Entologic

Daniel Mulcahy & Rory Hughes
Gonzaga College S.J.

January 2014

Abstract

The ability to program computers is an extremely important and useful skill yet learners often have a large amount of difficulty understanding what a given piece of code does. This obstacle is one we both endured and see in fellow students in school. While we just had to persevere, many people who want to start often end up quitting as code can be too overwhelming for them.

We have solved this problem by designing a system which takes learners code, parses it, analyses it, and presents to them a description of their program in their native tongue. It breaks language barriers and, overall, makes learning to code easier.

0.1 Comments

0.2 Supplementary info

TODO

Pdf version of this report
    `http://tiny.cc/entologicreport`

Project display
    `http://tiny.cc/entologicposter`

Contents

Chapter 1

Introduction

1.1 Initial thoughts

A few months ago, my friend Luke Gardiner and I (Rory Hughes) were discussing linguistics and the relationships between spoken and programming languages. Spoken ones have their own set of vocabulary which may originate from predecessor while key words in programming languages (if, else, print etc.) are most often based off of another spoken one. You reading this may or may not be familiar with code. If you are, have you ever thought about why the thirty plus most used programming languages are all based on English words?[1]

We realised that the main reason for this was that most of the computer revolution in the early days was happening in the UK and USA when the first programming languages started to appear. Also with very limited character encoding (i.e. ASCII), it meant that non Latin alphabet systems were hard to come across. What is usually the case when non-English speaking students study computer science is that they will have a relatively good foundation in English so the basic keywords will come quite easy to them. They then can also use their own language for variable names in their programs. This is fine for the computer science students who have some English and have all the support of their college to explain things they have trouble understanding.

However, today we are in an age when computer science is starting to make its way into primary schools. Most children today have some kind of computer of their own that runs games and other applications and many also want to learn to build their own. Initiatives like CoderDojo have set up to help young children write basic software are growing at immense rates. In the map below (figure 1.1, probably outdated as you read this) you can see how it is spreading outside of English speaking countries.[2]

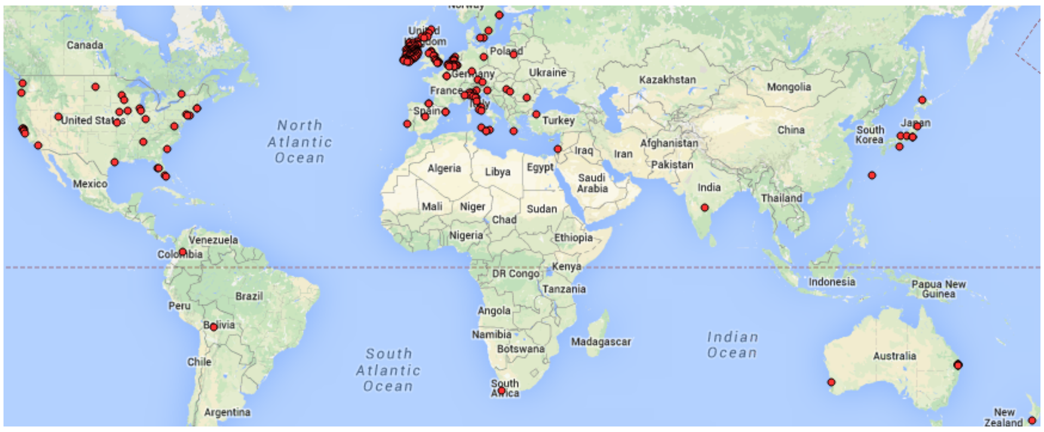

Figure 1.1: CoderDojos around the globe

This goes to show what kind of interest there is but many still don't have access to proper computer education and end up trying to teach themselves. Starting off learning to code on your own is hard but when the code isn't in your own language (not to mention the documentation for it), it's another story.

We thought about it and Luke suggested internationalising existing programming languages.[1] We imagined a kind of interface which would sit in front of the compiler or interpreter and look up each custom keyword, replacing it with the English version before going any further. That would definitely make things fairer and still allow learners to study the same technologies. We thought that this would be the project we would pursue until later that evening while I was looking at the Wikipedia page on non-English programming languages.[3]

The section 'Modifiable parser syntax' described multiple existing attempts which did exactly what we came up with, except each one only worked with a single programming language. We then came to realise why these (for the most part) have not become popular in the programming world. If you were to go to college in China and learn 'ChinesePython' and later go and seek employment, the company will more likely be using standard Python syntax. As none of the employees wouldn't understand your special dialect, they probably wouldn't be interested in you as collaboration and understanding of each others code is key in software development. We gave up on that particular solution but after seeing stories in the media about the large numbers of vacant IT jobs[4], we persisted with the objective of lowering the

[1]One project idea we had well before this time was to create a programming language that had keywords in Irish

barrier of entry for learners.

1.2 New Ideas

In school, my colleague Daniel and I run a computer club where we teach students from 1st to 6th year how to code. That week we set out to study how they learnt from us but more importantly, what part of the process did they find hardest. The most prominent problem we discovered was one that we both experienced in full teaching ourselves. Understanding what a given piece of code did and how. What does each function do? What's the difference between braces and square brackets? What is this expression evaluating? While most of them are shy too ask, I often question them after explaining the flow of the program on the board. They can surely tell me that when I click the button 'Hello World!' will appear yet, for example, they don't have an understanding of what passing arguments to a function means.

This type of understanding is similar to buying a phrase book when going to France and learning off “Comment allez-vous?” as “How are you?” but later needing to say “How is she?” and having no knowledge of what adjustments are to be made to the original phrase.

Learning programming by example is very effective when done correctly but often people just learn off (or worse, copy and paste) things from the Internet having no comprehension of what is going on in each line. This is detrimental to writing efficient, secure and understandable software. A few peers from school including some in the computer club have in the past tried learning languages like Java on their own from an on-line tutorial. It's much easier for newcomers to follow as opposed to reading through every class, method and piece of syntax in the documentation. However they always end up giving up due to complexity and in general finding it hard to understand.

```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Figure 1.2: What is actually going on here?

We decided to solve this problem with EntoLogic.

1.3 Explanation

Fast-forward a few days of discussion, and we came up with the basics of a solution that considers all the problems mentioned in the past two sections. We designed and built a prototype of a full stack system that allows users to provide code and get it explained to them. Furthermore, we architected it in a way that allows it to be extended to translate from any programming language to any spoken language using the help of the on-line computer science community.

Chapter 2

Planning

The majority of the time we spent working on this project was not sitting down and pumping out lines of code building a mediocre website. We spend it designing the system inside out so when we knew it was time to start some practical work, we wouldn't have to make it up as we went along.

2.1 Goals

We had a few prerequisites for system to accomplish (Other than aiding programming education in general).

- It could accommodate any programming language once there is access to a parser for it.
- Explanations for each part of the programs could be adapted for multiple spoken languages other than English.
- Getting your code explained would be as easy as pasting it in to a web app and having it display in an easy to read fashion.
- Its intelligence would be driven by the community. People could correct and vote on things which would make for better explanation.

Making each of these things possible was a big challenge and we knew it would use many heuristics.[1]

[1]Experience-based techniques for problem solving that give a solution which is not guaranteed to be optimal.

2.2 Fulfilment

Understanding code Figuring out how to make computer programs understand what was going on in other computer programs proved to be a hard but interesting task. Whenever code needs to be run on a computer, it firstly needs to be converted into a format (machine code/binary) that the processor can understand. The program that takes care of this is called a compiler. As you may have guessed, there are a lot of other steps between getting from the plain text code to an executable binary.

The first thing the code reaches in a compiler is the lexer. It breaks down all the words into tokens (keywords, identifiers and symbols) which can be manipulated in a simpler manner. After some more preprocessing, it hits the parser. The parser takes the tokens and converts them into something which we will be talking a lot about called an 'Abstract Syntax Tree' (or AST for short). This tree is basically a representation of the code's syntax in a traversable tree format. It represents the structure of the program in nodes where each node can be made up of more sub-nodes and so on. It is at this point in the compilation process where we found we could extract meaning from the code.

Every programming language As mentioned in our goals, we wanted to be able to support any programming language. The major obvious problem with this was that every programming language AST has its own format. They each have language specific parts and are not compatible with each other. This would make creating explanations very complicated as the 'explainer' would have to handle many different kinds of ASTs. After some thought, we came up with a solution to this which was to create our own tree format called a Universal Abstract Syntax Tree. Each language supported would have a UAST generator which would parse the code into its native AST format and convert it into the unifying one. The translator would then only have to support one kind of input. We looked around for existing work in this domain and noticed that the LLVM project had some versions they used in their open source compiler infrastructure.[5] However we decided in the end that we would design our own for simplicity and customisation.

Phrase System We spend a large amount of time working out how the system would give a natural language output. Doing research about it, we found a lot of studies and findings done by Google in Natural Language Processing.[6] While we did get some inspiration from them, many of the research papers talked about machine learning methods and other things that were generally out of the scope of our project. We decided to devise our own simple 'phrase' system which will be described on page 15.

Every spoken language We initially agreed all output could be in English and then sent through an on-line machine translation service but we then realised that it would be limiting to non-English speaking users. Thus, we adjusted the phrase system to be more flexible in handling multiple different natural language constructs. Also, if there were no human translated phrases available for the spoken language requested, it could still fall back to using a machine translator.

The Translator We decided that there would be one core application which would handle all of the translation from AST to spoken language output. This program would take in the inputted code, programming language to translate from and spoken language to translate to. It would first run the UAST generator for the given programming language as a separate process to parse the inputted code into an UAST and give it back to the translator program. Then it would translate each node in the UAST using the phrases for those nodes and give back an output.

The Web App and Community Dependence Making the system easily accessible was very important from the very start. Using the web was the obvious solution as it is platform independent and so programming novices wouldn't have to use the command line. We also knew we had to open it up for the input of passionate programmers to add more functionality like better translations for particular parts of programs and the ability to create UAST generator for languages they wanted to see compatible. (Page 19)

2.3 Choosing our tools

2.3.1 Technology

The Translator I (Daniel) decided to use the Haskell programming language [7] for the translator for several reasons. I had been learning Haskell for several months prior to starting the project, and while I had not written any large piece of software in it, I liked it and thought it would be a good language for this purpose. Haskell has many features that would make it appropriate for writing the translator in, chiefly its functional purity, which minimises unnecessary side-effects and helps break up the program into smaller pieces, and its strong static type system which greatly helps in managing bugs my catching many potential problems at compile time, as well as encoding properties such as possible failure and performing input/output in the types of functions.

The Web App I (Rory) chose to build the web app which would display all the information for the learners on the web. I chose to use to write the back end in Node.js as I had been using it for a while before this project and it is proven to be quite efficient when handling large amounts of data simultaneously. For the front end, I used the Angular.js framework.

Database Choosing a data store was something we both had to decide on together as it would be the link between the two parts. As we knew a lot of our data would be stored in trees, a NoSQL solution looked like the answer. We ended up using MongoDB as its popularity is useful for when we need any help solving problems.

2.3.2 Collaboration

Revision Control As we would be working with lots of code and it would have to be easily accessible for each other, using a revision control system was essential. We both had most experience with git[8]. Using git allowed us to each work on separate parts of a certain code base and merge our changes together. We used BitBucket.org for hosting our repositories.

Organisation For working together on documents and ideas, we used Google Drive and Trello respectively. They allowed us to collaborate in real time from wherever we were.

Communication We used Skype for all communication. Its video and screen sharing capabilities proved useful when wanting to demo ideas and features to each other.

Chapter 3

Explanation

This chapter describes the design of the explanation system in depth.

3.1 Parsing

We chose to use Ruby as our language to test the explanation of. Searching the web for parsers revealed Ripper[9], the Ruby standard library parser, to be the most used. It gives its output in lisp-like s-expressions.

```
[:program,
 [[:def,
   [:@ident, "hello", [1, 4]],
   [:params, nil, nil, nil, nil, nil, nil, nil],
   [:bodystmt,
    [[:command,
      [:@ident, "puts", [2, 1]],
      [:args_add_block,
       [[:string_literal,
         [:string_content,
          [:@tstring_content, "hello", [2, 7]]]]],
       false]]],
   nil,
   nil,
   nil]]]]
```

Figure 3.1: Ripper output

We then had to create a converter which would take this format and convert into our UAST format (described in the next section) preserving as much information as possible. There were often times when we found that ripper gave more information than we needed and other times when our own UAST specification demanded more than Ripper would give. One example of the latter is that we set up the display of the output explanation to highlight the start and end of the part of the program it was talking about. However, the parser was only able to give the starting characters of identifiers so we had to do away with some usability. For our prototype, we got the UAST generator to convert some basic constructs fine.

3.2 UAST

Creating the UAST specification was a significant challenge. In order to be fully effective, it had to be able to represent every syntactical feature of every programming language that the system could potentially support. As this was a momentous challenge, we decided to write a basic UAST specification to which we would add nodes as languages required them. The UAST is transmitted between the separate language parsers/UAST generators in a JSON representation, which is converted by the translator to its internal representation as a number of Haskell Algebraic Data Types (ADTs) representing each AST node. Except for a small number of basic nodes (such as integer literals, i.e. integers that appear directly in the source file), each AST node stores several attributes. These attributes include things like:

- The function name in a function declaration (e.g. 'hello' in the example)
- The operator in a binary operator expression (an expression with an operator that works on 2 arguments, e.g. x + 3 in the example)
- The arguments in a function/method call ("a function has been called" in the example)

```
def hello(x)
  puts "Called a function"
  x + 3
end
```

Figure 3.2: Example Ruby code

The code in this example would be converted into the JSON encoding of the UAST like this:

```
{
  "node": "FuncDecl",
  "name": "hello",
  "arguments": ["x"],
  "body": [
    {
      "node": "FunctionCall",
      "name": "puts",
      "arguments": [
        {
          "node": "StringLit",
          "value": "Called a function"
        }
      ],
    },
    {
      "node": "BinaryExpr",
      "operator": "add",
      "left": {
        "node": "VarAccess",
        "var": "x"
      },
      "right": {
        "node": "IntLit",
        "value": "3"
      }
    }
  ]
}
```

Figure 3.3: UAST representation of example Ruby code

3.3 Phrase System

The phrase system we designed allows us to provide support for many programming and spoken languages in a very extensible and user-friendly manner. A phrase is a translation for a specific AST node in a specific programming language for a specific spoken language. In order to allow differing translations based on properties of a specific AST node, phrases consist of several clauses. Some clauses are default clauses, and will always appear in the translation, but some clauses are conditional clauses and will only be added to the translation if certain conditions based on attributes are true. For example, a clause in a function declaration that depends on the presence of the attribute "arguments" will only be shown in the translation if the function is declared to accept arguments. There are two types of conditions for clauses: presence and comparison. Presence conditions test the presence of an attribute (or, if the attribute is a list, whether the list is empty), and comparison conditions compare an attribute (usually the length of a list) against a given number. The contents of the clauses themselves is an array of strings. These can be constant strings that will appear literally in the translation, or variables that will be replaced by the translator with their values. These variables are indicated by starting with a double dollar sign (e.g. "$$arguments"), and usually directly represent attributes of AST nodes (e.g. function arguments and body, the operator in a binary expression).

Phrase storage The phrases are stored in the database, with their specific programming languages, spoken languages and AST node types. As requiring different phrases for each AST node for every programming language would create a large amount of work fraught with unnecessary duplication, phrases can be made with the programming language name "default", in which case they will be used as the default phrase if no phrase for that node and spoken language is available in a particular programming language.

User contribution In order to vastly broaden the scope of programming and spoken languages the system supports, phrases can be contributed by users through a web interface. The exception to this is default programming language phrases: in order to encourage community improvement of phrases for specific programming languages, phrases for the default programming language can only be added by us, the administrators. As people can contribute phrases for any node, programming language and spoken language, we needed a way to sort phrases in a way that would hopefully ensure quality. The most obvious solution that presented itself to us was community voting: Users can vote on phrases and the most upvoted one for a given AST node, programming language and spoken language is what is used. To deal with maliciously submitted spam and nonsensical phrases, there is a report-to-administrators facility for phrases as well.

Phrase Example If the phrase in figure 3.4 is used to translate the function call to "puts" in figure 3.2, the translator will first check which of the conditional clauses it uses. It will always include the first clause as it has no condition. Because the function has arguments, it will include the second clauses whose condition is based on the presence of the "arguments" attribute However, it will not include the third clauses whose condition is based on the presence of the "arguments" attribute with "reverse" set to true, because the "reverse" property inverts the condition, meaning that clause will only be included if the function call has no arguments. Then the translator will go through each clause and replace the variables "$$name" and "$$arguments" with the name of and arguments to the function respectively. The result of this is the translation "call the function puts with the arguments 'Called a function'"

```
{
  "phraseName": "FunctionCall",
  "pLang": "default"
  "nLang": "en",
  "clauses": [
    {
      "condition": null,
      "words": [
        "call the function",
        "$$name"
      ]
    },
    {
      "condition": {
        "conditionType": "presence",
        "reverse": false,
        "nodeAttribute": "arguments"
      },
      "words": [
        "with the arguments",
        "$$arguments"
      ]
    },
    {
      "condition": {
        "conditionType": "presence",
        "reverse": true,
        "nodeAttribute": "arguments"
      },
      "words": [
        "with no arguments"
      ]
    }
  ]
}
```

Figure 3.4: An example of a phrase representing a function call

3.4 Other Spoken Languages

Our original plan for handling other spoken languages was simply to translate everything in English and send the results to a machine translator. However, we realised that there are many problems with this solution that would be disadvantageous and potentially deleterious for people of other languages.

- Machine translation is imperfect and the result of it would likely be inferior translations compared to the English version.
- Even if the translation is linguistically perfect, it would likely not produce the correct terminology for many programming constructs which would be confusing and misleading.
- Machine translation would be difficult or impossible to do while preserving all the formatting of the translation such as newlines and indentation, which is an important part of making the translations meaningful and understandable

As a result of this, we came up the phrase system as a better alternative. Though it would require much more work on the part of contributors, the result would be vastly superior to a machine translated version. However, in order to better serve users with relatively uncommon native languages, we realized that depending entirely on contributors who know enough of these languages to contribute translations would probably not result in complete translations. Therefore we are working on using machine translation as a fallback if a certain phrase is not available for a non-English language.

Chapter 4

The Web App

As we were creating a tool to help new programmers to learn, making it easy to use was essential. We knew the website had to have the following:

- An easy way to input code, similar to that of a desktop text editor.
- A presentable output which would show the connection between the inputted code and the explanation.
- The ability for users to be able to save their code and explanation for later viewing.

4.1 Backend

Building the backend API proved to be quite simple. All it needs to do is basic create, reading, updating and deleting of information.

Explanations One of the most important bits information that it has to work with is the explanations in the database. They are each a record of somebodies code wanting to be put into the translator The database record for an explanation includes:

- Plain text code input
- Languages (programming and spoken)
- Output explanation tree
- The user who created it and whether they have saved saved it or not

We have set it up so each of these pieces of data is saved once a user submitted code to be explained. Once the explanation has saved in the database, the translator then picks up on the job and process it, writing the output explanation into the record on completion.

Phrases The other crucial model the backend has to handle is phrases. We wanted users to be able to add phrases and vote on which ones they think are most appropriate for explaining a certain part of a program. The information stored with a phrase includes:

- The phrases clauses and conditions
- Languages (programming and spoken)
- What the phrase is describing
- Whether it's currently in use by the translator or not
- How many votes it has
- The user who created it

As mentioned in the previous chapter, this data is entered by users through the web app and the translator makes a database query for ones that are currently marked as 'in use'.

4.2 Frontend

Building the frontend with Angular.js allowed us to have a lot of the messy user interface work done and out of the way while still granting a lot of control. It provides the model view controller paradigm for development in the web browser and helped when working with the nested data structures. This came in especially useful when creating the output display with all of the phrases and clauses within each other.

The code editor For the code editor we used a open source project called CodeMirror[10] which is basically a mini, yet very customisable, text editor for the web browser. One of its most important features was the ability to set CSS classes on certain characters by their lines and columns. This was important when adding the code highlighting feature which showed what part of the code the explanation was talking about.

Explanation output As the output of the translator was in a tree format, we couldn't just display the phrases in a list format as this would also disrupt the nested nature of the program. We used Angular directives[1] for phrases and clauses which recursively sit inside of each other. Each of the phrase directives attach mouse hover handlers which call an external function passing the line and column of the code the node was originally on. This function talks to CodeMirror which adds highlights the line and column to the user.

The phrase editor The other section of the web app users may navigate to is the translation section. Here we built a phrase editor which allows users to enter alternate translations for particular phrases. They can insert static text, variables and add conditions to the clauses. Once they are happy with their attempt, they can submit it for review by other users voting and the admin approval to be put in use by the translator.

Information pages As we want to encourage the community's involvement in growing the technology, we included information on the site on how to help. They can learn about how to create phrases using phrase editor and can go to our GitHub[2] organisation to see the UAST specification and other advice on how to get started building a UAST generator.

[1]Directives are markers on a DOM element that tell Angular.js's HTML compiler to attach a specified behaviour

[2]Web-based hosting service for software development projects.

Chapter 5

Further Development

In the last few chapters we mainly spoke about the main feature set which we got working in the prototype. However, we have planned out many more features in detail which will got towards helping a learner get an even better understanding. We hope to continue this project and work on these features in the near future.

5.1 Machine Translation Fallback

One feature we are currently implementing is allowing the translator to fall-back to using a machine translator like Google or Bing Translate. The translator would use it when there is no phrase for a specific node available for the spoken language requested. (More on page 18)

5.2 Documentation

Many users will often also want to view the official documentation for a certain function they are getting explained. We are working on programs that search the documentations for the programming languages we support and store a reverse index of the keywords in the database. For example, when a user enters code along the lines of:

```
Math.sin(Math::PI)
```

The translator will recognise it and attach the keywords to the output tree. When a user then hovers over it, the web app will take the keywords and search the index for documents with ”Math” and ”sin”. It will then rank the results taking into consideration how many keywords were matched and how many people voted it as a useful article.

We also may allow contributors to add smaller static explanations and code examples using standard library function which can be curated in the same way phrases are.

5.3 Analysis

Another idea we came up with was to get the translator to count how many times a phrase in a certain category is used. This information could be used on the phrase editing page to show which ones are in need of translation the most.

5.4 Commenting

One feature that would be quite trivial to implement would be adding a forum/commenting section underneath thing which users can vote on like phrases. This would allow for people to point out the creators error and help correct mistakes.

5.5 UAST Generators

Alongside our Ruby one, we have started work on a UAST generator for Java code. We also contacted fellow programmers to look at the contribution info. They have responded to build Python and PHP generators which will widen the selection of programming languages we support the explanation of.

5.6 Contextualised Function Calls

One feature that would be extremely helpful but difficult to implement is to, when a function that was previously defined in the inputted code is called, display the definition of that function with the function arguments in the definition replaced by the actual arguments the function was called with. For example, if the `addThree` function in figure 5.1 is defined and later in the code the `addThree` is called with the argument 7, a popup box would appear on hovering with the contents of figure 5.2

5.6.1 Execution

When contextualised function calls are implemented, a possible extra feature we could add is function execution. Evaluating the contents of a function call

```
def addThree(x)
  x + 3
end
```

Figure 5.1: Example function `addThree`

```
def addThree(7)
  7 + 3
end
```

Figure 5.2: `addThree` function contextualized

and displaying the result and output would be a very advanced task, involving executing every function called within the function definition and keeping track of local and global variables, while ensuring that the the execution is limited to a subset of language functionality to prevent malicious code compromising the integrity of the system, but if it were implemented it would be even more useful in aiding people learning programming as not only would they be able to have their code explained step by step, but they could also see the results of and program state at each piece of the code.

Chapter 6

Conclusion

6.1 Accomplishments

Design We felt at the beginning that the most important part of our project was to design a full-stack system that could help learners understand code and we did just that. Our design is the first stepping stone to a much greater infrastructure that will help in computer science education.

Prototype The prototype we built is extremely important in that it demonstrates the implementation of our ideas in reality. We built the whole system from the ground up and it definitely shows our design's potential to go on further.

Experience Both of us benefited ourselves quite a lot working on this project. There were many technologies that we decided to use in the beginning which we didn't have as much experience using as we would have liked. Building the prototype set us goals and we definitely learnt more than we expected to.

6.2 Uses

We can see our project being used in the future in many places. Some of them include:

- Self learners following tutorials at home.
- Students in third level studying computer science.

- In enterprise to allow non-developers understand what pieces of company software does.
- Other websites could access the technology through an API to display in line explanation beside their code.

6.3 Feedback

After getting everything to work, we knew that we could do more to show the interest of others in the technology. We went and demonstrated it to some small groups of people in the sectors mentioned above and recieved comments back from them.

> Breaking down programming language into easy to understand human language is absolutely critical for learning and truly understanding code. Sometimes the biggest asset of CoderDojo to a young person is just having a mentor there to explain things to them. A project like this could greatly impact code learning in CoderDojo and help so many people around the world who strive to learn to code but have no-one to explain it to them.

- *James Whelton - Founder of CoderDojo*

> Deeply intuitive, and comprehensible to those without even basic programming knowledge. The value of this utility for those interested in teaching themselves how to code is obvious.

- *Conor Kelleher - Non-programmer*

> While learning to program, it's too easy to get caught up in syntax and notation. EntoLogic is a smart step forward in helping syntax fade into the background; enabling newcomers to focus on a program's algorithms and data structures, as they should. It's fascinating to think about a future where this technology might be built directly into development environments.

- *Ben McRedmond - Developer at Intercom*

> Having this technology available to me when I started programming would have helped me in that I would've been able to more easily understand what a program does, and more importantly which parts of the program do what, in a simple way.

- *James Eggers - Computer Science Student*

Chapter 7

Appendix

7.1 References

Here is some of the reference material the helped us to learn the technologies used in our project.

- Learn You a Haskell For Great Good!, Miran Lipovača, 2011
- Real World Haskell, Bryan O'Sullivan, Don Stewart, John Goerzen, 2008
- Programming Ruby 1.9, Dave Thomas, 2009
- JavaScript: The Good Parts, Douglas Crockford, 2008
- AngularJS, Brad Green, Shyam Seshadri, 2013

7.2 Thanks

We would like to thank the following people for their support and help throughout our project:

- Declan and Catherine Hughes, Parents
- Alan Mulcahy and Una Parker, Parents
- Mr. Kevin Whirdy, Headmaster
- Mr. Joe O'Briain, Science Teacher
- Ms. Siobhan McNamara, School Librarian

- Ben McRedmond and Patrick O'Doherty, Past Pupils
- Luke Gardiner, Fellow Pupil

Bibliography

[1] LangPop. Programming language popularity. http://langpop.com/.

[2] CoderDojo. Map of coderdojos around the world. http://zen.coderdojo.com/.

[3] Wikipedia. Non-english-based programming languages. https://en.wikipedia.org/wiki/Non-English-based_programming_languages.

[4] Silicon Republic. 700,000 unfilled ict jobs in europe, February 2013. http://www.siliconrepublic.com/careers/item/31404-fjf2013.

[5] LLVM. Project website. http://llvm.org/.

[6] Google. Natural language processing research. http://research.google.com/pubs/NaturalLanguageProcessing.html.

[7] Haskell. Language website/wiki. http://haskell.org/.

[8] Git source control management. http://git-scm.com/.

[9] Ripper (ruby standard library). http://www.ruby-doc.org/stdlib-2.0.0/libdoc/ripper/rdoc/Ripper.html.

[10] Codemirror browser text editor. http://codemirror.net/.