Anthony Liu

Professor Marlyse Baptista

LING 115

December 18 2017

Programming Languages and Natural Languages

A programming language is a set of instructions that tell a computer what actions to perform. Every application that runs on a computer is written using a programming language. If you've heard of most popular languages used today, *C, Java, Python*, etc, you may wonder - Why don't we use natural languages such as English to program? Surely programming in English is easier to learn and to teach than a programming language? In fact, using natural languages to create programs has been the goal of many computer science researchers (Veres). However, the fact remains that none of these efforts are truly using the "full capacity" of a natural language such as English in programming.

Consider an analogy of writing a list of instructions for baking a cake. Though the instruction of *1. Bake the cake.* may be humorous in its lack of information, the instruction is an essentially correct recipe of making a cake. The ambiguities of natural language cause them to be unsuitable for programming. Even the mundane *x. Stir dry ingredients in bowl until mixed.* is equally uninformative. Stir which ingredients? What does it mean for ingredients to be mixed? How does one stir ingredients? With a spoon? Stir clockwise? Of course, to the average cook, these questions are irrelevant and can answered by common sense. However, when telling a *computer* to bake a cake, what does common sense mean? Does a computer know what it means to stir ingredients? By using natural language to write a recipe, we are left with problems caused by ambiguities and a lack of precision. Programming languages must be exact. The programmer and the computer both agree on how a program should be executed.

Though natural language is unsuitable for direct use as a programming language, its influence and usage in programming languages is undeniable. In Figure 1, we show a simple "Guess the number" game in the programming language *Python*. The meaning of the program

```
1 answer = '42'
2 guess = input()
3 while guess != answer:
4     guess = input()
5 print('You win!')
```

Figure 1: "Guess the number game" in *Python*. In plain words, we can describe the execution of this program line by line. 1. Set the *answer* to be 42. 2. Let the user set the value of *guess*. 3-4. While the value of *guess* is not the same value as the value of *answer*, let the user set the value of *guess* (the *!=* word is an approximation for the mathematical inequality symbol $\neq$.). 5. Finally, print (or display on the screen), "You win!".

can be easily deciphered only using knowledge of English. Knowledge that most programming languages are evaluated line by line from top to bottom is also crucial.

The design and usage of Programming Languages is closely intertwined with spoken languages. The similarities in the number guessing game in Figure 1 to its natural translation is no coincidence. There are two reasons -

1. The general idea of programming languages is the same as natural language - to put ideas into words. When a programming language needs a word to describe an idea, that idea is often already encoded into a word of natural language.

2. As alluded to earlier, a primary goal of programming language designers and programmers is to have programming languages be both easy to learn and understand. Like a large manuscript, the greatest applications will have many subsections rewritten and modified when new features are added or old features are broken. This need is amplified when large teams must work on the same project, where programmers must read and understand their respective programs to build upon it. (As an example, as of 2014, the code for Facebook has over 9.9 *million* lines of code (Pearce).)

   The most *natural* way humans communicate is through natural language. The incorporation of natural language into programming increases the ease of understanding.

There are two main ways natural language is incorporated into programming - the design of the programming language and usage of the language, the way programmer's "speak" the language.

To speak of language design, we must revisit the problem of precision written earlier, the difficulty of encoding our thoughts into the programs. How do we tell a computer how to send our email? How do we tell a computer how to display a photo on the screen? A simple solution is to restrict our language only to *simplest* actions that a computer can do. Any application that is *possible* to create on a computer, is created by a limited set of instructions, or words. Analogous, *any* cooking recipe and technique can be described solely by the activation and release of each muscle in the human body. The act of stirring a bowl is carefully described from the usage of muscle groups in the forearm, to the tightening of the fingers around the spoon. Through this restriction, we sacrifice brevity and clarity for absolute precision. In the computer, these actions often include but are not limited to adding numbers, saving numbers into registers, or memory locations, and "jumping", which can be used to repeat already executed code [1].

Instruction	Mnemonic	Encoding
Load Byte	LB	32
Store Byte	SB	40
Add	ADD	0

Figure 2: Sample of the *MIPS* ISA. For example, when the computer reads the number 40, the computer will store a value into its register, or memory. [3]

These simple sets of instructions are called *Instruction Set Architectures*, or *ISA*'s. Most ISA's will contain less than 100 instructions. A sample of an ISA still used today is shown in Figure 2.

Although we have gained precision through this definition, there is perhaps too much sacrifice in ease of use. Programming in these simple languages, also known as assembly languages, is known to be difficult (Hyde). Programming in assembly is tedious and error prone. Similar to writing assemble is the futile effort of trying to write a recipe using only muscle movements. Aside from being impossible to read, writing a single incorrect muscle twitch can result in a flying knife in the kitchen. [4]

How do we both leverage the precision gained from using assembly languages while making it easier to program? Just like natural languages, programming languages can be translated from one another. Programmers can use *higher level languages*, languages more complex and easier to use, then translate their code into *lower level languages*, such as assembly. With the power of this translation, also known as compilation, language designers have the power to define precise definitions of instructions, the grammar and structure of their language, while being able to give precise instructions to the computer. An example of code in a higher level language, Python, is shown in Figure 1. When running on a computer, this program is translated into a low level language, such as MIPS.

Language(s)	Format
C-style languages (C/C++, Java, JavaScript)	`if (...) { ... } else { ... }`
Python	`if ... :\n\t ... \n else: \n\t ...`

Figure 3: If statements in programming languages.

Words from natural language are constantly borrowed in the design of higher level languages. A prime example of borrowings is English in the current most popular languages. `if (...) then (...) else (...)` statements, are found in almost every programming language, see Figure 3. The concept of this conditional is universal in programming and language, that most programming languages will "borrow" the words if, then, and else to use in the language. The most common words borrowed include
`while, for, var (variable), class`, etc. [5]

Despite the added power and flexibility higher level programming languages employ, the words borrowed from natural language are greatly restricted in meaning. Consider the following command "Print "Hello" to the screen if the value of x is 10." An equivalent statement is "If the value of x is 10, print "Hello" to the screen. The translation into the C language is
`if (x == 10) { printf("Hello"); }`. In C, and in most programming languages, the location and relative ordering of the `if` is important. A missplaced `if` at the back of the `printf` statement renders the program unrunnable!

Notes

1. You may wonder how it is possible a finite set of simple instructions can be used to construct useful programs. In fact, it was mathematically proven by computer scientist Alan Turing that any program that can be written on a computer, can be written using a simple *Turing machine* with (technically) three instructions, move, read, and write (Turing).

3. In practice, computers will read in binary format, a format of representing numbers using only two symbols, 1 and 0.

4. Although assembly languages are difficult, they are powerful in their precision and flexibility. Directly writing in assembly languages still find use today in applications where performance is critical. Though, as computer performance has increased, usage of assembly has equivalently decreased (Hyde).

5. All of the

Works Cited

Hyde, Randall. The art of assembly language. No Starch Press, 2010.

Pearce, James. ”9.9 million lines of code and still moving fast - Facebook open source ....” https://code.facebook.com/posts/292625127566143/9-9-million-lines-of-code-and-still-moving-fast-facebook-open-source-in-2014/. Accessed 17 Dec.
2017.

Turing, Alan M. ”Computing machinery and intelligence.” Mind 59.236 (1950): 433-460.

Veres, Sandor M., and J. Patrik Adolfsson. ”A natural language programming solution for executable papers.” *Procedia Computer Science* 4 (2011): 678-687.