Natural: A Legible Programming Language

Anish Bramhandkar, ab2929
Elba Garza, esg2128
Scott Rogowski, smr2167
Ruijie Song, rs2740

Motivation & Inspiration

Throughout the various brainstorming sessions that our group held, the idea of an easy read and understood language kept floating around. At first, it was a joke: we would call it PLEM, Programming Language for the English Major!

This idea, though, soon became our main focus. What if we could write a language whose construct is very similar to English, so that it's legible to just about anyone? Basically, if you speak English, then you can easily read the code for our language, and know exactly what it is doing.

Natural is friendly, intuitive, and straight up easy to read. It can perform simple algorithmic tasks, supports object construction and also function declarations.

Creation & Refinement

When beginning to design this language, we quickly realized that our biggest possible pitfall is the natural ambiguity of a spoken/written language, especially English. We had to simplify our vocabulary and keep to unambiguous, yet naturally sounding structures.

Our 'Hello World' program is among the simplest of any existing programming language.

```
say "hello world".
```

The 'say' command prints to the standard output, strings are delimited by double quotes, and the period replaces the semicolon as the command terminator.

If any statement construction sounded too awkward in English, we tried not to use it. The repetitive use of connecting words such as 'and' was kept at a minimum as much as possible. Because of this effort, the Oxford comma became our best friend.

```
(lists)
let y be a list containing 3, 6, 9, 12, and 15.
```

Note: To maintain a classic English language feel, commenting in Natural is in between parentheses and on its own line--i.e., a parenthesized expression that is not on its own line will not be treated as a comment.

```
(Modifying list y; indexing starts at 1, since it's more intuitive in English)
let z be the size of y.
let x be 1.
do the following z times:
```

```
let element x of y be ((element x of y) + 1).
let x be (x+1).
```

The creation of our list above illustrates how the Oxford comma allowed us to write a list in a succinct yet still natural manner. Commas separate elements of our list and to keep to proper English, the ", and" indicates that the last of element of our list follows.

```
(assignment)
let x be 5.
let x be 5.0.
let x be "sally".
let x be true.
```

Assignment is done by using the 'let' and the 'be' keywords while the data type is inferred by the compiler. If there are cases where the data type is being manipulated in a manner not inherent to the data type, an exception will be thrown.

Other constructs

```
(loops)
do the following 10 times:
    if x=5 or (y=10 and z>10) then:
        say "Hello ", and say "World".
        say "!".
    otherwise:
        say "X does not equal 5 and y does not equal 10 or z is less than 10".
```

Loops are intuitive and structured with indentation, very similar to Python. We relied on indentation for these loops to avoid the use of brackets. This was one case when commas separating statements would not be very legible or friendly in line length. Notice also that 'do the following' is a single keyword/keyphrase in our language. So, this construct will remain valid:

```
(demonstrating multi-word keywords)
let following be 20.
```

Our language is object oriented.

```
(objects)
a person has a name, an age, and a salary.

(adding a field to the object)
a person has a cat.

let adam be a person whose name is "adam", age is 20, salary is 50000, and
cat is "mimi".
```

Objects are quite simple and elegant. For example, the structure of a person is easily defined. Objects can then be created, with the Oxford comma helping keep legibility clear, while maintaining instantiation succinct.

```
(copy by value vs. copy by reference)
let mary be a copy of adam.
let joan be the same as adam.
```

Our language eliminates the endless confusion between pass by reference and pass by value using the keywords, 'a copy of', and 'the same as'.

```
(prints "charles")
let charles be a person whose name is "charles", age is 30, and salary is
10000.
say charles's name.

(getting a field of an object)
(prints "adam : 50000")
say adam's name, say " : ", and say adam's salary.
```

Instead of the dot operator as is done in most object oriented programming languages, our language simply uses the English apostrophe construction, " 's " possessive in exactly the same way.

```
(functions)
let quadratic of a, b, and c be:
      let x be (-b+sqrt(b^2-4*a*c))/(2*a).
      let y be (-b-sqrt(b^2-4*a*c))/(2*a).
      quadratic is a list containing x and y.

(we can now do...)
say quadratic of 2, 3, and 4.
(equivalently, for those who have some math background)
say quadratic(2,3,4).
```

Functions can also be defined and labeled to be used later on. Once again, we use indentation to linearly list the algorithm if the function. All our functions return something. To define what a function returns, the last line of the function says what the return is expected to be. In the case for quadratic, it's a list containing x and y.

Returning to our 'person' object, functions can also easily be defined within objects analogous to Java methods.

```
(methods)
let a person's birthyear be:
        birthyear is 2010-age.
(prints "1990")
say adam's birthyear.
```

Representative Program

Part of the beauty of our language is how our object structure makes it easy and intuitive to create complex data relationships. In a way this makes our language a natural extension of

LISP without the 60s syntax.

```
(redefining a person)
a person has a cousin, a theorem, and a mom.

(instantiating data)
let pythagorasTheorem of a and b be:
        let x be sqrt(a^2+b^2).
        pythagorasTheoram is a number x.
let Pythagoras be a person whose theorem is pythagorasTheorem.

let Mary be a person whose cousin is Pythagoras.
let Adam be a person whose mom is Mary.
let Rick be a person whose cousin is Adam.

(prints 5)
Say Rick's cousin's mom's cousin's theorem with 3 and 4.
```

Conclusion & Challenges

The most difficult thing about our language is balancing legibility and ambiguity. Human languages, especially one like English, tends to become ambiguous and has certain structural and grammatical rules that may not always be friendly. We also don't want a language too verbose, since typing in such a manner would get tiring and be tedious.

Fighting with English grammar has been the hardest thing so far, what with modifying the rules for the possessive, including an Oxford comma, and ruling out various methods of writing the same code.

For now, we've data structures, simple data types, and objects down. We hope to expand a bit more and making our rules succinct in order to ensure no loopholes or ambiguity.