

Snap!GPT - Building Blocks for Generative Artificial Intelligence

Jens Mönig, SAP SE, Walldorf

Summary

In the first IBIS issue (2023), Michael Hielscher introduced SoekiaGPT, a didactic language model based on n-grams. It generates similar-sounding but nonsensical texts from 25 Grimm fairy tales. His idea of explaining the operating principles and characteristics of generative artificial intelligence using simple Markov chains and hallucinating fairy tales (sic!) instead of Large Language Models or opaque neural networks excites me. I wondered if it's possible to further simplify the principle so that learners can program something like SoekiaGPT themselves, even in a graphical programming language like Snap!, which I am involved in developing.

The result is a small program with only two custom blocks and a few scripts. It allows users to generate amusing texts and, in the process, learn about programming with larger datasets. And if one rearranges the components a bit differently, something even more astonishing happens: The program can learn to generate music or images.

Introduction

The Snap!GPT project consists of three parts: Example data (in this case, fairy tales) is loaded and dissected to create a language model, a kind of database.

Subsequently, a function is developed that searches for an entry (here: a word) from the data model that plausibly continues an existing sequence (here: previous words). Finally, the entire process can be made interactive for users.

Snap! is a graphical, block-based programming language collaboratively developed by the University of California Berkeley and SAP for computer science education in schools and universities. In the United States, it is used to teach the College Board-certified "The Beauty and Joy of Computing" curriculum. There is also a growing community in Germany. Snap! runs in web browsers, is free, and open source.

Analyzing Data

Files in various common formats (txt, csv, json) can be loaded in Snap! by dragging them into the Snap! window with a mouse. Alternatively, they can be selected from the File menu using the "Import..." option. Snap! creates a global variable with the file name and assigns its content as the value. Structured data (e.g., csv or json) is automatically formatted into lists or tables, while plain texts remain unchanged as a whole. Because I appreciate Hielscher's example with the fairy tales, I also selected

a few websites with Grimm Brothers' fairy tales and downloaded 30 stories as txt files or copied them from the browser into local text files. I combined the individual texts into a single large file and loaded it into Snap!.

The "split" block can be used to break down a text into various components. I split the collected fairy tales by "word" to obtain a large list with the sequence of all words (see Fig. 1).

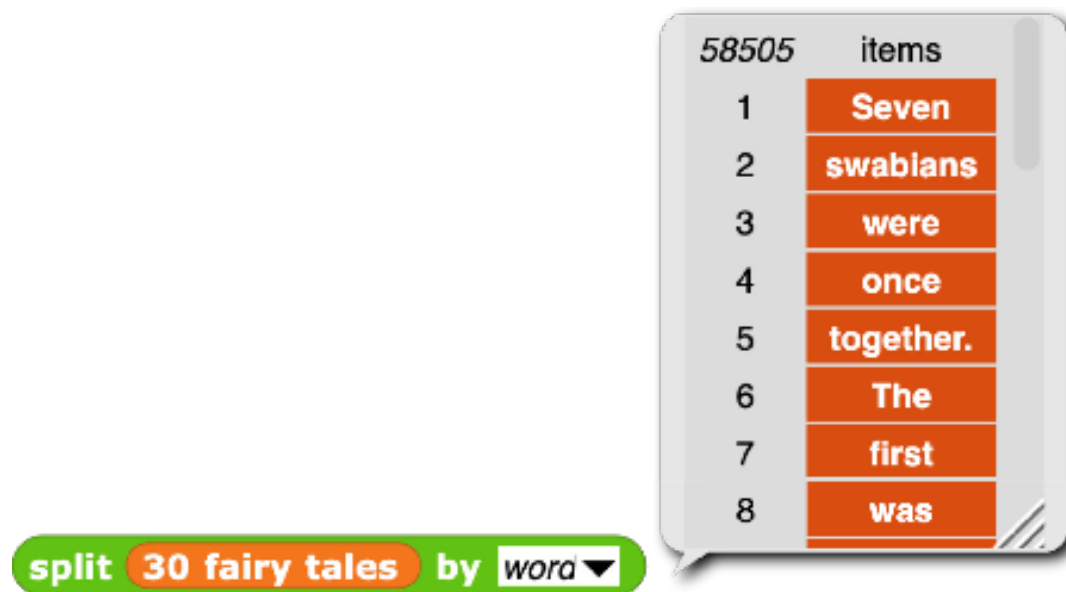


Figure 1: Create a list of words in Snap! using the "split" block.

The highlight of the language model described by Hielscher is the breakdown of this long word list into multiple versions with many small entries consisting of word pairs, triplets, quadruplets, and quintuplets, known as n-grams. For this purpose, I use a function with two parameters: "n" for the desired length of the chains and "corpus" for the initial list of consecutive words, which I have defined in Snap!

N-grams can be implemented in Snap! in various ways, such as imperatively with commands and even hyper-dimensionally with vector operations. I opted for a functional variant because it is the most expressive, requiring the fewest blocks, and accomplishes the task quickly (Fig. 2). Those willing to accept slightly longer runtimes can also work with loops and variables just as effectively.

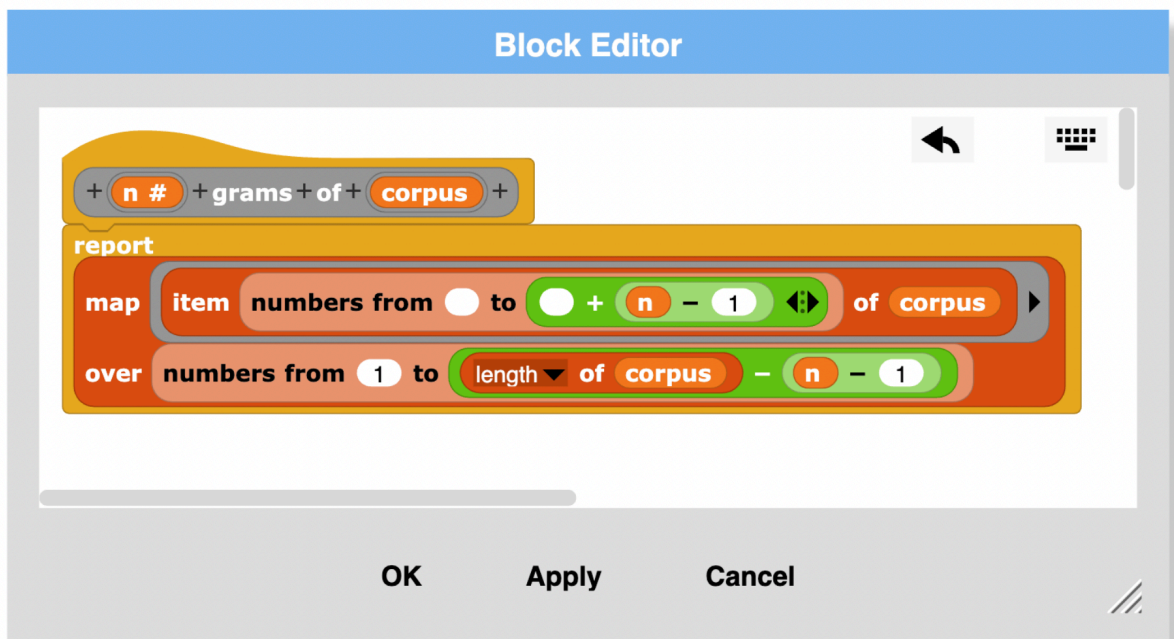


Figure 2 Example for a custom "n-gram" function in Snap!

Next, n-grams are calculated for word chains ranging from length one to five and stored in a language model. For this, I created another global variable named "model." It should be assigned a list whose elements are the results of the corresponding "n-grams" function. The first element is a list of all individual words, followed by an element containing all word pairs, the third element being a list of all 3-link chains, and so forth. This can be achieved, for example, with a for loop and a loop variable. Once again, I opted for a short functional script (see Fig. 3).

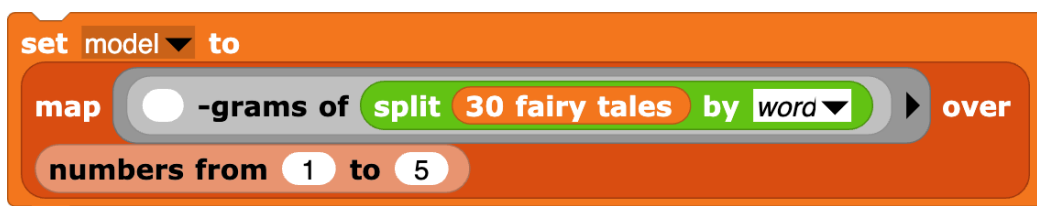
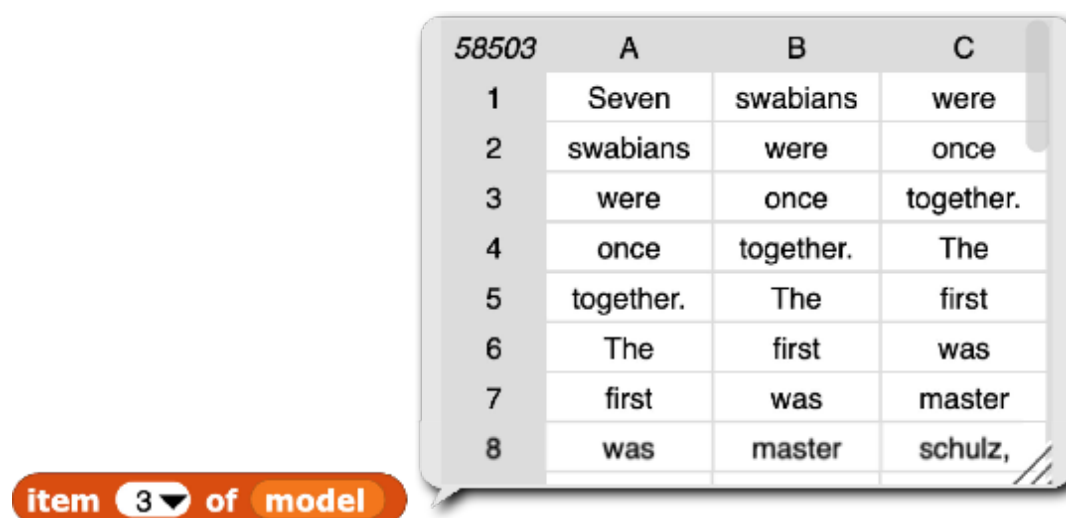


Figure 3 The final Snap! script for "training" the statistical language model.

It is sufficient to click the script once with the mouse and execute it. After a (hopefully short) period, the language model is ready for use. To verify this, the model can be queried with the "item of" block for different n values from 1 to 5. Each time, a list or table should be displayed, with the number of columns matching its position in the model (see Fig. 4).



58503	A	B	C
1	Seven	swabians	were
2	swabians	were	once
3	were	once	together.
4	once	together.	The
5	together.	The	first
6	The	first	was
7	first	was	master
8	was	master	schulz,

item 3 of model

Figure 4 Inspecting the language model in Snap! The number of columns in the table should match the index selected in the "item of" block.

Typically, frequency analyses are conducted at this point, examining how often a word or a specific n-gram appears in the sources. To simplify the project, I skipped this step and instead just write word chains that occur multiple times into the model multiple times. When the next word is selected randomly later, more abundant word chains will be picked with a higher frequency. Here, probability calculus from math class can be addressed.

Tip: Marking a variable as "Non-Persistent"

When saving a Snap! project, the values of all global variables are retained. This has the advantage that, for example, you don't have to re-import the file with the fairy tales every time you continue working on the project. The same goes for the model of n-grams. However, this can lead to the project accumulating large amounts of data, potentially exceeding the storage limit for the cloud. To prevent this, individual variables, such as the model, can be marked as "non-persistent" in their context menu in the palette. Of course, when reopening the project, you will need to execute the script with the n-grams (Fig. 3) again to restore the model.

Continuing a Sequence

Next, the task is to complete a chain of words to a sentence using the language model. Specifically, this means searching for a word in the model that can be added to a list of previous words without resulting in gibberish. This seems like a challenging task, as it requires extensive knowledge of language, grammar, sentence components, cases, forms, exceptions, and familiarity with punctuation and capitalization. The model, however, knows nothing about any of these. Nevertheless, it has a vast reservoir of words that have appeared after other words. These correlations are sufficient to form somewhat authentic-sounding sentences.

The Markov chain text generator algorithm takes the last "n" words of the started story as token and first retrieves all entries in the model that are exactly one word longer than the token. From these entries, it retains those that match the token for the last "n-1" words (i.e. except for the last word). One of these selected candidates is randomly chosen, and its last word is returned as the result. If no candidate is found, the search is repeated with successively shorter tokens until a random word is returned from the 1-gram list eventually.

When generating language, the token length for the search is occasionally chosen at random to make the system more "creative," avoiding the reassembling of chopped-up original texts. In Hielscher's SoekiaGPT, this can be configured using a "temperature" parameter. I have omitted this detail to facilitate the reprogramming. Additionally, I find it interesting if the generated sentences still reveal their origin, i.e., from which fairy tale they were taken. SoekiaGPT visualizes the sources in the generated text with colors, but I have also skipped this.

For the implementation of the custom "next item in" function in Snap!, I opted for a classic imperative variant this time (see Fig. 5).

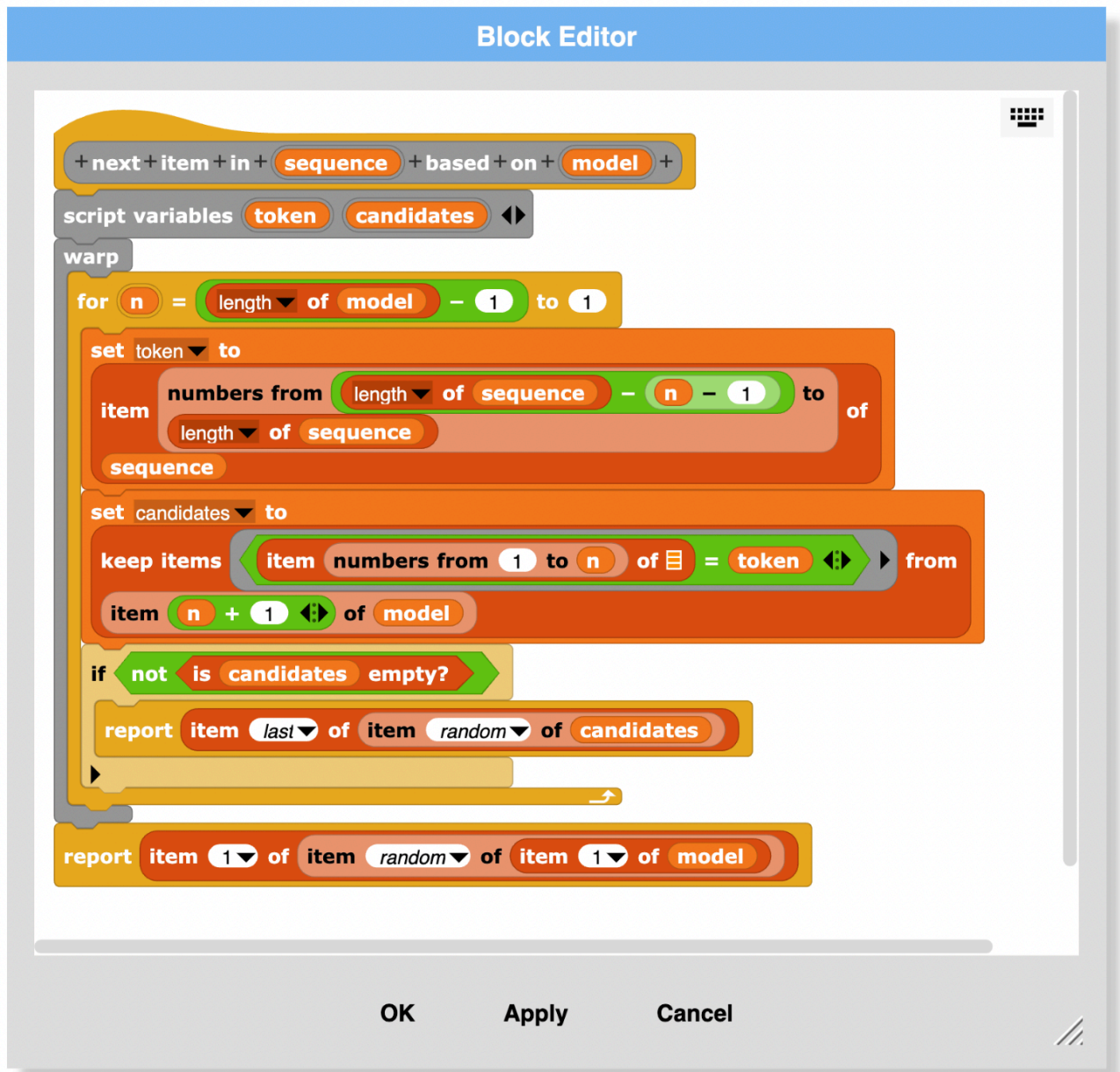


Figure 5 Example for a custom query function for the simplified Markov chain text generator.

You can test this block by providing it with a list of a few words and a data model, then click on it to execute (see Fig. 6).



Figure 6 Testing the query block with an exemplary list.

Tip: Comparing Texts and Lists

Snap! supports modeling projects related to language by allowing the testing of structural equality between lists and tables. Two or more lists can be tested for equality of all their elements, regardless of their rank (the depth of their nesting), without the need to individually handle the elements, as required in some other programming languages. Additionally, Snap! defaults to ignoring case sensitivity when comparing texts. Both features follow the example of the LOGO programming language, which originally focused on word- and sentence-based language projects.

Designing an Interactive Program

With the existing building blocks, you can now create a small program that a user can interact with. For this, I created another global variable named "story" and considered the following user guidance: When the green flag is clicked, the user is prompted to enter the first words of a new fairy tale. The program generates a word list from this input, assigns it to the "story" variable, adds the next word to the story, and outputs it as running text. Each time the user presses the spacebar, another word is added to the story, and the output is updated.

The completed program consists of four small scripts, the model script described above (Fig. 3), and an interactive text generator (see Fig. 7).

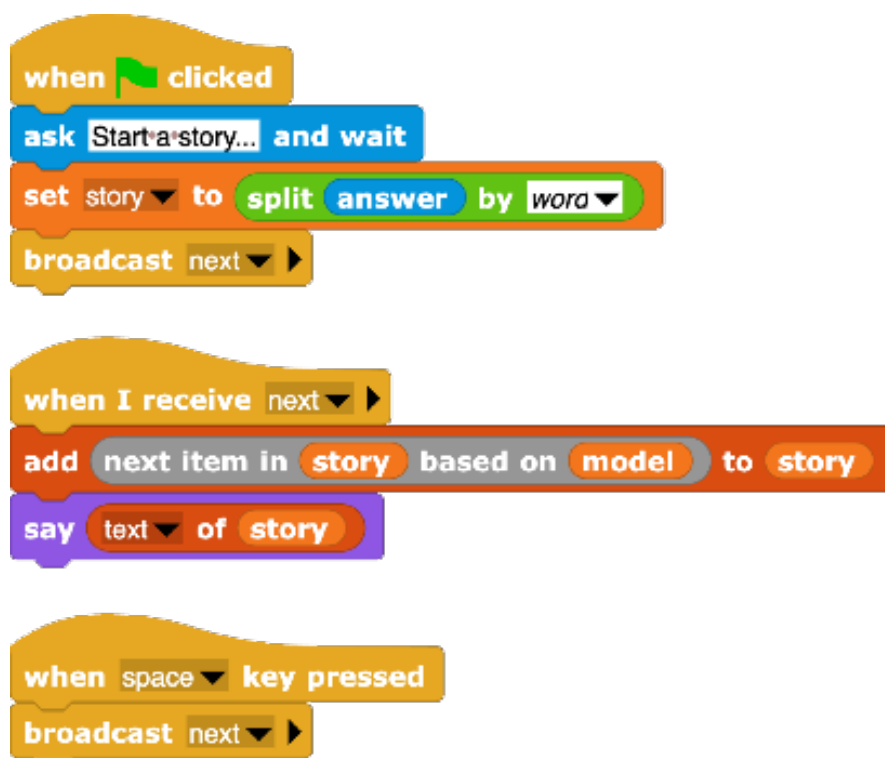


Figure 7 The finished text generator in Snap!

If you try it out and enter the start a new fairy tale when prompted, input something like "Once upon a time,". Then repeatedly press the spacebar to generate different sentences and stories. For example:

"Once upon a time a king and a queen who lived happily together and had twelve children, but they were all boys. Then said the king to his wife, if the thirteenth child which you are about to bring into the world, is a girl, the twelve boys shall die, in order that her shuttle might be stained with blood, she stuck her hand into a thorn bush and pricked her finger."

Tip: Outputting a word list as text

Because the language model is based on individual words, the value of the variable "story" is also a word list. To output this as text, all the words need to be concatenated with spaces in between. This can be done in Snap! in various ways. The simplest method is to use the "length of list" block and set the option to "text" instead of "length."

Tip: Letting the stage "speak"

In Snap!, every object has a "say" block, which allows it to output information in a speech bubble, including the stage. The speech bubble of the stage is larger than that of other objects and can display a longer excerpt of the story. You can export the content from a speech bubble by right-clicking it and select export. This will save the output as a file to your downloads folder, which is useful for sharing the generated story with someone else.

Transfer: Generalizing the Principle

Programming a text generator yourself is enjoyable and thought-provoking. Its creations are both surprisingly realistic and complete nonsense, interspersed with clearly plagiarized passages. You can play with the parameters, for example, create the model with n-gram lengths only up to 2 or 3 instead of the numbers from 1 to 5, and observe how it affects the generated texts. Instead of fairy tales, you can use other templates and then generate texts "in the style" of those different sources. This (obviously) works even with texts in different languages. In Snap!, you can import any number of files as variables and replace the variable in the script from which the language model is created. Multiple models with different contents can be managed in the same project, and you can choose based on which model the next element is determined.

An intriguing insight is that the content of the processed source data doesn't matter at all, only that the data points are in a specific order. Such sequences occur not only in texts but everywhere "lists" in the broadest sense are used. Shopping lists: People who bought this often buy that. Playlists: Someone who listens to or watches this may also be interested in that. Disease progressions: Those with these symptoms often suffer from that. Game moves: After this chess move, successful players often choose to do that. The building blocks used to break down example data into n-grams and plausibly complete incomplete sequences should, therefore, be applicable to any other context with similar patterns.

Letters Instead of Words

Instead of separating fairy tales by words, you can also split them by letters. To do this, you just need to select the "letters" option in the two "split" blocks in the model script (Fig. 3) and the green flag script (Fig. 7). Additionally, you must replace the red "text from list" block in the "When I receive 'next'" script with a green "join" block because otherwise spaces will be inserted between the letters. Now click on the model script (Fig. 3) once to create a new model based on letters - this takes a bit longer because there are many more letters than words - and then your program - slowly - generates plausible words. You can make this significantly faster by using fewer source data. Thirty fairy tales are just enough to form realistic sentences, but for plausible words, a single fairy tale or a single newspaper article is sufficient. The language model is then much smaller and quicker to search through.

Music Instead of Language

I was curious whether the principle of "guess what might come next" can be made not only visible but also audible. For this, I painstakingly transcribed 20 children's songs from "My Bonnie is over the Ocean" to "Twinkle, Twinkle Little Star" to into MIDI notes in another Snap! project. I made sure to use the same key (C major) consistently so that the melodies are more similar to each other. To play and experiment with the melodies in Snap!, you can use a loop with the "play note" block. The "play note" block has two inputs, the pitch, and the number of beats, indicating how long the note should last. Therefore, each note requires a list containing these two values. A song as a sequence of notes thus becomes a two-dimensional list or a two-column table (see Fig. 8).

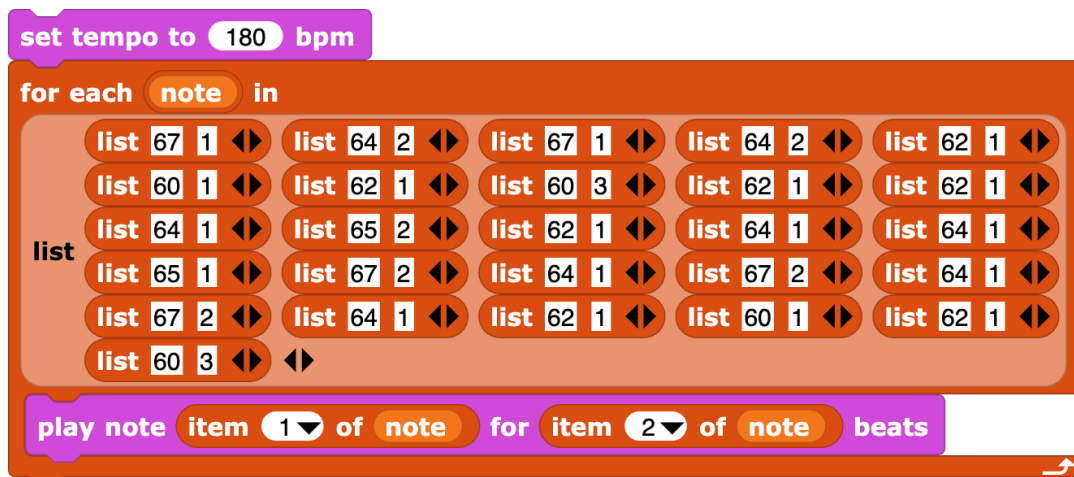


Figure 8: Playing the German children's song „Kuckuck, Kuckuck, ruft's aus dem Wald“ in Snap!

I concatenated all songs, saved them as a CSV file, and imported it into my SnapGPT project.

To create a music model from these notes, I reused the same script I used earlier to generate the language model from the fairy tales (Fig. 3). Even though the data now means something different and is in a different form (multi-dimensional), the principle remains the same. The "n-grams" function block only expects a corpus in the form of a list. It doesn't matter how many dimensions this list has.

Now, melodies can be improvised in real-time using this model. Instead of letting the user press a key for the next note, I opted for an infinite loop for this purpose (see Fig. 9).

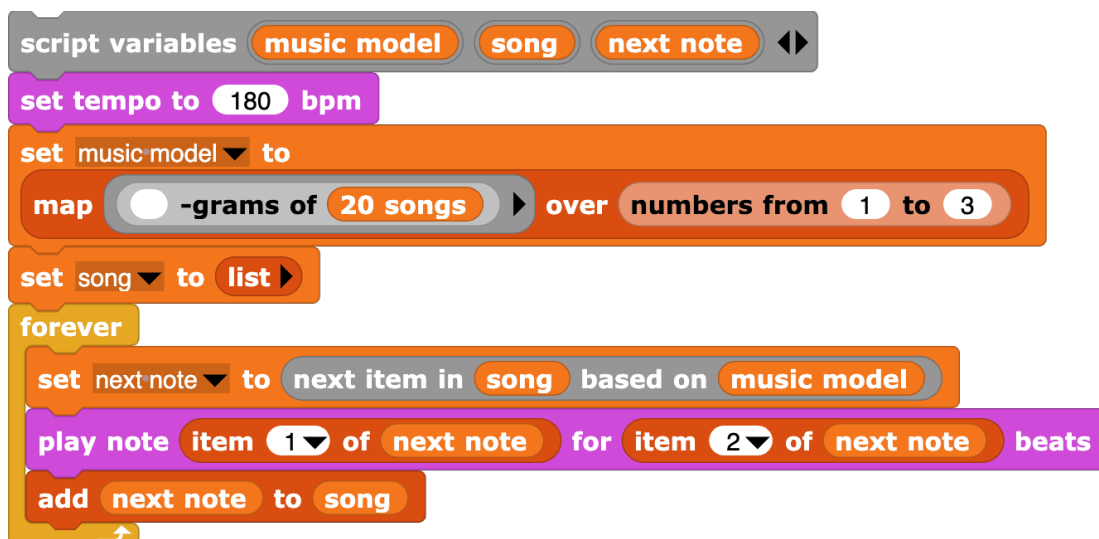


Figure 9 : The same building blocks that were used for the text generator can be used to create a melody improvisation program in Snap!

Scribbling Skylines

How about images? Do they have the “guess what’s next”? Other than words in a story or notes in a melody, elements of a picture aren’t ordered in a row but in a plane. Sketching however is sequential. The way of a pencil on a paper consists of a succession of directional decisions.

Encouraged by the melodies that were created by throwing together chopped up children’s songs, I created a sketching program that records the changes in directions in regular intervals.

To keep it simple, I focused on the data of a single stroke, i.e. from putting the pen down to putting the pen up again. This results in a list of numbers, a direction for each line segment. This list of directions can then be used similarly to the fairy tales and children’s songs to create a data model of n-grams (Fig. 10).

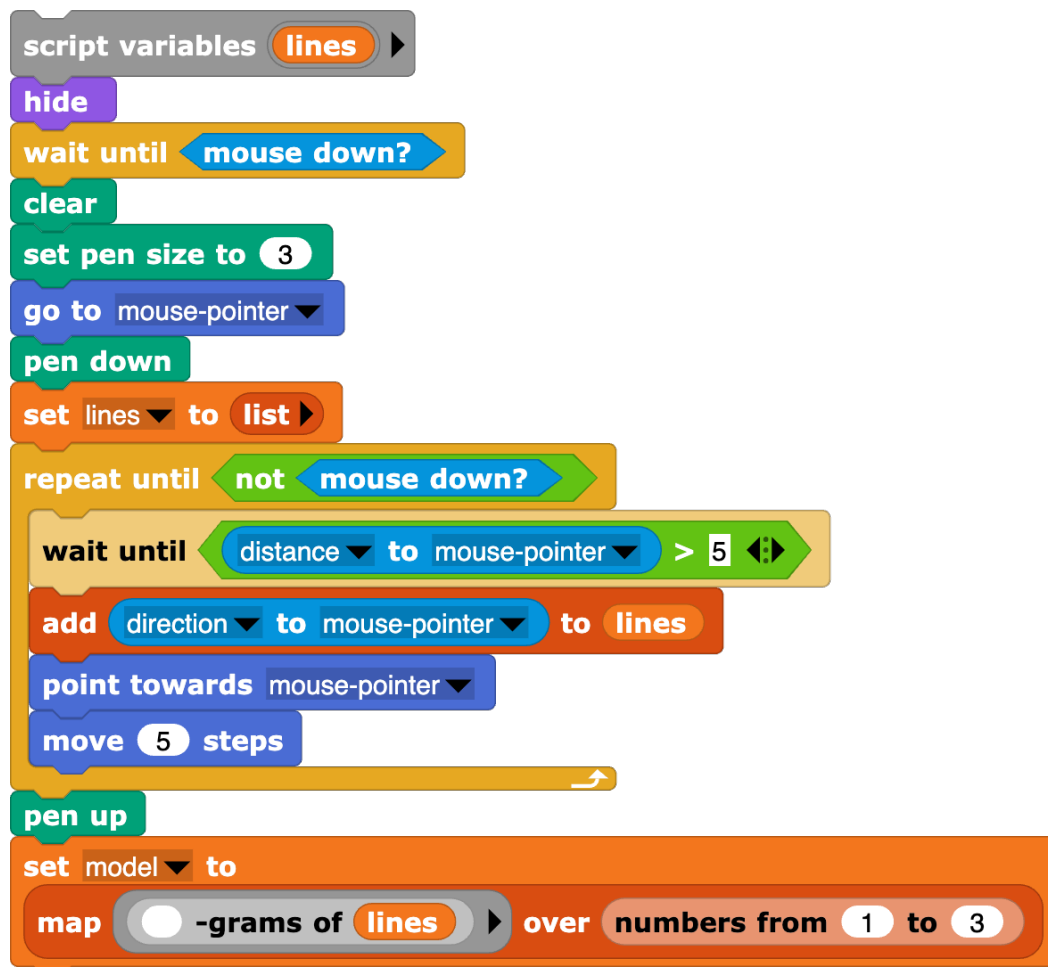


Figure 10: Example for a sketching program that creates a sketch from a list of changes of direction and calculates a data model from that.

You can click on the script to start scribbling something on the stage using a mouse or track pad. You could try writing a word in cursive handwriting (Fig. 11).



Figure 11: The sketching program can only create sketches that consist of a single stroke, e.g. a word in cursive writing

Subsequently, the model fantasizes new doodles based on the initial sketch, if – once more – the query block that suggests a next plausible element in an arbitrary sequence is used (Fig. 12 and Fig. 13).

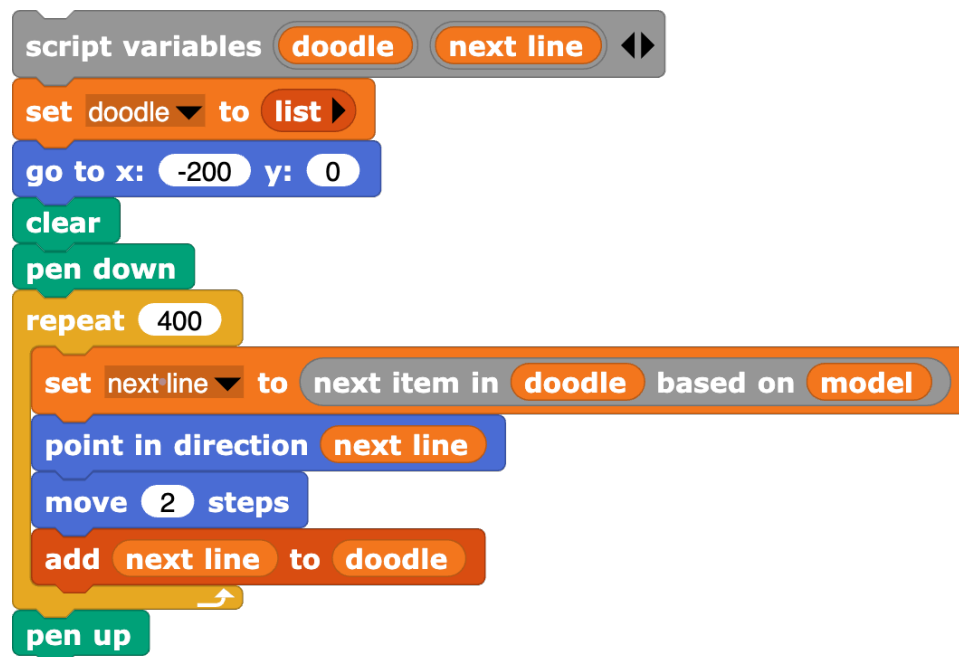


Figure 12: Snap! can use the same two blocks to fantasize stories, melodies and doodles.



Figure 13: Exemplary doodle based on the written "hallo" in Fig. 11

Only being able to draw single strokes limits the abilities of the project to, e.g. when imitating your own handwriting.

While we played with the scribbling program, my friend Bernat Romagosa brought up the idea to try it with waves and angular shapes instead of letters and circles. That way, the program creates more “organized” structures that remind of a horizon at the sea or a skyline of a city (Fig. 14).

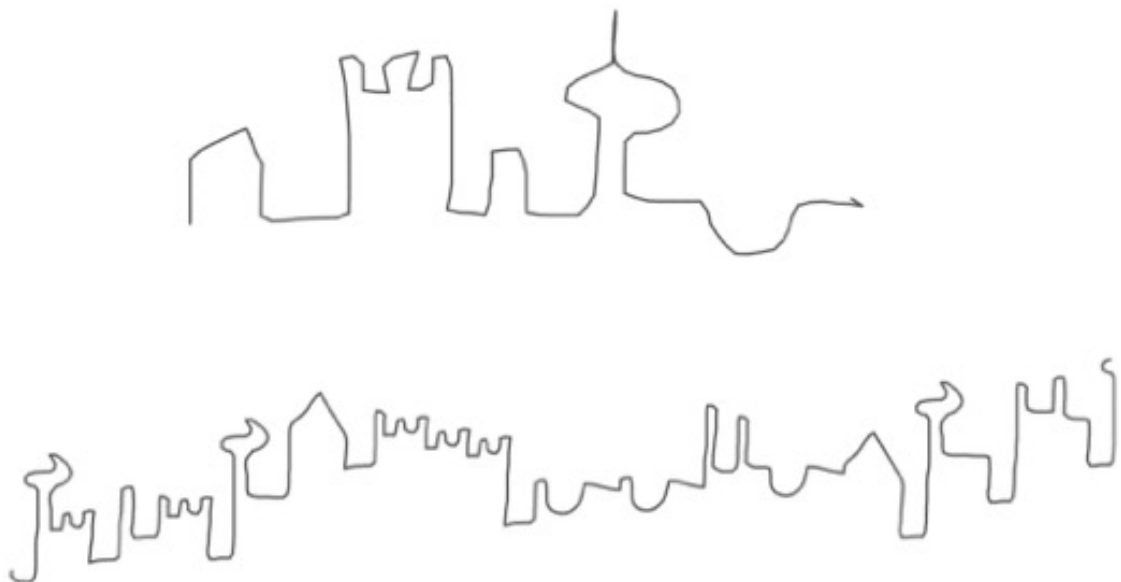


Figure 14 Example for a “skyline”, the top sketch is the manually created template, the bottom sketch is the version generated by Snap!.

Conclusion

The completed Snap! project can be found at <https://tinyurl.com/SnapGPT-IBIS-EN>. It contains the custom Snap! blocks needed for the project as well as the collection of 30 fairy tales and 20 children's songs. The data can be exported in csv format from the context menu of the variable watcher on the stage and provided to the learners.

Programming something like this GPT in computer science classes is certainly within the realm of possibility. The project can be implemented in various ways, and conveys skills that are useful in other contexts, such as data transformation and filtering. I personally appreciate that the metaphor of "building blocks" plays a role in multiple aspects: the two blocks for the text generator can be directly applied to improvising a melody or scribbling a silhouette without having to modify them. For underlying idea of the n-grams, it doesn't matter what data they represent.

Many thanks to Michael Hielscher for the brilliant idea of introducing the challenging topic of generative AI pedagogically using a Markov chain text generator, for his beautiful SoekiaGPT software, and for his guidance and inspiration in the playful exploration presented here in Snap!.

Sources

All websites/links were last checked on December 2, 2023.

Hielscher, Michael, SoekiaGPT – ein didaktisches Sprachmodell (2023), IBIS 01-01-04

Snap! – Build Your Own Blocks (2023), <https://snap.berkeley.edu>

The Beauty and Joy of Computing (2023), <https://bjc.berkeley.edu>

Rosetta Code: Markov chain text generator, draft (2023), https://rosettacode.org/wiki/Markov_chain_text_generator

License

This article is available under the CC BY NC 4.0 license.

Contact

Jens Mönig

SAP Walldorf

jens.moenig@sap.com