Newspeak: An English-Like Programming Language for Beginners

Tim Meredith

March 17, 2021

Abstract

Learning to program computers is difficult. A student who is just beginning to learn to code may not have any analogous experience to draw upon; concepts like functions and loops are often entirely novel. When we teach a beginning programmer to use a traditional programming language like Java, we asked them to learn not only these core concepts, but also a bevy of non-functional syntax rules. A new student may not even get to the point of understanding functions if they feel lost in a sea of unfamiliar brackets, braces, and semicolons.

This paper introduces the hypothesis that a student would learn the core concepts of programming more quickly if there were a way to set aside these syntax issues. In these pages, the programming language Newspeak is introduced to test this hypothesis. Newspeak is designed to look and feel as much like the English language as possible, so that students can use a syntax that they are already substantially comfortable with while they are learning the fundamentals of computer programming.

Contents

1 Introduction

There are a lot of complexities involved in learning computer programming. Not only are new students inundated by concepts of the logical functioning and capabilities of a computer, but they are also expected to learn new and occasionally confusing syntactical constructions to express these concepts.

In my three semesters as a teacher's assistant for beginner Java courses, I have noticed that both of these facets can trip students up, and in about equal measures. For every minute I have spent explaining the general function of a `while` loop, I have spent another minute explaining that every method call needs to end in parentheses, even if it takes no parameters.

The burden of managing both of these facets at the same time isn't insurmountable, but I posit that it is unnecessary. I have developed the following hypothesis: Students would learn the fundamental concepts of computer programming more quickly if they could deal initially with a programming language that had a familiar and intuitive syntax, and later transition to a traditional language. A proper study of the validity of this hypothesis (the process by which it would become a theory, to use scientific terms) would involve entire semesters worth of teaching as many students as possible using such a language, with further control classes teaching substantially similar curriculums with traditional languages. I won't pretend that this project has the resources for that type of study, but I would like to lay the groundwork for one.

The intermediate step that I introduce, one that could be used to test my hypothesis, is Newspeak. This language has a simple mission: to facilitate the core concepts of computer programming, while maintaining a syntax that will be as natural and intuitive to English speakers as possible. A student can begin their journey by using Newspeak to learn about variables, control flow, functions, and the like, and then they can learn about how these concepts translate into more concise general-purpose languages like Python or Java. Within this paper, the design and implementation of Newspeak is presented.

1.1 An Aside on Naming

In George Orwell's 1949 novel, *Nineteen Eighty-Four*, the government decides that the ambiguities and inefficiency of the English language are a drain on productivity, so they impose their own highly structured variant of English, which they call 'Newspeak'. It reduces ambiguity mainly by severely limiting vocabulary. Similarly, Newspeak the programming

language often looks like English, but it certainly does not permit an arbitrarily large vocabulary, nor does it permit any large amount of creative license. The price we pay for a language that a computer will easily understand is that it ceases to be suitable for the next Great American Novel. The language of Orwell's novel and this programming language have one key difference: if you refuse to code with Newspeak, you need not fear reprisals.

2 Background

Let us examine the concepts and past work that have influenced Newspeak.

2.1 Natural Language Programming

In his 1967 article "Foundations of the case for natural-language programming" [1], Mark Halpern outlines the debate between those who favour strict, mathematical programming (he calls them "calculists"), and those who are interested in natural-language programming. The article outlines two types of natural language programming: *passive* languages, which read like English but have a traditional rigid syntax; and *active* languages, which should accept arbitrary natural-language constructs in any way the programmer chooses to express them.

Halpern's main case in favour of natural languages for programming is that they would allow any well-informed computer user to program, and not just trained programmers. By his account, this requires an active language; he derides passive ones as exhibiting the "worst features of both [rigid calculist languages and natural languages] and the virtues of neither". They are too wordy for experienced programmers, and too rigid and specific to be picked up by casual users without investing time in learning the particulars. This may be true for general-purpose programming languages, but the target users of educational languages are neither experienced nor casual. His view also relies on a world where actively-natural languages exist, but all popular languages of the day are closer to the rigid calculist vision[1], so novice programmers should want to get used to writing in such a constrained fashion. As students, they also should not mind investing some portion of time into learning the particulars of a language; the point is that using natural-language constructs in intuitive ways should reduce this learning time, for the same reason that unconstrained active languages might reduce it to near-zero. In these ways the passively-natural language, dismissed by Halpern for the case of general programming, is in fact ideal as an educational stepping-stone on the path to the more mathematically rigid languages of his calculists.

For the most part, passively-natural programming languages are rare (and therefore so are natural-language programming languages in general, as active ones are even more rare), but they are not unheard of. One of the most thoughtfully-considered of these is the Inform language [2]. Unsurprisingly given Halpern's objections, Inform is not a general-purpose programming language; it is designed to allow writers to create text-based interactive-fiction games with source code that resembles natural prose wherever possible. A key insight from the design of Inform is that people will not always be most comfortable interacting with the

[1]This is not necessarily because Halpern's arguments would lose out given an even playing-field; the requisite technology for his active language vision is likely not yet mature enough to make them feasible.

computer by issuing it commands. In fact, it's quite normal in Inform for a programmer to interact by making a statement of fact about the game world, as can be seen the third line of this snippet:

```
Before taking the crate:
        if the player is wearing the hat:
                now the hat is in the crate;
                say "As you stoop down, your hat falls into the crate."
```

The line in question does still implicitly order the computer to do something; specifically, to alter its internal state to register the fact that the hat is in a new location. It is simply often more natural to express the new state than to command that the state be changed.

Upon reflection of the above facts, it was decided that the goals of this paper would be best accomplished by a passively-natural programming language, the pitfalls of such a concept being duly considered and deemed unimportant in context. Additionally, for a more natural feel, the language should permit its programmers to incorporate multiple modes of speech.

2.2 Educational Programming

The idea that programming languages used for education should be carefully selected is not novel. Kruglyk and Lvov conducted an analysis [3] in 2012 to determine the best existing programming language for use in education. They were trying to find a general-purpose language that was the most suitable to be taught to a beginner, and were not outlining parameters for a new intermediary language, so this leads to some differences of opinion, but in general this paper aims to comply with their criteria. The following is a selection of the requirements for an educational programming language that they outline, to be either accepted or refuted.

> The language and the programming system, connected with it, should represent all main programming concepts in their structure.[2]

This is fairly straightforward. Opinions may differ on which concepts are to be considered "main", but certainly once that decision is made, the point of an educational programming language is to cover all of them.

> The structure of educational language should be methodically conjugated with the structure of modern standard high-level programming languages, i.e. learning this language should make further transition to learning other languages (for instance, within professional training) easier or not complicated.[2]

Certainly, a primary goal of this paper is to outline a language that makes for an easy transition to popular modern languages.

The following requirements are specific to the programming environment and not the syntax of the language itself.

[2]According to Kruglyk and Lvov, these requirements come from A. P. Ershov, with the reference given as `http://ershov.iis.nsk.su/archive/eaimage.asp?did=41919&fileid=224284`. The original content is in Russian, so I have decided to trust the English version from Kruglyk and Lvov.

> Cross-platform – it is advisable not to bind a student to a certain platform, because a user should choose the platform to work on his/her own.

This is important because a student should be able to install the software on their own personal computer, and not need to rely on access to, say, a school lab computer.

> ... modern means of interface are so well developed, that their existence should not be ignored even at the elementary stages of learning programming. From our point of view, the use of console for the input-output is nowadays insufficient.

On the contrary, this paper takes the view that it is advisable to quickly disabuse students of the notion that everything a computer does will be part of a graphical application of the sort they will be most familiar with. No part of the basic elements of programming requires a GUI; rather, this is a more complex concept that can easily be left for later stages of learning.

> The syntax of programming language should be as simple and clear as possible, to make a program not only easily written, but also easily read and understood.

More than any of the others, this was a guiding principle for the project at hand. In fact, it is this principle that leads to the use of natural language syntax. Among proponents of natural language models, the idea that natural-language programming is simpler for non-programmers to understand is near-universal. Certainly, Halpern thought so; additionally, in their paper "Feasibility Studies for Programming in Natural Language," Lieberman and Liu have the following to say:

> We want to make computers easier to use and enable people who are not professional computer scientists to be able to teach new behavior to their computers. The Holy Grail of easy-to-use interfaces for programming would be a natural language interface—just *tell* the computer what you want!

In general, these proponents are referring to active languages (see Section 2.1 for the distinction), but the hypothesis outlined in the introduction can be restated under this lens as positing that even a passively-natural programming language syntax will be easier to absorb for newcomers than that of a traditional language.

3 The Newspeak Language

This section will cover the terminology and syntax of Newspeak, and the reasoning behind the choices made.

3.1 Outline

As with most languages, Newspeak programs are made up of statements, and the statements do work on values as resolved from expressions. Statements in Newspeak are designed to look like English sentences wherever possible, and end with a period. If statements can be viewed as sentences, then expressions are to be found within statements as the grammatical

subject and/or object. One or more statements within a file constitutes a Newspeak Program (with the conventional extension `.nsp`), which can be run by being passed to the Newspeak Interpreter.

3.2 Statements

There are four types of statement in Newspeak:

- Commands
- Facts
- Questions
- Control Blocks

These are all statements from the perspective of programming language theory, which is to say, they are basic units of code, and as this is a technical document, they will continue to be referred to collectively as such. However, they are separate from each other within the user-facing terminology of Newspeak. This is because the term "statement" will not mean much to a beginning student, and each of the concepts, by their standard definitions, presents a linguistically distinct category as outlined in the following paragraphs.

3.2.1 Commands

Commands are orders directly issued from the programmer to the interpreter. They are designed to be written as imperative sentences. The standard IO commands are good examples (here and in following examples from Section 3, `x` is the name of a variable):

```
Print x.
Read into x.
```

Commands are the typical mode for interacting with the interpreter, and are the most numerous type of statement. For a full list of built-in commands, see Appendix A.

3.2.2 Facts

Facts are statements of fact about the program's state. The only built-in fact statement is the Equality statement:

```
X is "hello world".
```

This is akin to the **Set** command, but may feel more natural to some students, and may work better in some contexts; for example, consider the following two stubs that begin to store a function in a variable:

```
Area is a function given length, width:
Set area to a function given length, width:
```

The first is fairly natural, as far as a complex concept can be stated in plain language; the second, conversely, is awkward, and although the interpreter will interpret it as equivalent the first, to a human eye it may be somewhat unclear whether the `given` clause is supposed to pair with `area is a function` or with `Set`. A more experienced user would probably deduce that the latter of those interpretations is meaningless and therefore the former is likely correct, but a beginner could certainly be confused.

3.2.3 Questions

Questions are the linguistic counterpart to facts, and query the interpreter about the program's current state. They are an exception to the general rule that a statement should end with a period, as questions should end with question marks. There are a few useful built-in questions; the most obvious for demonstrative proposes is called the **Identity** question, and is the counterpart to the Equality fact:

```
What is x?
```

For the most part, this behaves like the **Print** command, except that it surrounds strings that it outputs in quotation marks, so that for example, the integer `7` and the string `"7"` are disambiguated. This makes it more useful than Print for debugging, but less useful for UI output. In general, questions are debugging tools, since they are asked by the programmer to the interpreter, and therefore the response from the interpreter is directed to a programmer, and not a program's end-user. A complete list of built-in questions can be seen in Appendix A.

3.2.4 Control Blocks

A control block is a statement that doesn't do anything by itself, but which informs the interpreter about how and when to execute one or more statements within a block of code that it provides.

Control blocks are compound statements, meaning they have their control portion, and then a further portion composed of other types of statement. The control portion will end in either a comma or a colon; the comma indicates that the statement will be completed with another statement on the same line following the comma, and be terminated with a period, whereas the colon indicates that an entire block of code on subsequent lines will be part of this compound statement. We can see these two different styles in the examples below of two ubiquitous control statements, **If** and **While**:

```
While x is less than 0:
        Print x.
        Increment x.

If x is equal to 7, print "x is 7".
```

The single-line variant is simple and concise. The block variant, though equally intuitive, merits extra discussion. The main concept of note is that the indentation is syntactically significant; this is sometimes called the "offside rule" [5]. The coiner of this term, Landin,

felt it was a way of removing the physical representation from the syntax, so that it would feel as natural written longhand as typed. As this is the natural way to indicate a block of related lines under a common heading in other forms of writing, it was deemed the most intuitive and therefore the best choice for Newspeak to meet its goals. The specific requirement is that all statements at the same level within a block begin with the same whitespace indentation, character-for-character, and that nested blocks be indented more deeply. Either tab characters or space characters may be used in the indentation, but they cannot be mixed within the same block. A document is permitted to use both tab and space characters in different locations, but it is not recommended[3].

A full list and description of the built-in control block statements can be found in Appendix A.

3.3 Expressions

An expression is the component of a statement that is meant to generate a value. Newspeak offers the following categories of built-in expressions:

- Literal expressions.
- Names of variables, which resolve to their values.
- Math expressions for binary operators.
- Expressions for operators that have boolean results.

In addition to these categories, there are several other ones less conducive to being grouped together, and a complete list of available expressions with descriptions and syntax for each of these can be found in Appendix B.

Most of the above feature uninteresting syntax[4], but a couple of the most complex expressions are worth drilling into: the **Function** expression, and the **Function Result** expression (both of which are closely tied to the **Run** command, which runs a given function without looking for a result). The following is an example of a Function expression being used to assign a function to a variable with the Equality fact:

```
Area is a function given radius:
        Pi = 3.14
        Result: pi * radius^2
```

Setting this apart from all other expressions is the fact that the block of code is part of the expression, and not directly tied to the surrounding statement as other blocks are. In general, these blocks are not different from blocks in control statements, but they do feature

[3]In general it is best to forbid practices that aren't recommended, but this decision was made so that files which have been moved between different text editors could be used naturally without strange errors popping up, even if some of these insert tabs from the Tab key, and others insert a set number of spaces. There is still the possibility of problems arising if the same block is edited from two such different editors, but that was deemed unavoidable, since there is no universal for the number of columns a tab occupies.

[4]Indeed, intuitive syntax should never be surprising.

the ability to use the special pseudo-command `Result`, which sets the expression to the right of the colon as the result of this function. Interestingly, unlike many programming languages, providing a result does not cause the function to return; there is no intuitive need for these actions to be intimately tied. Exiting early from a function body can be accomplished with the **Exit** command, but this need not be tied to a result.

Functions can be run with their results ignored, as mentioned, but generally if a function provides a result it will later be accessed, and this can be accomplished with the Function Result expression. Observe the following line, which makes use of the `area` function defined above:

```
Print the result of area with radius 7.
```

This will run the function stored in variable `area`, and resolve to the result set within the function. An error will be raised if the function does not set a result, or exited before a result was set. Each parameter is always named before the corresponding argument is provided, so that the code will read better and be less telegraphic. A consequence is that arguments do not need to be given in the same order as the parameters were initially specified. Multiple arguments are separated by commas, as in the expression `the result of area with length 7, width 3`.

3.4 Flexibility

The English language is in general not rigid, and although a formal programming language must of necessity be less flexible than a natural language, there are still some flexible elements incorporated into Newspeak so users can express themselves as naturally as possible under a variety of circumstances. The option to sometimes issue commands and other times make statements of fact, even for the same functionality, is one example of this.

Another area of flexibility in Newspeak that traditional programming languages may not feature is selective case insensitivity. This is designed so that any element that begins a statement, and therefor a"sentence", can start with a capital letter, as is tradition. In particular, all identifiers in Newspeak are completely case-insensitive, as most nouns are in English. This allows, for example, that the line `Rad is 7.` could be followed smoothly by `Print the result of area with radius rad.` Additionally, all keywords that can come at the beginning of a line are permitted to start with a capital letter (`Print "text".`), but are not required to (`If x is 9, print "text".`)[5].

Finally, Newspeak is flexible in the style of mathematical notation it permits. The most intuitive manner of expressing equations, and thus most suited to Newspeak's mission, is to spell the words out:

```
If x times 8 plus 3 is less than 7:
        Result: x.
        Exit.
```

[5]Arbitrary letters in keywords are not permitted to be uppercase; there is no sense in allowing forms like `iF` or `sEt`.

As this exemplifies, typing out equations in such a manner is painfully verbose. It will be very rare that a student who is being taught Newspeak will not have taken some form of math course in the past, and so to avoid frustrating and needlessly burdening students who are perfectly comfortable with standard mathematical symbols, the following syntax is also accepted for the first line of the above example:

```
If x * 8 + 3 < 7:
```

Statements and expressions that have a math variant will list this variant in their entries in the Appendices.

4 The Newspeak Interpreter

The Newspeak language is implemented by an official interpreter[6], inventively named the Newspeak Interpreter. An interpreted style was chosen because it eliminated the compilation step, thus being simpler for a new student to understand. Projects written in Newspeak will often be small and single-use, and will never be performance-sensitive, so the benefits of compilation are largely moot.

The interpreter is written in Kotlin, with lexing and parsing via the Antlr framework. The Kotlin code pre-processes the input, detecting changes in depth of indentation and translating them to characters representing indent increase (arbitrarily chosen as the little-used backtick `), and indent decrease (arbitrarily chosen to be the tilde ~).

Antlr is not well-equipped for dealing with arbitrary-length input, so the Kotlin frontend delivers the input to the parser on a line-by-line basis, except in the case where a line opens up a block, in which case the lines will be buffered until the outermost block closes, so that the parser is always receiving an entire statement. As a consequence of the significant-indent style, the interpreter cannot know a decrease in indentation has occurred until it reads the next line. This has little effect on a file being passed through the interpreter, but a user in interactive mode will notice that they need to add an extra line after the outermost block closes if they want the effects of that block to take place before they enter the next line of code. The formal Antlr grammar file used in the current version is included as Appendix C.

Newspeak evaluates its expressions eagerly. Newspeak is lexically scoped. Since there are no separate declarations and assignments of variables in Newspeak, shadowing does not generally exist; referencing a variable that exists in an outer scope will refer to the existing variable, while if a variable is referenced for the first time in an inner scope, it will go out of scope when the block closes. The exception is when a function expression is used; then, even if an outer variable has the same name as one of the function parameters, the function parameter will be a new variable that shadows the outer one within the function's body. Otherwise, only outer variables at the global scope are accessible within function bodies, because other variables could go out of scope before the function is called. The behaviour outlined in this paragraph is technically implementation detail, it would be possible to fully implement the language described in Section 3 with different behaviour; however, such an implementation would be considered non-compliant.

[6]Third-party implementations are not discouraged, but for the remainder of this section, terms like "the interpreter" will refer specifically to the official implementation.

The command-line interface of the current version of the interpreter is quite simple; if no file is provided, it will start in interactive mode so that code can be typed to standard input, and if the name of a file is provided, that file will be run as a program.

5 Examples

This section will lay out some programs written in Newspeak so as to demonstrate its qualities.

5.1 Hello World

Newspeak's syntax allows for a quite concise implementation of the traditional Hello World program:

```
Print "Hello World!".
```

This will result unsurprisingly in the following output:

```
Hello World!
```

5.2 Dimension Calculator

Let us complete a typical early assignment for a Computer Science student. The task: Create a program that allows a user to select either a rectangle or a circle, then asks for dimensions, and finally outputs the area and perimeter of the shape. We won't worry much about input error resistance, we just want a basic example of how Newspeak code is written.

```
Pi is 3.14.
CircleArea is a function given radius:
        Result: Pi * radius^2.
CirclePerimeter is a function given radius:
        Result: Pi * r * 2.
RectangleArea is a function given length, width:
        Result: Length * width.
RectanglePerimeter is a function given length, width:
        Result: Length * 2 + width * 2.

Print "Are you entering a rectangle, or a circle?".
Read into shape.

Set area to 0.
Set perimeter to 0.

If shape = "rectangle":
        Print "Enter the length:".
        Read a real number into l.
```

```
        Print "Enter the width:".
        Read a real number into w.
        Area is the result of rectangleArea with length l, width w.
        Perimeter is the result of rectanglePerimeter with length l, width w.
Otherwise:
        Print "Enter the radius:".
        Read a real number into r.
        Area is the result of circleArea with radius r.
        Perimeter is the result of circlePerimeter with radius r.

Print "Area:".
Print area.
Print "Perimeter:".
Print perimeter.
```

6 Future Development

Newspeak is a project that warrants continuous improvement. It has reached a critical point where it has become functional, but this section will outline ideas for greatly improving it.

6.1 Type Definitions

Newspeak is not an object-oriented language, because that adds complexity, and Newspeak values simple and intuitive design over all else. However, almost every language has the ability to define custom types of some sort, and adding a type-definition format for Newspeak would make an excellent late-stage learning opportunity. The syntax would be a little less natural-language because the complexity would be higher, which is why it would be designed for use by students who are already comfortable with the basics, or by instructors who want to add in certain functionality for students to use on assignments. A Newspeak type would outline properties, and allow for custom versions of each of the statement types outlined in Section 3.2 within a predefined syntax framework. Certain facts, commands, and expressions would be automatically generated for getting and setting properties.

6.2 Standard Library

Newspeak currently supports all basic functionality, but a greater host of standard functionality would make it much more usable. For instance, a student should not need to manually define pi every time they wish to use it, as was done in Example 5.2. Many math functions could have standard implementation as well, and there could be an even greater set of built-in statements that work on, for example, lists. And certainly, once types are introduced into the language, it would make sense to ship a strong standard library of types written in Newspeak. As a more distant goal, it would be useful to provide a portal through which Java code and classes can be accessed from Newspeak, although this poses some challenges for

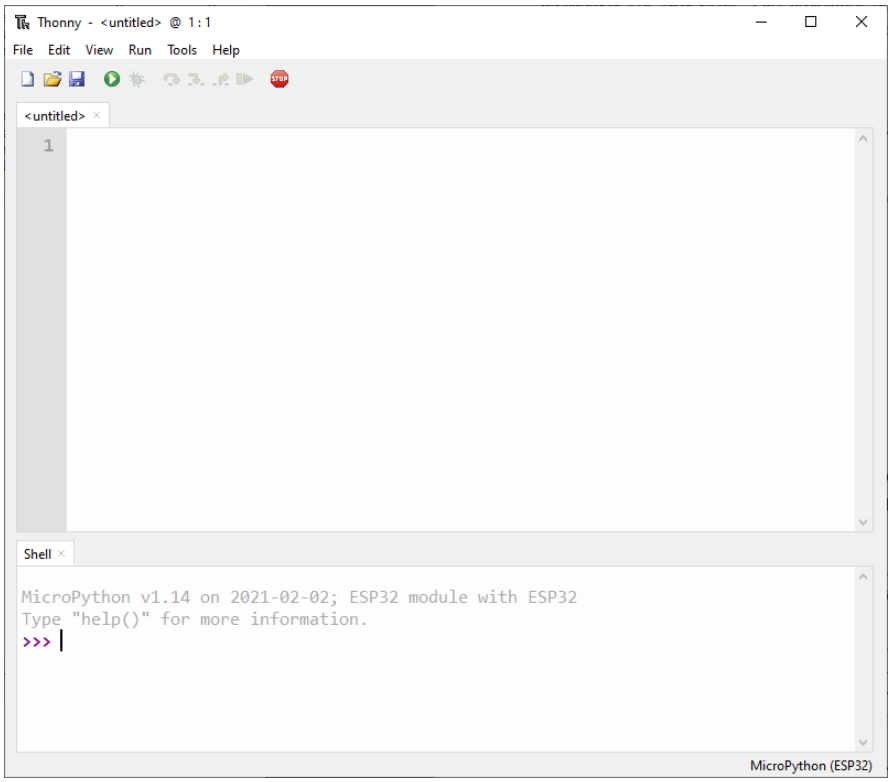
natural-language integration, and is less pressing as the target audience of Newspeak does not generally need advanced features.

6.3 Debugging

The question statements are the main debugging tool currently present in Newspeak, but they are severely elementary, and a learning student might need more help than the current interpreter can offer them. One intriguing option proposed by Myers, Pane, and Ko [6] is to keep track of decision trees, so that a user can ask the interpreter why a certain decision was made, and it can retrace the trees to explain itself. Full debugging tools with breakpoints and such are likely too advanced for the target users to need or understand, but better general debugging facilities could be helpful.

6.4 The Environment

Currently the whole environment of Newspeak amounts to a simplistic command-line interface with a low-feature interactive mode. As a tool that will suffice for studying the effectiveness of this paper's hypothesis, that's probably sufficient. However, if Newspeak is ever to become a genuine teaching tool, it needs more. At the very least, it should have a minimal IDE, with a text editor on top that has features like auto-indentation, and an interactive interpreter at the bottom of the window. The Python IDE Thonny[7] is a good example of the type of interface I mean:



[7] `https://thonny.org/`

Additionally, Newspeak should have more than a barebones interactive mode. A better Newspeak for classroom use would have a fully-featured interactive Newspeak Shell with command history and tab completion, making use of a tool like JLine[8].

6.5 Beyond English

English is the language that I speak, so Newspeak is English-like. Any programming language designed to look natural to native speakers of some language will be relatively inaccessible to large portions of the world; English was only about the 4$^{\text{th}}$ most natively-spoken language in 2015 [7]. The good news for Newspeak is that English is the most commonly-learned foreign language in the world, so there should already be a large audience with some ability to make use of it. Still, it would be a dream come true if other programmers decided to follow the same model and create programming languages that are intuitive to speakers of many more worldwide languages.

7 Conclusion

Let us restate the hypothesis that led to this paper: students will learn the fundamentals of computer programming more quickly if they start out using a language that is more familiar and intuitive than traditional languages. This paper must be evaluated based on the fitness of Newspeak to test such a hypothesis. I am satisfied with the natural-language appearance of Newspeak, and I think it is successful in being very intuitive with the English language as reference. Several elements of the Future Development section would be best implemented before the beginning of a study of the hypothesis; without custom types and a larger standard functionality, Newspeak may not be usable enough to compete with traditional languages despite its benefits. Once a few of these more complex steps are taken, Newspeak will be the ideal language to test out the idea that a student will learn better when they aren't constantly confused by syntax.

The ongoing development of the Newspeak Language and Interpreter can be found at `https://github.com/tradeJmark/Newspeak`.

References

[1] M. Halpern, "Foundations of the case for natural-language programming," *IEEE Spectrum*, vol. 4, no. 3, pp. 140–149, 1967. DOI: 10.1109/MSPEC.1967.5216263.

[2] G. Nelson, *Writing with Inform*, Available online at `http://inform7.com/book/WI_1_1.html`.

[3] V. Kruglyk and M. Lvov, "Choosing the first educational programming language," in *ICT in Education, Research and Industrial Applications: Integration, Harmonization and Knowledge Transfer: Proceedings of the 8th International Conference*, ser. ICTERI, 2012, pp. 188–198.

[8] `https://github.com/jline/jline3`

[4] H. Lieberman and H. Liu, "Feasibility studies for programming in natural language," in *End User Development*, H. Lieberman, F. Paternò, and V. Wulf, Eds., Dordrecht, The Netherlands: Springer, 2006, pp. 459–473.

[5] P. J. Landin, "The next 700 programming languages," *Commun. ACM*, vol. 9, no. 3, pp. 157–166, Mar. 1966, ISSN: 0001-0782. DOI: 10.1145/365230.365257.

[6] B. A. Myers, J. F. Pane, and A. Ko, "Natural programming languages and environments," *Commun. ACM*, vol. 47, no. 9, pp. 47–52, Sep. 2004, ISSN: 0001-0782. DOI: 10.1145/1015864.1015888.

[7] "The world's languages, in 7 maps and charts," *The Washington Post*, Apr. 2015. [Online]. Available: https://www.washingtonpost.com/news/worldviews/wp/2015/04/23/the-worlds-languages-in-7-maps-and-charts/.

Appendix A Built-In Statements

A.1 Commands

- **Print**

  Syntax: `Print <expression>.`

  Prints the result of `<expression>` to standard output.

- **Read**

  Syntax: `Read [a[n] <type>] into <id>.`

  Reads a line from standard input into a variable named `<id>`. By default the value will be a string, unless `<type>` is specified. For possible values of `<type>`, see A.1.1.

  Examples:

  ```
  Read an integer into x.
  Read a real number into y.
  ```

- **Set**

  Syntax: `Set <id> to <expression>.`

  Sets the value of the variable named `<id>` to the result of `<expression>`.

- **Increment**

  Syntax: `Increment <id>.`

  Increments the value of the integer stored in the variable named `<id>`.

- **Decrement**

  Syntax: `Decrement <id>.`

  Decrements the value of the integer stored in the variable named `<id>`.

- **Add To List**

  Syntax: `Add <expression> to <lexpr>.`

  Adds the result of `<expression>` onto the end of the list obtained from `<lexpr>`.

- **Remove From List**

  Syntax: `Remove item <iexpr> from <lexpr>.`

  Removes the item at the location specified by resolving `<iexpr>` from the list found by resolving `<lexpr>`.

- **Run**

  Syntax: `Run <id> [with (<pid> <expr>)...].`[9]

  Executes the function stored in the variable named `<id>`. If the function takes arguments, they must be passed using the `with` construct, where `<pid>` is the name of the parameter, and the value to be passed is resolved from `<expr>`. The `...` notation indicates that multiple arguments can be passed in such a fashion; they must be separated by commas.

  Example: `Run printArea with length 7, width 3.`

- **Exit**

  Syntax: `Exit`

  Ends the currently running program if executed at the top level. Exits a block otherwise (will break a loop or cause a function to return).

A.1.1 Readable Types

The following types can be read from the command line as processed by the same rules outlined in B.1:

- integer
- real number
- boolean

A.2 Facts

- **Equality**

  Syntax: `<id> is <exp>.`

  Math variant syntax: `<id> = <exp>.`

  States that the variable named `<id>` is equal to the result of `<exp>`. The internal state is then updated to record this fact, as if the **Set** command had been used.

[9] The brackets here are not literal, they just indicate that both `<pid>` and `<expr>` will be repeated together.

A.3 Questions

- **Identity**

  Syntax: `What is <expression>?`

  Shows the value of the result of `<expression>` on standard output. Similar to using the **Print** command, except that strings will be wrapped in quotation marks so that it's clear that, for example, the integer `7` is different from the string `"7"`.

- **List Membership**

  Syntax: `Is <vexpr> in <lexpr>?`

  Answers by outputting `Yes` if the value found by resolving `<vexpr>` is present in the list resolved from `<lexpr>`, or `No` otherwise.

- **List Emptiness**

  Syntax: `Is <lexpr> empty?`

  Answers by outputting `Yes` if the list found from `<lexpr>` is empty, `No` otherwise.

A.4 Control Blocks

In the following syntax examples, `*` represents either a single statement or a block of code, as detailed in Section 3.2.4.

- **While**

  Syntax: `While <expression> *`

  Executes * in a loop for as long as the result of `<expression>` is true. `<expression>` must resolve to a boolean.

- **If**

  Syntax: `If <expression> *`

  Executes * if the result of `<expression>` is true. `<expression>` must resolve to a boolean.

- **Otherwise**

  Syntax: `Otherwise *`

  Invalid unless it immediately follows an **If** statement. Executes * if the corresponding If statement's boolean expression resolved to false.

- **For Each**

  Syntax: `For each <id> in <expression> *`

  Executes `*` once for every entry in the result of `<expression>`. `<expression>` must resolve to a list. Each successive iteration makes a successive item from the list available to `*` under the name `<id>`.

  Example: `For each line in lines, print line.`

A.5 Special

Special syntax exists so that both the **Set** command and the **Equality** fact can set an item at a given position in a list: `Set x at position 3 to "I'm #4".` Although in this context the list item selector is not seen as an expression, the syntax is fully described in the Appendix B.4 expression entry for a **List Item**.

There is also a special pseudo-statement for specifying the result of a function, which is only valid within the body block for a function. It is described in Section 3.3.

Appendix B Built-In Expressions

B.1 Literals

- **String**

  Syntax: `"<text>"`

  Resolves to a string with value `<text>`.

- **Integer**

  Syntax: `<digit>+`

  Resolves to an integer. The `+` notation is here meant to denote "one or more", and `<digit>` is any one of the base-10 Arabic digits from 0 to 9. The `<digit>` values will be interpreted together as a base-10 number.

- **Real Number**

  Syntax: `<digit>+.<digit>+`

  Resolves to a real number. Here `+` and `<digit>` have the same meaning as above for Integers[10]. The digits before the `.` are the whole part, and the digits after are the fractional part.

- **Boolean**

  Syntax: `true | false`

  Resolves to a corresponding boolean. Not case-sensitive.

- **List**

  Syntax: `[<expr>...]`

  Resolves to a list of given `<expr>` values. The `...` notation indicates 0 or more iterations of `<expr>`, which are separated by commas.

  Example: `[1, 3, 6]`

[10] Although `+` here has the same meaning as in regular expressions, I am not in general using regular expression syntax; for example, the `.` in Real Numbers is meant to be interpreted as literal.

B.2 Algebraic Operators

The following are organized with highest precedence first. Unless otherwise indicated, `x` and `y` stand in for expressions that resolve to numbers (real or integer).

- **Exponentiation**

  Syntax: `x to the power of y`

  Math Variant: `x ^ y`

  Resolves to a real number if either `x` or `y` is real; otherwise, an integer. The value is `x` to the power of `y`. This operator is right-associative.

- **Multiplication/Division**

  Syntax:

  Multiplication: `x times y`

  Math Variant: `x * y`

  Division: `x divided by y`

  Math Variant: `x / y`

  Multiplication will resolve to a real number if either `x` or `y` is real; otherwise, an integer. Division will always resolve to a real number.

- **Addition/Subtraction**

  Syntax:

  Addition: `x plus y`

  Math Variant: `x + y`

  Subtraction: `x minus y`

  Math Variant: `x - y`

  Resolves to a real number if either `x` or `y` is real; otherwise, an integer. In the case of the addition operator, `x` and `y` may also be strings, in which case the result is a concatenation.

B.3 Boolean Operators

The following all have equal precedence. Unless otherwise indicated, `x` and `y` stand in for expressions that resolve to numbers (real or integer).

- **Greater Than**

  Syntax:

  Exclusive: `x is greater than y`

  Math Variant: `x > y`

  Inclusive: `x is greater than or equal to y`

Math Variant: `x >= y`

Resolves to true if x is greater than y (or, in the inclusive case, equal to y), otherwise false.

- **Less Than**

  Syntax:

  Exclusive: `x is less than y`

  Math Variant: `x < y`

  Inclusive: `x is less than or equal to y`

  Math Variant: `x <= y`

  Resolves to true if x is less than y (or, in the inclusive case, equal to y), otherwise false.

- **Equality**

  Syntax:

  Positive: `x is equal to y`

  Math Variant: `x = y`

  Negative: `x is not equal to y`

  Math Variant: `x != y`

  If `x` and `y` are equal, resolves to true in the positive case and false in the negative case; the inverse is true when they are not equal. This operator is defined for any type. Lists will use an element-wise comparison, strings must be composed of exactly the same characters in the same order to be equal, and numbers are considered equal if they are within 10^{-10} of each other.

B.4 Miscellaneous

- **Variable**

  A variable name in a position where an expression is expected will resolve to the value of the variable.

- **Function**

  Syntax: `[a] function [given <parameter>...] <block>`

  Results in a function that can then be called at a later time. The `given` clause is optional, and will provide a comma-separated list of parameter names. The code to be executed is provided in `<block>`, as expanded upon in Section 3.3.

- **Result of Function**

  Syntax: `[the] result of <expr> [with (<pid> <expr>)...]`[11]

[11] The brackets here are not literal, they just indicate that both `<pid>` and `<expr>` will be repeated together.

Resolves to the result of running the function found by `<expression>` with the given `<expr>` results as arguments. A more complete account of the `with` clause can be found at the description for the **Run** command.

- **List Item**

  Syntax: `<lexpr> at position <iexpr>`

  Retrieves the item in the list found by `<lexpr>` at the position expressed by `iexpr`.

  Example: `[1, 2, 5, 8] at position 2`

Appendix C Formal Language

The following is the formal description of the Newspeak Language in Antlr grammar code used in the current official implementation. Note that several consequences of the implementation described in Section 4 result in the divergence of this grammar in some ways from the official description of the language.

```
grammar Newspeak;

ASSIGN : ([Ii]'s');
PRINT : [Pp]'rint';
WHILE : [Ww]'hile';
IF : [Ii]'f';
INC : [Ii]'ncrement';
DEC : [Dd]'ecrement';
EXIT : [Ee]'xit';
SET : [Ss]'et';
TO : [Tt]'o';
FOREACH : [Ff]'or each';
IN : 'in';
POS : 'at position';
A : 'a' | 'an';
FUNC : 'function';
GIVEN : 'given';
RUN : [Rr]'un';
WITH : 'with';
THE : 'the';
RESULT : [Rr]'esult';
OF : 'of';
READ : [Rr]'ead';
INTO : 'into';
REALNUM : 'real number';
WHAT : [Ww]'hat';
ELSE : [Oo]'therwise';
ADD : [Aa]'dd';
```

```
REMOVE : [Rr]'emove';
EMPTY : 'empty';
ITEM : 'item';
FROM : 'from';

DOT : '.';
OPAREN : '(';
CPAREN : ')';
COLON : ':';
COMMA : ',';
INDENT : '`';
DEDENT : '~';
SLIST : '[';
ELIST : ']';
QM : '?';

POWER : 'to the power of' | '^';
PLUS : 'plus' | '+';
MINUS : 'minus' | '-';
TIMES : 'times' | '*';
DIV : 'divided by' | '/';
LT : 'is less than' | '<';
GT : 'is greater than' | '>';
fragment OEQ : ' or equal to';
LTE : 'is less than'OEQ | '<=';
GTE : 'is greater than'OEQ | '>=';
NEQUALS : 'is not equal to' | '!=';
EQUALS : 'is equal to';

EQS : '=';

STRING : '"'.*?'"';
fragment DIGIT : [0-9];
INT : DIGIT+;
REAL : DIGIT+ '.' DIGIT+;
TRUE : [Tt]'rue';
FALSE : [Ff]'alse';

ID : [a-zA-Z][a-zA-Z0-9]*;

WHITESPACE : [ \n\r\t]+ -> channel(HIDDEN);


line : (statement | question | command | control | special) DOT?;
```

```
control : whl | iff | foreach;
whl : WHILE expression (sent | block);
iff : IF expression (sent | block) elsee?;
elsee : ELSE (sent | block);
foreach : FOREACH ID IN expression (sent | block);
block : COLON INDENT line ( line)* DEDENT;
sent : COMMA line;

statement : assignment;
assignment : idOrPos (ASSIGN | EQS) expression;

question : (ident | lmem | lemp) QM;
ident : WHAT ASSIGN expression;
lmem : ASSIGN expression IN expression;
lemp : ASSIGN expression EMPTY;

special : result;
result : RESULT COLON expression;

command : print | incdec | exit | set | run | read | add | remove;
print : PRINT expression;
incdec : (INC | DEC) ID;
exit : EXIT;
set : SET idOrPos TO expression;
run : RUN expression (WITH ID expression (COMMA ID expression)*)?;
type : REALNUM | ID;
read : READ (A? type)? INTO ID;
add : ADD expression TO expression;
remove : REMOVE ITEM expression FROM expression;


idOrPos : ID | expression;

bool : TRUE | FALSE;
list : SLIST (expression)? (COMMA expression)* ELIST;
func : A? FUNC (GIVEN ID (COMMA ID)*)? block;
fres : THE? RESULT OF expression (WITH ID expression (COMMA ID expression)*)?;
expression : INT | REAL | STRING | bool | ID | list | func | fres
        | OPAREN subex=expression CPAREN
        | <assoc=right> expression bop=POWER expression
        | expression bop=(TIMES | DIV) expression
        | expression bop=(PLUS | MINUS) expression
        | expression bop=POS expression
        | expression bop=(GT | LT | GTE | LTE | EQS | EQUALS | NEQUALS) expression;
```