Behind Blocks

A text-based article

Bernat Romagosa i Carrasquer
bernat@romagosa.work

February 19, 2018

Along history, computers have been programmed in several ways. In Babbage's Analytical Engine, a machine designed in the 19th century but never physically built, programs would be punched into cards and then fed into its mechanical reader. Ada Lovelace, regarded by many as the first ever computer programmer, designed a series of algorithms on paper for Babbage's engine, even though she could never test them, as the machine did not exist at the time.

About a century later, programs for the electromechanical computers of the time would be punched into cards or films, while others used to be wired into a plugboard. By means of a combination of cables, sockets and switches, these early coders would build the software that powered several companies and governments of the 1950s.

An important fact to consider is that most of these computer programs were first drawn or written on paper and only punched or wired for the computer afterwards, as the computer would not understand these programs in their textual form. This was the case even for Lady Lovelace's algorithms, which she wrote as mathematical formulae in paper sheets and never got to punch for the Analytical Engine.

Humans are clearly much more used to communicating via written words than to interpreting perforated film strips or intricate wire and switch configurations, so as soon as computers evolved to support programs written in text it was just a matter of time before punch cards found their last resting place in museum displays.

We are now quite used to coding in text form by using a keyboard, but this is just one of the ways in which we can represent programs for the computer to interpret. There are, in fact, numerous modern programming languages in widespread use that do not represent programs as text. Some of them are being used by engineers or mathematicians, others by musicians or visual artists, and others are being used to describe the movements and interactions of 3D characters and objects in animation movies.

The computer does not really care whether we write, draw, punch or wire our programs. The computer will be happy to interpret code in any graphical, physical or written form that we wish to come up with, so why not put our emphasis in designing our languages for the human beings that will use them, instead of designing them for the machine that will run them?[1]

Our natural languages are extremely flexible, to the point that we can make up words and linguistic structures on the fly and still be understood by others. We can quite easily figure out the meaning of sentences written in languages that we do not speak but bear similarities with other languages we know. Our comprehension skills are so advanced that we can understand the most poorly punctuated texts, even when they are ridden with typos and grammatical mistakes. It is only natural, then, that beginners tend to be puzzled when a computer language is unable to interpret a whole program just because a semicolon

[1]The answer used to be "because it's faster", but this argument has been losing weight for a long time now thanks to faster machines and more efficient interpreters.

is missing somewhere, or just because a particular letter in line 24 should have been capitalized.

Most mainstream computer languages expect the programmer to remember in an unforgiving exactitude the names of every single operation and the proper order of each parameter, whereas in our natural languages we can very easily switch a word for any of its synonyms or invert the order of our adjectives while still making ourselves understood.

Fast forward to the present day, we find a relatively new trend in the representation of computer programs, commonly known as blocks-based programming, a term that actually groups together a large variety of very different languages and systems that share only one thing in common: they use a particular graphical metaphor to represent code.

As Abelson and Sussman put it in SICP[2], any language can be defined by the means it provides for us to combine simple ideas into more complex ones. In particular, all languages make use of three different mechanisms to do so: primitive expressions, means of combination and means of abstraction. Primitive expressions are the bare elements that the language exposes to us, i.e. its basic vocabulary. We can construct more complex expressions by making use of the means of combination in the language. That is, we can combine primitive expressions together to form groups of expressions that perform more intricate tasks. Finally, the means of abstraction in a language let us group these combined expressions and give them a name, so that we can reuse them. In some languages, these new structures bear exactly the same weight as any primitive ones, so we can use them seamlessly, as if the language had provided them for us from the very beginning.

In blocks-based programming, primitive expressions are represented as a type of graphical object known as a block. Different block shapes represent different types of primitive expressions, and they also visually give us a hint of how these primitive expressions can be combined with each other. This brings us to the means of combination of blocks-based languages, in which we combine blocks together by dragging and dropping them into matching slots or union points of other blocks. Finally, there are the means of abstraction, and this is where things get interesting.

Some blocks-based languages simply do not have a way to abstract groups of combined blocks into a new block. Others are, in fact, just visual representations of textual languages that get translated into text before being executed by the computer. In these cases, the means of abstraction are given by the original language and often do not map very elegantly to its blocks-based visual representation, which is why some of these representations even omit them.

In some cases, we can give names to combinations of blocks, but the resulting structures are completely different from the primitive expressions (blocks) that the language offers[3].

[2]Structure and Interpretation of Computer Programs. Harold Abelson, Gerald Jay Sussman, Julie Sussman. ISBN 0262011530, 9780262011532. MIT Press, 1985

[3]This is the case of custom blocks in Scratch, that can only be of the "command" type

Finally, there are some languages that let us build our own blocks out of combinations of preexisting blocks and treat them as first class citizens, visually and functionally indistinguishable from the other blocks in the system. This holds true up to the point that we can build blocks that make use of other blocks built by ourselves, as any block in the language truly is treated in the same way[4].

Thus, blocks-based programming is a term that does not tell us much about the language behind these blocks. Punch cards, flow charts, modeling diagrams, blocks, switches, wires and text are all different ways of representing programs, but they do not constitute programming paradigms or programming languages by themselves. Saying "I am a blocks-based programmer" gives out as little information as saying "I am a score-based musician" or "I am a written-word based literature enthusiast". These sentences only tell us about your preferred representation, not about your favorite programming paradigm, musical style or literary category.

However, all music score systems share a bunch of properties with each other, and so do all blocks-based systems. Let us explore the advantages these systems have in common.

In textual languages primitive expressions are represented as words, and the programmer is expected to type these exactly as the interpreter or compiler wants them to be, but blocks-based languages represent these primitive expressions as graphical blocks that are pre-given to us, making it impossible for the user to make a mistake at the primitive expression level. Additionally, text-based languages typically use punctuation, indentation and other written constructs to visualize the means of combination in the language, whereas blocks are combined by stacking them together or embedding them into other blocks. It is easy to forget a punctuation mark or misplace a comma, but these errors simply do not exist in blocks-based systems, as syntax is replaced by visual constructs. We can obviously still make a programming mistake and connect two blocks together when we meant to connect different ones, but we cannot make a syntax error even if we try to.

Another shared advantage of pretty much all blocks systems is that they typically expose all the primitive expressions to the user, categorized according to their purpose, thus eliminating the need to memorize instructions and symbols. When we want to use a block that deals with sound, we just need to head to the "Sound" category and scroll to find the one that fits the purpose. In some languages, we can search for blocks by typing and, in some others, we can parse text into actual blocks[5].

To summarize, what blocks bring to programming is not a new way to program computers, but a new way to *represent* programs. Two blocks-based languages can bear as many differences with each other as x86 Assembly and Smalltalk, or Python and Common Lisp. What this new way of representing

and whose definition must always be visible in the scripting area.

[4]Snap*!* stands out in this area.

[5]GP is a blocks-based language where you can parse textual expressions into block combinations on the fly. Notice that it is usually the other way around.

programs brings to computing is that it does away with the possibility to commit syntax errors, and it removes the need to memorize instructions.

Scratch, Snap*!* and GP, among others, are part of a particular kind of blocks-based programming languages that we like to call *Live, Blocks-Based Languages*. Conceptually, they share much more in common with systems like Logo, Squeak or Self than with other blocks-based environments like App Inventor and most Blockly-based systems.

To build an analogy with music, imagine an instrument that requires players to first write their songs in a music score, then feed it to the instrument and listen to it play the tune by itself. The player-piano, the programmable music box or a software MIDI player are three different instruments that work in this way.

Now imagine an instrument that lets you create sounds in real time by interacting with it with your own body in real time. The guitar, the oboe or the theremin are examples of such instruments.

Both instruments require a human to make music, but they require it at different moments in time and for different purposes. In the first kind of instruments, let us call them static instruments, a human composes a song and feeds it to the instrument, then lays back and listens to it. If there is a mistake in the score, the instrument will need to be stopped. The composer will then examine the score until the offending note is found and corrected, and then the instrument will be fed the patched sheet to be played again.

The second kind of instruments, that we shall call dynamic instruments, work in a very different way, as it is the performer who plays the song in real time. A dynamic instrument allows for improvisation, playing songs by-ear, learning by doing and detecting mistakes just as they happen. Playing the guitar is not by any means easier than composing a tune for a player-piano, but one can tinker with the instrument without any understanding of musical representation systems. The guitar gives the inexperienced beginner the ability to try out stuff without any previous formal training at all, while also giving the professional the ability to reshape and adjust a song on the fly, improvising new riffs or changing the flow of the tune.

Computer languages can be classified in similar categories. Some languages require that you first write your program, then feed it to either a compiler or an interpreter, and then observe how it runs. If the program fails, an error message may be generated, hopefully giving us a hint on what may have caused it to fail. Terminologies in computer science can be quite subtle and controversial, but while the nomenclature police is not watching we will be calling these languages static programming languages. In these, the programmer would be more of a composer and not so much a performer. Some mainstream languages that work in this way are Java, C, C++, and most Blockly-based blocks environments can also be classified as static.

Dynamic languages work in similar ways to the guitar, for they run code as you build it, in real time. You can test a piece of code at any given time

and observe the result immediately, in the same way that pressing a key on a piano will immediately play its corresponding note. Of course, there is a huge difference between dynamic instruments and dynamic languages, and it lays in the fact that, once we have tinkered enough and we are happy with what the program is doing, we can have it be run by itself, independent of human interaction, as if it was a static one.

There have been dynamic languages for a very long time, and the family of these such languages is huge and growing, but it would be very negligent of me to talk about dynamic programming and not begin with Smalltalk, the king of dynamism. Smalltalk was invented at XEROX PARC, the most prolific birthplace of computing marvels, still unmatched to date[6]. Smalltalk embraces dynamism as if it was a holy commandment. In Smalltalk everything is changeable in real time, and you can modify the language itself while it is running, because the language is actually written in itself and you can inspect and change its very implementation at your will, while it is running. In fact, when you write an application in Smalltalk you are changing the system itself, the application becomes just an extension of the language. Smalltalk is the ultimate guitar for programmers, where you can change the tuning of your strings on the fly while you play, or modify a string to have it sound like a trumpet in real time, or inspect a sound wave right while it is being generated and change its gain.

But Smalltalk was not born from nothing, it was inspired by many earlier languages among which we find Logo, a pioneering and incredibly visionary dynamic language designed by Seymour Papert, Cynthia Solomon and Wally Feurzeig in the late 1960s. Logo is probably most famous for its turtle graphics abstraction, a different way to think about maths and geometry based on instructions we feed to a virtual turtle in real time. The turtle also existed in its physical form as a moving, drawing robot, so it is only natural that other educational languages that followed the steps of Logo also tried to stay in touch with the real world through a variety of electronic gadgets and devices. One of the latest languages in this tradition is Scratch, unquestionably the most widespread programming language in the educational community nowadays, and the de facto mother of all dynamic blocks languages.

In Scratch, we are presented with a playful environment where programmable graphical objects interact with each other in a 2D microworld. We use blocks and combinations of blocks (scripts) to give behavior to these graphical objects (sprites). We can run these scripts at any time by clicking on them, or by triggering different kinds of events like key presses, mouse movements or message broadcasting. The system has a parallel task scheduler, meaning that we can run several scripts on several sprites at the same time. Not only we can do that, but we are encouraged to do so, and it is in fact the preferred way to modularize our programs and make them easier to understand and extend.

Blocks, however relevant they may be, are but the façade of Scratch, and thinking of the language only in terms of its blocks-based representation of code is missing a big chunk of the whole point. Blocks do away with many

[6]I won't cite any sources to these claims because a) this is not a formal paper and b) I'm too blinded by Smalltalk's brilliance to be able to look for sources.

entry barriers to programming, but what is really essential to Scratch is its live, dynamic, parallel nature.

Besides all of the seemingly obvious educational advantages, a dynamic language does not guarantee that you will be learning programming as if by magic. The instrument does not make the player, and there are in fact different ways to learn music by using the same instrument. Or, in a similar fashion, you could indeed be using a dynamic language as if it was static. You could write your programs, feed them to the interpreter, watch for the results, correct any mistakes and repeat until you are happy. And, actually, this is how many dynamic languages are taught and learned, and also how many dynamic instruments are taught and learned.

If you were born into a community of music players and got to practice live with them every day, as part of your daily life and in a forgiving and playful manner, it is very likely that you would learn how to play very well, and your instrument would certainly feel very natural to you. Lots of world famous flamenco or jazz players that have learned to play in such environments cannot read a single note out of a music score to save their lives, but will vastly outperform many classic musicians in a live setting.

After all, as Victor Wooten would argue, that is exactly how we learn our natural language, by being immersed in a world of professionals who can already speak it perfectly (he calls it "jamming with the pros"), and by mimicking their sounds in an experimental, forgiving and playful manner until we become speakers of that language. So, although it is clear that dynamic languages are not the sole salvation of computer science education by themselves, it should also be quite clear that static languages (or player-pianos) present many more educational barriers to the novice as they fail to provide means for live interaction, immediate feedback and playful experimentation.

So, when you hear about the next blocks-based language -no further description given- that is going to change the way we all program, try to draw a parallelism and imagine how suspicious you would be of a language that claims to be about to change the world but only tells you that it is text-based.

This work is licensed under a Creative Commons "Attribution-ShareAlike 3.0 Unported" license.