http://www.google.com/codesearch`.

[11] Paul Grice. Logic and conversation. In *Speech Acts*. Academic Press, 1975.

[12] Catherine Havasi. *Discovering Semantic Relations Using Singular Value Decomposition Based Techniques*. PhD thesis, Brandeis University, June 2009.

[13] Catherine Havasi, Robert Speer, and Jason Alonso. ConceptNet 3: a flexible, multilingual semantic network for common sense knowledge. In *Recent Advances in Natural Language Processing*, Borovets, Bulgaria, September 2007.

[14] Catherine Havasi, Robert Speer, James Pustejovsky, and Henry Lieberman. Digital Intuition: Applying common sense using dimensionality reduction. *IEEE Intelligent Systems*, July 2009.

[15] Jun-Jang Jeng and Betty H. C. Cheng. Specification matching for software reuse: a foundation. *SIGSOFT Softw. Eng. Notes*, 20(SI):97–105, 1995.

[16] M. J. Kaelbling. Programming languages should not have comment statements. *SIGPLAN Not.*, 23(10):59–60, 1988.

[17] Henry Lieberman and José Espinosa. A goal-oriented interface to consumer electronics using planning and commonsense reasoning. *Know.-Based Syst.*, 20(6):592–606, 2007.

[18] Greg Little and Robert C. Miller. Keyword programming in Java. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 84–93, New York, NY, USA, 2007. ACM.

[19] 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.

[20] Hugo Liu and Push Singh. ConceptNet: A practical commonsense reasoning toolkit. *BT Technology Journal*, 22(4):211–226, October 2004.

[21] Audris Mockus. Large-scale code reuse in open source software. In *FLOSS '07: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, page 7, Washington, DC, USA, 2007. IEEE Computer Society.

[22] Andrés Monroy-Hernández and Mitchel Resnick. Empowering kids to create and share programmable media. *interactions*, 15(2):50–53, 2008.

[23] John F. Pane, Brad A. Myers, and Chotirat Ann Ratanamahatana. Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. Hum.-Comput. Stud.*, 54:237–264, February 2001.

[24] Christos H. Papadimitriou, Prabhakar Raghavan, Hisao Tamaki, and Santosh Vempala. Latent semantic indexing: a probabilistic analysis. *J. Comput. Syst. Sci.*, 61(2):217–235, 2000.

[25] Processing. `http://www.processing.org/`.

[26] James Pustejovsky, Catherine Havasi, Roser Saurí, Patrick Hanks, and Anna Rumshisky. Towards a generative lexical resource: The Brandeis Semantic Ontology. *Proceedings of the Fifth Language Resource and Evaluation Conference*, 2006.

[27] Steven P. Reiss. Semantics-based code search. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society.

[28] Mitchel Resnick, Yasmin Kafai, and John Maeda. A networked, media-rich programming environment to enhance technological fluency at after-school centers in economically-disadvantaged communities. Proposal to National Science Foundation, 2003.

[29] Scratch website. `http://scratch.mit.edu/`.

[30] Robert Speer, Catherine Havasi, and Henry Lieberman. AnalogySpace: Reducing the dimensionality of common sense knowledge. *Proceedings of AAAI 2008*, October 2008.

[31] StackOverflow. `http://stackoverflow.com/`.

[32] Antal van den Bosch and Walter Daelemans. Memory-based morphological analysis. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics)*, pages 285–292, 1999.

[33] Taciana A. Vanderlei, Frederico A. Dur ao, Alexandre C. Martins, Vinicius C. Garcia, Eduardo S. Almeida, and Silvio R. de L. Meira. A cooperative classification mechanism for search and retrieval software components. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 866–871, New York, NY, USA, 2007. ACM.

[34] R.C. Waters. The Programmer's Apprentice: A session with KBEmacs. *IEEE Transactions on Software Engineering*, 11(11):1296–1320, 1985.

[35] Haining Yao, Letha H. Etzkorn, and Shamsnaz Virani. Automated classification and retrieval of reusable software components. *J. Am. Soc. Inf. Sci. Technol.*, 59(4):613–627, 2008.

[36] Yunwen Ye. *Supporting Component-Based Software Development with Active Component Repository Systems*. PhD thesis, University of Colorado, 2001.

[37] Yunwen Ye and Gerhard Fischer. Reuse-conducive development environments. *Automated Software Engg.*, 12(2):199–235, 2005.