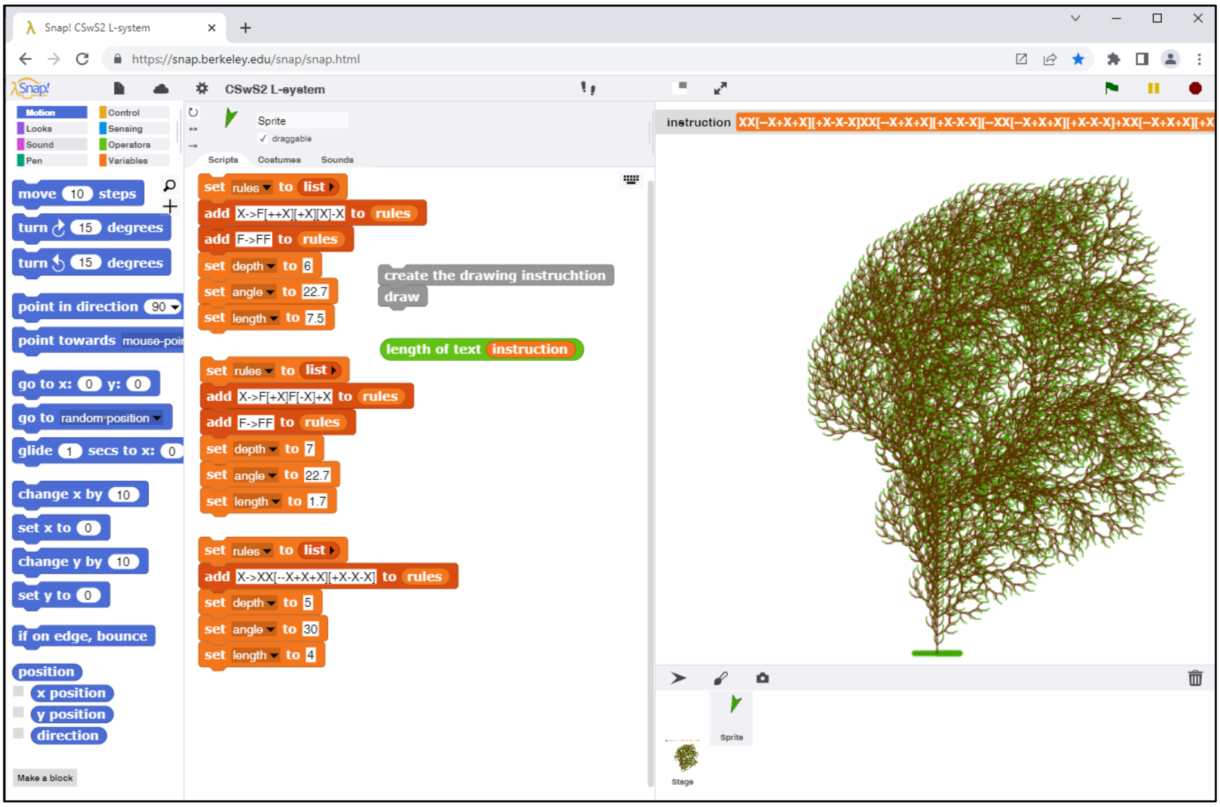
Eckart Modrow

Computer Science with Snap!

– *Snap!* by Examples –

Version 2



emodrow@informatik.uni-goettingen.de

http://emu-online.de/projectsOfCSwithSnap2.zip

The scripts are developed with Snap! 8.0.0.

Prof. Dr. Modrow, Eckart:

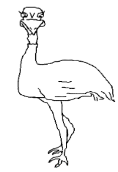
Computer Science with *Snap!*

Version 2

\- *Snap!* by Examples -

If this book is helpful for you and you would like to express your appreciation in form of a donation, you can do so at the following PayPal account:

emodrow@emu-online.de
Intended use: Snap! book



Preface

This book, similar to its predecessor "*Computer Science with Snap!*"[1], uses a collection of programming examples to explore the scope of the graphical language *Snap!*. It does not replace a textbook that conveys CS content but shows how to use *Snap!* to apply CS methods. In this second version, some reflections on computer science education, especially on the concept of objects and the relationship between information, data, and visualization, are prefaced. Examples explaining their consequences can be found below.

Snap! in the present version 8.0.0 represents the next step in the development of graphical tools. The current version was extended among other things by features in the area of object-oriented programming (OOP), list operations and multiple stages as well as metaprogramming and thus meets all requirements up to high school graduation and far beyond. Since also drastic improvements were reached with the working speed and libraries for different ranges, e.g. with the pixel access, in the audio range or with the use of external resources are available or can be developed easily new, hardly restrictions in the application areas exist. If it must be, one can still use JavaScript functions for time-critical operations or extensions within *Snap!*. The libraries contain numerous examples of this.[2]

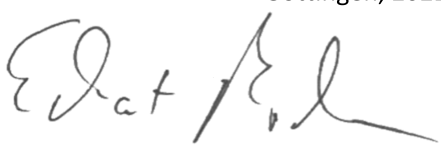
The selection of problems in the following chapters is relatively conservative, in some cases leaning closely on existing computer science curricula, but also going beyond these. This is intended. I hope on the one hand to "pick up" the teaching colleagues from traditional courses, and on the other hand to provide contexts that give meaning to the computer science content to be acquired from the learners' point of view. This way should result in lessons that are very much oriented towards creativity, but also towards the teaching of informatics concepts. The examples describe in detail the handling of *Snap!* from different aspects. After a few considerations about didactics in this area, an introductory chapter follows, which explains how to work with *Snap!* "on the fly". Then the next chapters illustrate the possibilities of the language. Sections without direct application reference also follow. This compromise is due to space requirements because extended concepts actually require extended problems. The examples are not arranged hierarchically, even the second part contains rather simple ones. At the end of the script there are overviews of the methods used in the examples as well as an index.

This book is a translation from German. Unfortunately, I do not speak English well, so it will be bumpy. I apologize for that. Because all programs had to be changed as well, this task could only be done by me. Be strong and hold it! Many thanks for the wonderful help of the *DeepL*[3] translation program. I would probably never have finished without these.

I would like to thank Jens Mönig for his support - and for the results of his work. The learners will be thankful!

I wish you a lot of fun working with *Snap!.*

Göttingen, 2022/9/15



[1] E. Modrow, Informatik mit Snap, https://emu-online.de/ComputerScienceWithSnap.pdf

[2] *SciSnap!2* is discussed in more detail in https://emu-online.de/ProgrammingWithSciSnap.pdf

[3] https://www.deepl.com/translator

Content

1 Didactical Remarks

1.1 Data, Information, Stories, and Visualizations[4]

Modeling and *implementing* as well as *reasoning* and *evaluating* belong to the core of the process-related competencies of school computer science. For teaching, their relationship is crucial: on the one hand, learners should independently create solutions to problems, for which they acquire technical knowledge and, of course, also need training in the use of tools; on the other hand, the subject matter should enable discourse on social and political issues based on the acquired technical competence. The relationship between the three areas of *tool use*, *technical issues* and *social impact* determines the framework for general education. Or to put it more sharply:

> *How much time should be spent on tool training, i.e. learning how to use the programming language and its development framework, so that there is enough time for the students to solve problems independently and to reflect on the results?*

Without this time, the subject actually has no place in general education schools. In the following, we will examine in a little more detail the *information-oriented didactics of computer science* prevalent in German-speaking countries, the terminology used therein, and the implications for the choice of tool and its use.

The German Society for Computer Science (GI) writes on the above competencies:

"The process of modeling is not only learning content, but also a consistent method of computer science teaching, although implementation is also indispensable to make the result of modeling tangible. Reasoning and evaluation promote the learner's ability to communicate and to argue; without this area, dealing with computer science systems is only intuitive or playful and often determined by influences from media."[5]

The GI mentions as contents for the middle school the *connection between information and* data, *different forms of representation* and *operations on data* and their *interpretation* in relation to the represented information. In the upper secondary school[6], a distinction is to be made between *characters*, *data* and *information* as well as between *syntax* and *semantics* and information is to be represented as data with data types and in data structures. The current curricula largely adopt these specifications.

In addition to the contents, the sample tasks are particularly interesting for teachers, because from them an idea of the intended teaching can be gained well. In the area considered, there are traditionally treated topics from the field of data structures and databases, but almost nothing about information. This term appears mostly only within word combinations (*information technology*, *information society*, ...), and it is used contradictorily. If, for example, information is defined as "*the semantics of a statement, description, instruction, communication, or message*" [7], then it is not quite clear to me how these semantics are to be

[4] largely from Modrow, E., (2017). Ist der Informationsbegriff für die Schulinformatik hilfreich? LOG IN: Vol. 37, No. 1. Berlin: LOG IN Verlag. (S. 38-43).

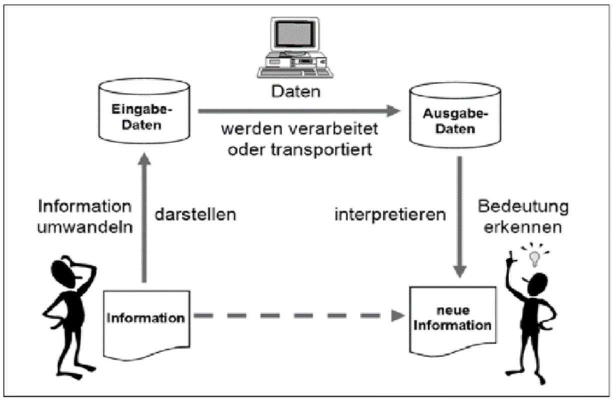
[5] Attempt of a corresponding translation from German by me. The same is true for the following translations.

[6] may be high school

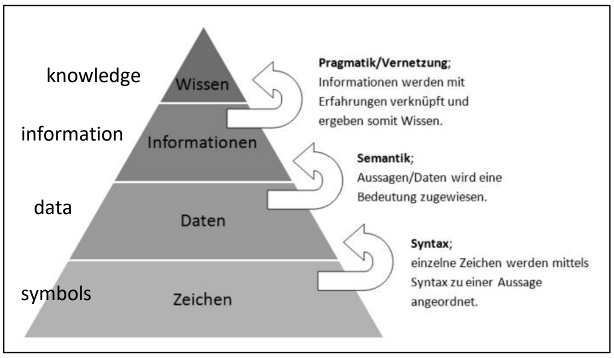
[7] https://kultusministerium.hessen.de/schule/kerncurricula/gymnasiale-oberstufe/informatik

"*processed automatically by machines*"[8]. It seems that teaching cannot be easily derived from a term that is not sharply enough defined, even if it is used prominently in the competency domains. So, it is worthwhile to look a little deeper into the meaning of information.

In information-centered computer science didactics, the concept of information is usually explained using the diagram on the right[9]. If one derives content areas from it, one comes very fast e.g. to the automatic processing and linking of *representations*, thus to data. Information-centered didactics just as quickly turns into data-centered didactics when it comes to concrete teaching. From the diagram it becomes clear that the concept of information used in computer science didactics has neither to do much with Shannon's information theory nor with the everyday equivalence of information and data. The level of information is hardly linked with computer science content, so that an implementation in teaching is difficult or requires breaks in content.



We therefore need precise and mutually compatible definitions for the terms used. The *knowledge pyramid*[10] seems to me to be helpful for this, which, in addition to *data* and *information*, also contains the levels of *knowledge* and *symbols*. As a starting point we choose the definition of knowledge from Wikipedia[11]:



> ***Knowledge** is [...] understood as a collection of facts, theories, and rules available to persons or groups, which are characterized by the greatest possible degree of certainty, so that their validity or truth is assumed.*

Knowledge is thus bound to persons and consequently cannot exist within today's machines. There we find data. Since knowledge cannot be complete and can even be wrong, gaps in certainty arise which can be closed or reduced by information[12].

> ***Information** is the subset of knowledge needed by a particular person or group in a specific situation and is often not explicitly available.*

This definition is similar to that from the GI Education Standards, "*Information is the contextual meaning of a statement, description, instruction, communication, or message.*", but related to the knowledge modified by the information. Information is also tied to individuals who recognize and evaluate the meaning of the data. It is time and situation dependent. If a person receives a message twice, for example, the information content is much smaller the second time, because the knowledge gap was already closed by the first information. Information depends on the one hand on the data used for its transmission, but on the other hand

[8] http://www.schulentwicklung.nrw.de/lehrplaene/upload/klp_SII/if/KLP_GOSt_Informatik.pdf
[9] http://www.informatikstandards.de/index.htm
[10] https://derwirtschaftsinformatiker.de/2012/09/12/it-management/wissenspyramide-wiki/
[11] https://de.wikipedia.org/wiki/Wissen
[12] https://de.wikipedia.org/wiki/Information

it also depends on the state of the receiver. The receiver pragmatically integrates information into his existing knowledge, links it with it - or not. Up to this point, there are no problems: Information is in the head, data in the computer. The concept of information has no place on the machine level, to which we now turn.

> ***Data** is represented by **symbols** of the selected character set, which we can understand here as code. The **syntax** of this representation describes the structure of this representation.*

The above-mentioned concept of information is person related. Information can therefore not be seen without the interpreting person, e.g. because the same data can represent completely different information for different persons. Without their **context**, data lose the property of being information. They are reduced to what they are without meaning: just data. In the knowledge pyramid model, the relationships are clear: the receiver interprets the received data and tries to make sense of its semantics. This step occurs before the linkage with his existing knowledge and largely independent of it. The interpretation depends on the receiver and its state; it cannot be done solely based on the data. After the interpretation, the receiver decides whether the meaning of the data represents information for him.

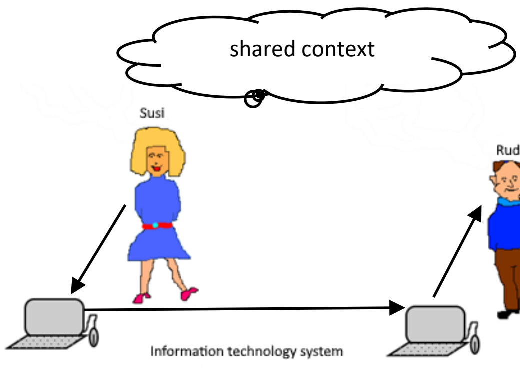
In my opinion, we should refrain from squeezing the concept of data into the information-centered scheme as was done above. Data is a category in itself, bound to a physical representation. If, for example, an ocean sonde measures temperatures, stores them, and then is lost, then the physically represented measurements exist as data, even if, unfortunately, they never become information. If an operating system stores system status in log files, then these data exist, even if they are never evaluated by humans. The definition of data as representations of information confuses the above concept of information with the colloquial one and leads to the unattractive situation that the meaning of information seems to be satisfied when data and their structures are considered. But this is not true.

Our investigation has a simple result: the two lowest levels of the knowledge pyramid are accessible to computer science. They are linked to the traditional content areas. The two upper ones are at least partially intrapersonal, going beyond pure computer science. Like the area "computer science and society" they refer to the meaning of computer science systems, this time not so much politically and socially, but related to personal concern. The concept of information is part of the general educational contribution of school computer science. This is not achieved if the treatment of data-related topics is equated with information-related ones.

We want to work out the consequences of our considerations in four situations. For this we name the two actors of the information transfer scheme shown above as *Susi* (sender) and *Rudi* (receiver) and reduce the labeling in the scheme.

<u>Case 1</u>: Susi sends the message "*Have arrived!*" to Rudi.

The message can only have meaning for Rudi if Susi and he are clear about its sense. Thus, if Rudi knows that Susi is either on her way to Hanover or to herself, then he can interpret the message, even including subtexts such as the missing "well" that might suggest some complications. Susi, on the other hand, knows that Rudi is waiting for her message and will understand it in its brevity. She can express her information through appropriate data. Susi and Rudi act within a shared context that allows them to interpret the message. Without this context this is not possi-



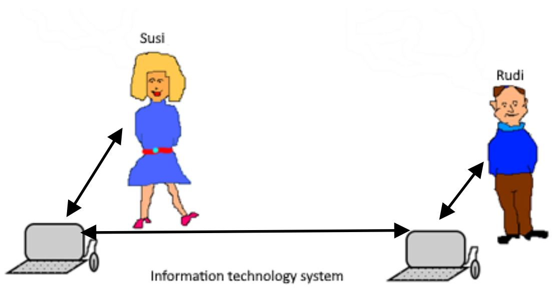
Communication in a given context

ble, and therefore the context should also be included in the schema. However, it should not remain there, because of course the classroom consequences are relevant, not the schematic ones. In the classroom, such a context can be well realized through **stories**, as we are doing right now. Thus, not only suitable data structures and protocols result from a problem, but also the **visualization** of the situations, the connection of the technical topics to the actors of the story, may it be the inhabitants of a farm, the story of the relationship between Susi and Rudi, or the elements of a simulation. It should also be possible to manipulate the data occurring in it, in order to be able to observe what is happening without effort and to control the results. What one sees usually does not need to be explained separately. Both, the visualization of the context and the data, should be easily possible in a development environment suitable for schools.

In this first case, the role of the computer system is completely secondary, clearly separated from the exchange of information. Susi could also have called out loud, sent a postcard, drummed the message or had it transported by carrier pigeon. And vice versa, the ability of the computer system to encode texts appropriately, to transport the characters and to represent them again is completely independent of the information transport. The task of the system is to mark the characters in such a way that they can be recognized as text and represented by a suitable subsystem. This task is performed automatically, e.g. by marking the data packets or the file on the basis of the established syntax. This has nothing to do with understanding.

All in all, the example suggested by the basic scheme is unproductive from an informatics point of view. In my opinion, it should only be used if the information aspect is to be particularly emphasized in the lessons.

<u>Case 2</u>: Susi sends the message "*mostly in the afternoon*" to Rudi.



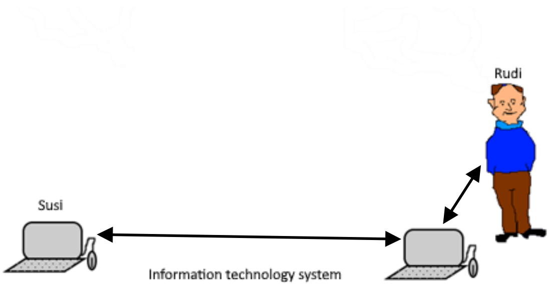
Communication with an open question

In this case, the common context, steeled in many crises, is not supposed to be present because Susi and Rudi are more or less random communication partners in the network. Since Susi cannot arrange and transmit appropriate data without this context, Rudi must first establish the context. Therefore, the communication process has to be started by him by asking an appropriate question to Susi. The question is interpreted by Susi in such a way that she can identify the desired information and convert it into data. In turn, Rudi must interpret the received data as an answer to his question and evaluate it as the information he is looking for. This is represented in the schema by double arrows. A lot can go wrong on both sides. Susi can misunderstand the question if it is not formulated completely clearly. So, she can receive wrong information from Rudi and generate correspondingly wrong answers, which can be misunderstood by Rudi again.

Again, the interesting things happen in the minds of the participants. We could discuss the importance of non-verbal communication and address the role of emoticons, examine text comprehension in different social or cultural contexts, or the need for video telephony. All of these are important school topics worthy of discussion. What they have in common, however, is that we do not approach them through knowledge of either the network protocols or the data structures used. The specialized informatics topics are irrelevant to the role of information discussed here.

Case 3: Susi sends the message *„Berlin, Bern, Bucharest“* to Rudi.

In this case, Rudi has asked his question so precisely that Susi can evaluate it unambiguously. An interpretation and thus a context suitable for understanding is not necessary. But this also eliminates the role of Susi as a person. She can be replaced by a computer that answers the question as long as some syntax rules are respected. Rudi can ask e.g.:



Communication with a clear question

SELECT name FROM cities WHERE istCapital = „yes“ AND name like „B%“ LIMIT 3;

The information is distributed very one-sidedly in this case. Rudi knows what information he needs. He describes the data required to close the knowledge gap and retrieves it from an information system. Neither on the way from Rudi to the system nor within the system there is even a hint of information. This arises only in Rudi's head after he has received Susi's answer.

Since this third case corresponds very directly to communication in and with information systems, its analysis is important for learners. Whether they use digital assistants, consult databases, or use search engines, they are expected to have an unambiguous description of the data needed to answer questions - whether it suits them or not. While the systems may reflect understanding, or it may be attributed to them by the users, they do not possess it. Awareness of this prevents overestimation of the answers received and underestimation of the user's responsibility for his or her question. The more the role of the communication partners is blurred, the less clear the evaluation of the results becomes.

Case 4: Rudi transfers his tasks to a program and goes swimming.

Data exchange without human partners

After Susi has already been replaced by an algorithm, in this case by an SQL server, Rudi could also get the idea that his tasks can be performed better and faster by an algorithm. He claims that he can describe his interpretation of Susi's data sufficiently precisely by a program that extracts information from the data and also immediately initiates any necessary actions. Is this true? We choose high frequency trading in the banking system as an example. Susi transmits the current prices at her stock exchange, Rudi evaluates the differences to his stock exchange and initiates corresponding buy or sell instructions.

Since Rudi, now as a machine, has no knowledge, also no knowledge gaps can be closed with him. Therefore, it cannot be information in the defined sense. The algorithm Rudi has indeed emerged from the knowledge of the person Rudi about the processes in stock exchange trading, but it does not represent this knowledge completely, and above all, it does not link it with Rudi's remaining knowledge. The gaps in this knowledge, which must be closed for concrete reactions in stock exchange trading, require the current stock exchange values. For this purpose, the algorithm has variables, i.e. blanks, which are updated by Susi. Depending on these values, Rudi runs through its sequences of instructions in different order and triggers the corresponding actions. No interpretation is required for this. It is a pure automation process.

We can learn a few things from the four cases considered. The first two show that human communication can be problematic, regardless of the medium used. The latter gets its meaning from the fact that it makes communication possible and from its distribution. The concept of information is irrelevant to the technical issues of data processing that arise in the process.

The other two cases are more interesting. The third describes quite well the roles of the user and the IT system in information retrieval. The intelligence here lies entirely with the user. The user describes the data required to generate the information sought and is thus also responsible for this description. If the description is imprecise, then he receives corresponding answers. If, as in case 2, the questioner asks a human expert, the expert must infer the information sought from the context and, with the help of the IT system, collect and transmit the data required to answer the question. He or she then also assumes responsibility for its relevance. In case 3, the demands on the questioner increase considerably, because he must now be an expert. There are no more excuses. His question is always evaluated, e.g., via statistical correlations or by searching for matches to the question text verbatim in the network, but it is not understood. In order to be able to sort the resulting data at all, the system must supplement the missing context, e.g., by evaluating past questions or similar questions from others. The danger that this creates "echo chambers," for example, which always generate data with the same tendency, is discussed as a current problem that endangers democracy.

In this scenario, the information aspect leads to the question of what the user needs to know in order to be able to ask appropriate questions; to know both about the subject of the question and about how the used system works. The traditional subjects of school computer science are thus extended by an aspect that is suitable for evaluating the relevance of these very subjects against the background of life in a society shaped by computer science systems. Information-centered didactics understood in this way requires the development of new teaching components to further develop the subject in the direction of current general education. It links the subject content with its social significance. To be able to do this with reasonable effort, it requires tools that, on the one hand, keep the time required for tool training small, i.e., free up time for other things, and, on the other hand, give space to the context in the form of stories in addition to an appropriate consideration of the specialized topics.

The fourth case describes the transfer of human tasks to information technology systems. People can describe from their knowledge and experience how to react in different situations. Machine learning methods then transfer descriptions of this knowledge into suitable (data) structures. Within the framework of these structures, the automated systems react in a manner comparable to humans, usually even faster and more reliably. But what happens if the description is incomplete or new situations arise? Since the evaluated data retain their data character throughout the entire process, i.e., they never become information, their semantics are also never made accessible. If they mean something different than actually intended, then no one understands this change in meaning because it cannot be linked to existing knowledge from perhaps completely different areas. (By the way: the use of neural networks does not change this assessment). In these cases, the clear separation of data and information makes it possible, for example, to discuss responsibility for the consequences of automation (for example, in autonomous driving) and to explore the ethical boundaries (for example, in the selection of training data). The information aspect creates clarity in argumentation and prevents socially relevant issues from being clouded by a retreat to technical content. It enables political discourse on the position of informatics systems.

Information-centered didactics has led to a somewhat inflationary use of the term information in almost all areas of school computer science, at least in German-speaking countries. It loses its sharpness and especially its function to give orientation in the planning of lessons. The traditional content areas, such as data and data structures, are not harmed by the fact that they now have the additional claim of also taking the information aspect into account. But it reduces the chance to accentuate teaching of computer science in the direction of its general educational function. If, on the other hand, we reduce the concept of information to its original meaning, then we expand the subject canon of school computer science to include socially relevant aspects that can have a direct impact on the planning of the curricula.

As an example, let's look at the concept of the "knowledge society", from which it is sometimes concluded that knowledge no longer needs to be acquired if it is available to everyone "on the Net". On the basis of our considerations, we can immediately see that it is not that simple after all. In the net we do not find knowledge, but data. Instead, there are a number of questions that need to be clarified before information acquisition in the knowledge society can really work out:

- What basic framework of knowledge do learners need to have in order to be able to identify their knowledge gaps at all?
- What competencies do the learners need to acquire in order to be able to accurately describe the data required to close the knowledge gaps? Can they even describe what they don't know?
- What knowledge about the informatics systems providing the data do learners need to acquire?
- How do learners learn to assess the relevance of the data provided relative to their question?
- What happens if the answers are "colored", e.g. adjusted to the questioners?
- What data does the answering IT system obtain from the questions? What information can be derived from it?

The concept of information thus proves to be quite clearly effective in the area of "computer science and society". We should leave it there. In my opinion, this restriction does not limit its importance. On the contrary: if a term can clearly accentuate the orientation of a school subject, that is not little. It is a lot.

If the concept of information is not very productive with respect to data, but helpful with respect to the area of computer science and society, the meaning of the content area "data" must follow from itself - otherwise it will be difficult to justify the existence of this area from a general education point of view. Among the mentioned content areas of this domain, besides the somewhat interspersed concept of information, the classical topics of a standard area of computer science can be found: the area of *algorithms and data structures*. It seems to me that computer science structures its contents differently than current computer science didactics for a good reason: obviously there are not too many points of contact between data and information, but there can be no separation between algorithms and data structures, because the other area is indispensable for the one. This also becomes clear in the GI demand for "*modeling and implementation as a continuous method*". Even if the scientific structuring of the content areas is not a mandatory requirement for didactics, it should be taken into account, because it is certainly not senseless.

The question is somewhat different: *Which parts of the basic scientific curriculum are relevant for general education-oriented didactics at a certain point in time?* Or still differently: *If "in former times" certain technical questions were also important for schools, because the technical and professional development had only reached a certain state at that time, then this does not yet result in a compelling justification for the relevance of these technical topics at a later time.* Linear data structures (stack, queue, ...) may serve as an example: In universities they are still relevant and there closely related to the algorithms working on them. In school they had their importance, because without their implementation advanced student work could hardly be realized. With the tools available today, however, it must be asked whether the implementation of these structures is still necessary. If lists are available that can be visualized well, then the linear structures actually only differ in the place of their access, the beginning or end of the list - and this can be seen. It is intuitively clear what causes which operation.

Now, in the area of schools, it hardly makes sense to consider the processing of data as a purpose in itself. What is required is again a context from which the need for its transformation arises. Data thus acquire a meaning; their processing takes place for a specific purpose. Without this context, the acquisition of competencies from the area of "reasoning and evaluating", which is central to the justification of the school subject computer science, is also hardly realizable. The context is therefore to be taken seriously. It is of equal importance to the subject. Pseudo-contexts, which only serve to get to the subject content as quickly as possible, are rather counterproductive. If the context is obviously meaningless, then this "nonsense" is easily transferred to the subject, which consequently also appears meaningless to the learners.

Data originates from the context and flows back into the context in a modified form. *Physical computing* may serve as a prime example, where sensor values are collected by the computer system and used to generate actuator control data. ("*When it rains, the windows are closed.*" "*When the train comes, the barrier is better down.*") The example also shows that simple numerical values can have their meaning as data. However, they do not have this meaning "per se" but gain it within the given framework. The example also shows that the learners do not necessarily have to solve tasks that the teacher has set but can work on problems that they themselves have derived from the context. They do not work on exercises, but act as problem solvers, in this case as small constructors who make life easier for other people or prevent catastrophes. The transformation of data is not an end in itself, but a means on the way to a goal they have set themselves.

The context does not necessarily have to be real (as in *physical computing*) or simulated (through the multimedia properties of visual programming languages). It can also be a story from which the informatics question arises. Computing a certain percentage according to a given procedure need not motivate everyone. But if one asks the question about the contribution of e.g. Germany to the damages of a hurricane in a completely different place[13], then a single number gets an immense meaning, even if we can determine it only rudimentarily. Even the recognition of a digit in a picture becomes interesting for learners if, for example, the problem of recognizing car license plates arises from an exciting story or a current case. No matter how the context is chosen: its importance for the motivation of the learners requires that the development environment can take it into account, through graphics, sounds, animations. To ensure that this representation does not displace the actual subject content, the context representation must be very easy to manage.

Structured data of the same type usually occur in direct form as strings or images. Therefore, there are separate data types for them. Linear data sets occur either as sequences of input/output values (*data streams*) or as character strings which are transformed for certain purposes (*cryptography*, ...). Both possibilities show that the embedding in a meaning-giving context follows as if by itself. In the case of images, the required transformation usually follows directly from the problem definition. *Image enhancement*, *color changes*, *edge detection* and consequently *object detection*, *classification of images*, etc. may serve as examples. Since the representation of these data sets as list-like structures or tables is intuitively clear, their algorithmic treatment is usually not a big problem. The situation is somewhat different with the control of the developed algorithms. Since these structures can contain a lot of data, the ease of visualizing them, from which the current state of the data set can be seen, is crucial for learners. In schools, therefore, it is not so much the algorithmic components (which are always present) that matter, but the visualizability of their effects.

[13] Friedericke Otto, World Weather Attribution, https://wwa.climatecentral.org/

Direct "data processing" no longer plays a major role in schools because special tools such as database systems have taken over the partial tasks. The data are therefore largely elements of models in which they describe the parts of the systems and represent interrelationships. Together with the demand for a context that seems meaningful to the learners, it follows that the subject area "data" should be predominantly embedded in a subject area "modeling".

1.2 Computer Science and Media Education

In schools and universities, the teaching of media competence is being hotly debated as part of the " digitalization offensive". Since the term " digitalization" obviously concerns computer science, the latter should take part in the discussion. Teaching institutions need to think carefully about what exactly their contribution to overall education is. On the one hand, children and young people gain knowledge and experience also - and in many areas predominantly - outside these institutions; on the other hand, the goals of "education" and "training" should be sharply distinguished. Young people do not need to master the use of current professional tools; they can safely leave that to adults. But they must be prepared to take over their role with future tools.

It is and has often been argued that learners need to learn how to use modern media in order to lose their "fear of them". I think this is absurd. Firstly, children and young people are normally not afraid of media, they are curious about them. Secondly, they learn how to use them quickly and easily from others and through use. The fear is more on the part of the older ones, who have not grown up with this technology and therefore feel insecure about it. Those who are currently older should remember that in their youth, those who were older at the time discussed how they could be gently introduced to mouse-controlled interfaces in order to take away their fear of them. We can learn from this that handling current technology, such as smartphones, is learned along the way, but that this obviously does not automatically lead to using future technology in the same uncomplicated way.

> *Conclusion: Learners must be enabled to understand the fundamentals of future technologies and to acquire the skills to use them. For this, they need general knowledge of the technical fundamentals of information technologies, but not specialized knowledge of the current technology.*

It goes without saying that media use is not the same as media consumption. The passive use of media of whatever kind, e.g., simply "gawking," cannot be the goal of the educational system. When we deal with media, they must occur in a context that activates learners.

> *Conclusion: The learners must be enabled to select and use tools, e.g. for the creation of media, depending on the problem. To do this, they must learn to solve problems independently.*

Education for independent problem-solving is usually not seen as a central task, at least in schools. Creative subjects such as art, music and sometimes languages at least sometimes strive for this. Mostly, however, the focus is on good learning. Computer science now provides tools that can be used to realize, test, and improve one's own ideas even in a relatively rudimentary form. It would be a missed opportunity if the subject did not realize creative teaching for the learners. However, this will only work if the teachers themselves have experience in independent, creative problem solving and if they trust the learners to do so. If the teachers have only " well learned" the informatic contents, then it will not work out with the creativity in the lessons. If independent problem solving is to be aimed at in schools, then this should and must also have consequences for teacher training at universities.

> *Conclusion: Teachers must be enabled to plan and implement creative lessons. Opportunity and space must be given for this in their own training.*

Modern media such as social networks have changed social life, communication, etc., in some cases profoundly. The consequences can hardly be foreseen while this process is still ongoing. Much less were they foreseeable before it was started. I would therefore consider it a complete overload for teachers to demand that they deal with the actual social consequences of IT systems, which include the effects of digital media, in the classroom. That would also not be effective, because looking at the consequences that have already occurred is necessarily backward-looking. What can be demanded, however, is to show that the use of information systems has social consequences and that these depend very much on how the systems are designed. Different problem solutions therefore have different consequences - and vice versa: If certain consequences are undesirable, then it will usually be possible to find another technical problem solution.

> *Conclusion: The learners must experience that there are almost always different solutions to a given problem. They should think about their effects, which are of course not conclusive. They learn that these effects are not given but can be shaped.*

What does this have to do with *Snap!*

Graphical programming environments like *Snap!* not only contain the algorithmic components but are embedded in a media environment that not only allows, but requires the use of graphics, sound, ... If a problem is being worked on, then cameras and graphics programs can and should be used to create the appropriate costumes and allow costume changes that visualize the current state of the system. Sound programs allow to comment on the process itself, to edit and insert music or to design it by oneself. And, of course, the results must be presented, because product pride is an important motive for dedicated work and interest in the results of others is great. *Snap!* supports just the presentation aspect by the new possibility to switch between several stages.

Snap! allows algorithmic problem solving on a high level, but it does not only allow the analytical approach, but also the playful, the experimental, the creative, ... What it does not allow is passivity, because nothing happens by itself. Media are essential system components, e.g. for visualizing the results - and they can also be the results themselves. *Snap!* therefore offers the chance to construct model solutions to current problems, e.g. also and especially in the media field. Through the self-created algorithmic framework of the model, understanding for the observed processes in the real model emerges. The experience of being able to gain this knowledge oneself enables the active, critical examination of future technology. The examples in this book are intended to show that this is possible in many areas with the aid of elementary methods. They are intended to encourage people to get started themselves. 😉

1.3 Objects and Inheritance by Delegation

If somewhat more extensive problems are processed, then the number of subproblems to be solved also grows. Often, these can be combined into groups that can be assigned to concrete *objects*. An important aspect of this way of working is that teamwork based on division of work can be realized well in this way, in which the different teams create objects that solve subtasks. The object-oriented way of working is often realized by creating *classes* that describe the behavior of a group of similar objects. *Instances* (exemplars) of these classes are then created to solve the problems. The approach is largely top-down and requires some abstraction. More suitable for beginners is the *prototype*-based approach used in *Snap!*, in which an example, the prototype, is created for each group of objects, which is developed and tested step by step. If one is satisfied with the result, then further objects of this kind are derived by duplication (*cloning*) of the prototype.

To object-oriented programming the concept of the inheritance belongs centrally, which can be realized by classes or by delegation. In the original article of Lieberman[14], which describes the prototype-oriented procedure with the delegation already very early, objects are understood as embodiment of the concepts of their class. Thus, the elephant *Clyde* stands there for everything, the viewer understands by an elephant. If he imagines an elephant, then it is not the abstract class of elephants that appears in his mind's eye, but Clyde. If he speaks about another elephant, here: *Fred*, then he describes him like this: "*Fred is just like Clyde, except that he is white.”*

What does this approach mean for the learning process? If the learner knows only one copy of a class (here: Clyde), then the prototype describes his knowledge completely, an abstraction is senseless for him. If he then gets to know other copies and describes them by modifying the original, i.e. replaces some methods by others, changes attributes and adds new ones, then the image of the class itself slowly emerges as an intersection of the common properties. Only now the abstraction process is comprehensible to him and, after a few attempts, viable itself. Delegation is thus a method that maps the learning process itself by creating prototypes instead of classes. In *Snap!* we work predominantly according to this principle, which is presented in detail below.[15]

In *Snap!* prototypes are created as sprites and equipped with the desired attributes and methods. Once their behavior has been sufficiently tested, clones can be created dynamically using the *clone* block. For each sprite it can be displayed from which sprite it was derived (*parent*) and which children it has (*children*...). The *parent* property can also be set and/or changed afterwards, so that the system of dependencies is dynamic. If the program stops, then all dynamically created clones are deleted, which is beneficial

A clone initially inherits (almost) all local attributes and methods of the parent object. This is indicated by a "paler" representation in the palettes. If a sprite overwrites inherited attributes or methods, then these replace those of the prototype as usual. If you delete the overrides again, the inherited attributes or methods appear in the palettes.

[14] Lieberman, Henry: Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, 1986, http://web.media.mit.edu/~lieber/Lieberary/OOP/Delegation/Delegation.html
[15] If you absolutely want it, then you can also implement a class system.

2 About *Snap!*

2.1 What is *Snap!*?

Snap! [16] was (and is) developed by *Brian Harvey* and *Jens Mönig* for the *Beauty and Joy of Computing* project[17] and is freely available on the Internet. Since the system runs in the browser, it does not require any installation and works on almost all devices[18]. Its interface and behavior are similar to *Scratch*[19], another free programming environment for children developed at MIT. However, the implemented concepts go far beyond that: here the roots lie with *Scheme*, a dialect of the *LISP* language, which has long been used at *MIT*[20] as a teaching language in the education of computer science students. It is introduced, for example, in a famous textbook by Harold Abelson and Gerald and Julie Sussman[21]. *Snap!* is thus a fully developed programming language, which consequently can be used in (almost) all problem areas. For most of them it is now also sufficiently fast. This is not self-evident and was a shortcoming of its predecessors. Graphical languages are largely concerned with controlling the state of the system and thus allowing, for example, infinite loops to be interrupted or access errors to data structures to be "tolerated". This leaves little time for the actual program execution.

Snap! is a graphical programming language: Programs (*scripts*) are not entered as text, but are composed of tiles. Since these tiles can be put together only if this makes sense, "wrongly written" programs are largely prevented. *Snap!* is therefore largely *syntax-free*. Nevertheless, it is not completely free of syntax, e.g. because some blocks can process different combinations of inputs: if you put them together incorrectly, errors can occur. However, this is more likely to happen with advanced concepts. If you use them, you should know what you are doing.

Snap! is exceptionally "peaceful": errors do not cause program crashes but are indicated by the appearance of a red mark around the tiles that caused the error - without dramatic consequences. The used tiles, which include the newly developed blocks, are always "alive". They can be executed by mouse clicks, so their effect can be directly observed. This makes it easy to experiment with the scripts. They can be tested, modified, disassembled into parts and reassembled in the same or different ways. This gives us a second approach to programming: in addition to problem analysis and the associated *top-down* approach, there is the experimental *bottom-up* construction of subroutines that are assembled to form an overall solution.

Snap! is descriptive: both the program sequences and the assignments of the variables can be displayed and tracked on the screen if required. This makes it ideal for simulations, for example.

Snap! is extensible: by the implemented LISP concepts new control structures can be created, which work e.g. on special data structures.

Snap! is object-oriented, even in different ways: Objects can be created both by creating prototypes with subsequent delegation and in different ways via classes.

[16] https://snap.berkeley.edu/snap/snap.html
[17] https://bjc.berkeley.edu/
[18] Meant, of course, computers, tablets, smartphones, …
[19] http://scratch.mit.edu/
[20] Massachusetts Institute of Technology, Boston
[21] Abelson, Sussman: Struktur und Interpretation von Computerprogrammen, Springer 2001

Snap! is first-class: all structures used are first-class, i.e. can be assigned to variables or used as parameters in blocks, can be the result of a function block or the content of a data structure. Furthermore, they can be unnamed (*anonymous*), which is important for the implemented aspects of the lambda calculus, the basis of LISP. Consequently, the logo of *Snap!* contains the same proud lambda that used to be found in the hair of Alonzo, the mascot of *BYOB*.

Alonzo

2.2 What is *Snap!* not?

Snap! is not a production system. It is a learning environment that was developed, among other things, on behalf of the U.S. Department of Education as part of CE21 (*Computing Education for the 21st Century*) and is also intended to reduce the dropout rate in technical subjects. It is a tool for implementing and testing informatics concepts in an exemplary manner.

Snap! is primarily used for work in the field of algorithms and data structures, but essential areas of computer science such as access to files or hardware can also be embedded in the browser environment, sometimes via libraries. The microphone and the camera of the computer are directly addressed, and the built-in *url* block allows quite simple accesses to the Internet and thus, for example, via intermediate servers, the use of databases or external hardware

Since the code of *Snap!* is freely available, there are different modifications. Whether this is a blessing or a curse remains to be seen. In any case, there are now specialized versions e.g. for the areas of *physical computing*, *robot control* or work in the *network*, so that corresponding simple examples of the first version of this script have been deleted.

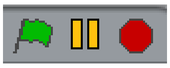
2.3 The *Snap!* - Screen

The *Snap!* screen consists of six areas below the menu bar[22].

- On the far left are the command tabs, which are divided into the categories *Motion*, *Looks*, *Sound*, and so on. If you click on the corresponding button, the tiles of this section are displayed below the button. If they do not all fit on the screen, then you can scroll the screen area in the usual way. If you want, you can display the tiles of all sections one below the other.
- To the right of this, i.e. in the center of the screen, the name of the object currently being edited - called a *sprite* in *Snap!* - and some of its properties are displayed at the top. You can - and should - change the default name of the sprite here.
- Below this is an area where, depending on the tab, the sprite's *scripts*, *costumes* and *sounds* can be edited or created.
- At the top right is the output window in which the sprites move: the *stage*. This can be resized using the buttons above it, the entry in the settings menu (*Stage size ...*), a corresponding command block or by simply "dragging" with the mouse. If you set the checkmark in front of the variable name in the *Variables* palette, the variables will be displayed on the stage, if necessary, with a *slider* that allows you to easily change the values. Since variables can contain anything (numbers, texts, lists, sprites, programs, ...), the state of these variables can be visualized at any time.
- At the bottom right, the available sprites are displayed. If you click on one, the center area changes to its scripts, costumes or sounds - depending on the selection. To the left of the sprites, an icon of the stage, or, if available, the icons of several stages are shown. You can also switch between them by clicking on them. Each stage has its own project, which is independent of those of the other stages. However, it is possible to exchange data between the projects.

[22] The layout of the areas can be changed using .

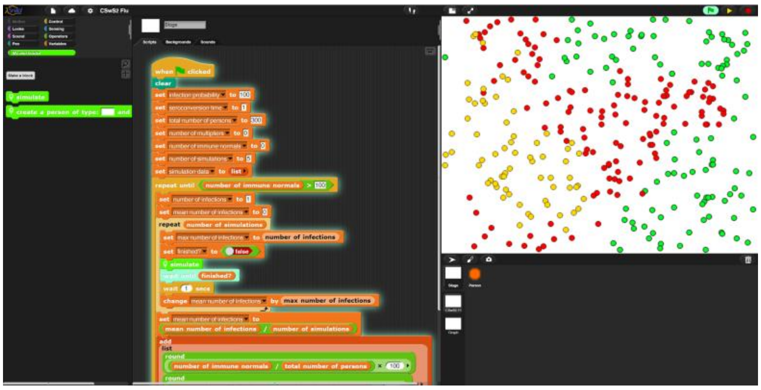
- The menu bar itself offers the usual menus for loading and saving the project and individual sprites on the left. Furthermore, several settings can be made. One possibility is to set the language. I still recommend staying with the English version, because this way you can distinguish your own blocks, e.g. named in German, from the native ones at first sight.
- On the far right we find the green flag known from Scratch, with which several scripts can be started simultaneously when using the corresponding block. The pause button next to it pauses everything and the red button ends all running scripts. Individual scripts or tiles can be started by simply clicking on them.



2.4 Example for Experienced Users: Flu

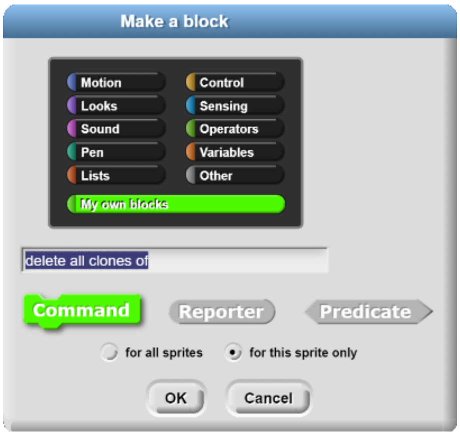
Level: *high school* Materials: *Flu*

The example simulates the spread of a flu epidemic under different conditions. It serves as a quick overview of the main possibilities of *Snap!* and is intended especially for experienced programmers. Beginners should rather read the next chapters first.



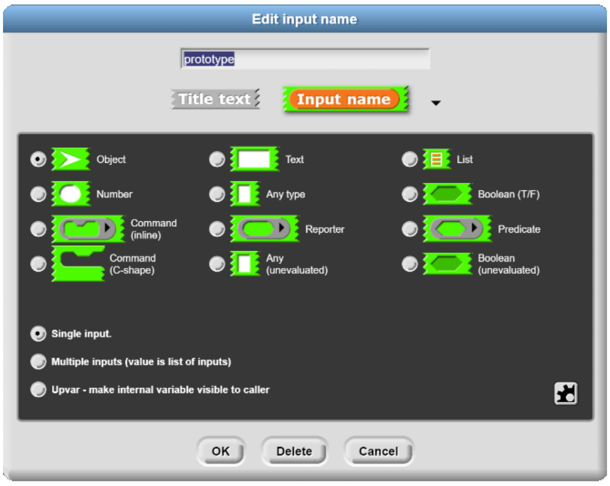
The question is what proportion and which particular groups of people in a population should be vaccinated if the spread of an influenza epidemic is to be stopped. The question is not so easy to answer, because the result depends on various parameters: the *probability of infection* indicates how likely it is that a healthy person will be infected when in contact with a sick person, the *seroconversion time* is the time between infection and immunization, the *numbers of healthy and sick persons* at the beginning of the simulation determines the number of contacts between them, and the type and number of *multipliers* indicates how many persons in the population have particularly many contacts or contact with particularly widely separated groups. If one of them becomes infected, for example, the disease is quickly carried to distant areas. Since contacts, infections, etc. are random, we will only obtain viable results if we run the simulation several times with the same parameter values in each case - and then it still remains to discuss which values represent "results" in the sense mentioned at all. The topic is therefore perfectly suited for a small classroom project. A "steering group" develops the superordinate scripts, which we want to assign to the *Stage* here. It coordinates the distribution of tasks with the other groups. The other groups develop auxiliary methods as well as the prototypes *Person* and *Graph*, each with its own stage, which are almost independent of each other, and think about the data exchange.

Writing own Methods



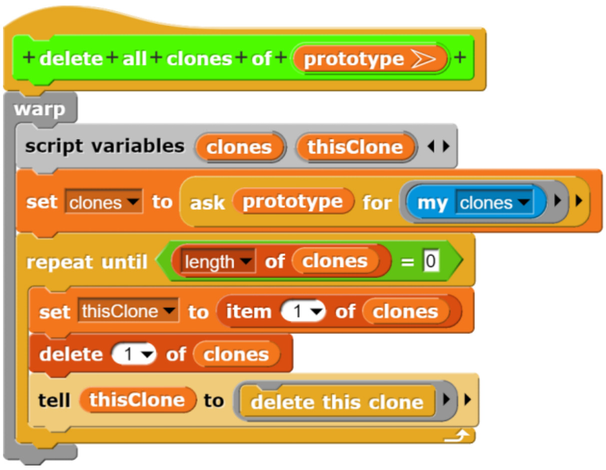
It is often necessary to get rid of the created clones of a prototype without terminating the program. We achieve that here by a new local method *delete all clones of <a prototype>* of the stage. This is a *command* block, that is, a command that (here) has a parameter. (Function blocks are called *reporters* in *Snap!*) New blocks are written in the block editor, which is invoked with the *Make a block* button we find in the palettes or by right-clicking on the script layer and there in the context menu. First, we specify the method name, with spaces and special characters if desired, select the type (*Command*, *Reporter* or *Predicate*) and specify whether it is a *global* (*for all sprites*) or *local* (*for this sprite only*) method. We can also choose the palette in which the block will be included and the color it will be given.

In this case, we first create a new palette (*category*) using the file menu (*New category...*), name it *My own blocks* and select an optimistic green as the color, which clearly distinguishes the own blocks from the default ones. After pressing the Return key, the block editor opens and the block name appears - with + signs in the spaces and margins. There, by mouse clicks, we can open another menu that allows to insert parameters (or more texts/symbols) in these places and specify their type if needed. In our case we click on the far right, enter the parameter identifier *prototype* and click on the small right arrow for typing. Then a selection box opens.[23] We select *Object* (the arrow) as the type, return to the block editor and drag the required commands into its script area.



Our method uses two script variables (*clones* and *thisClone*) which are known only in this block. It asks the parameter *prototype*, which is later passed with a reference to the "parent person", for its children - these are then all dynamically created "persons" that occur.[24] As long as there are any of them left, it remembers the first one in one of the script variables, deletes it from the list and then asks this person to delete itself with *tell <thisClone> to <delete this clone>*.[25]

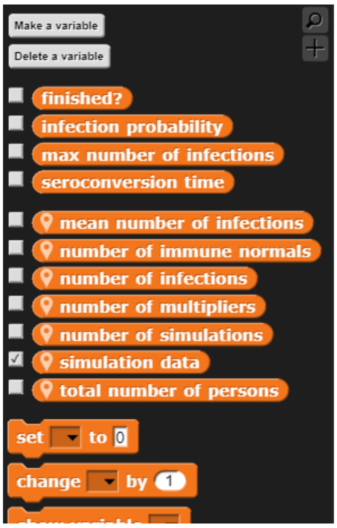
The method is called by passing an object (here: person) to it.



[23] This box and the details of the current Snap! version are described in great detail in the Snap! reference manual, which can be obtained by clicking on the Snap! icon at the top-left of the window.
[24] The clones created statically via the context menu in the sprite area are not found there.
[25] The delete block can only be found in the sprite palette. But you can reach it in the stage by using the search function at the top of the palette area.

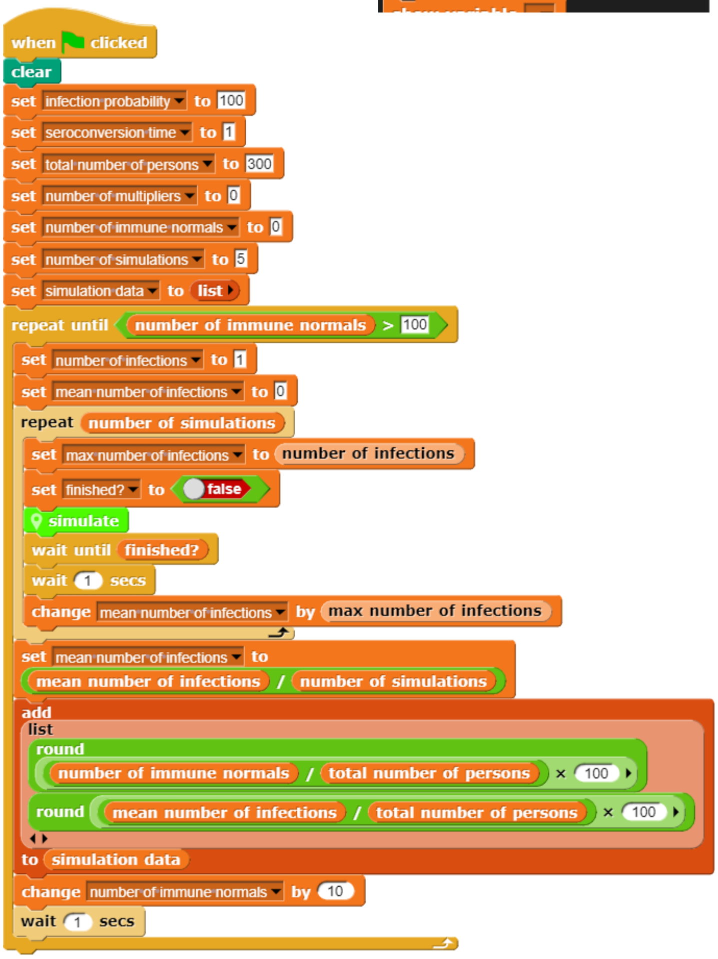
Elementary Algorithms and Variables

To set the parameters and other control values, we use the *Stage*, which we click on in the Sprite area. This stage reacts to the message "*Green flag clicked*" by setting the initial parameters and determining which variables are to be measured during the simulations. After that, corresponding simulation runs are started.



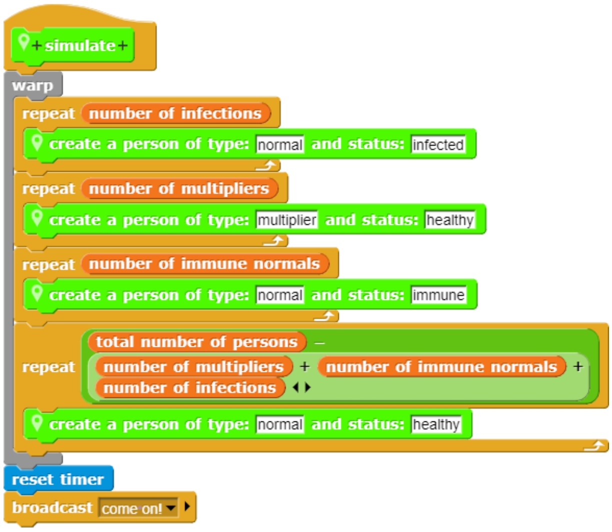
In detail: We can "fish" a reference to the *Person* prototype using the *object* block from the *Sensing* palette. If needed, we can store it, like any other value in *Snap!*, in a variable, which can be either global (*for all sprites*) or local (*for this sprite only*). Variables are created in the *Variables* palette using the *Make a variable* button. At the same time, we can create all other required variables, whereby those that are only required within the stage are marked as local. You can recognize them by the "marker" in front of their name. The others are global. Global variables are displayed at the top of the *Variables* palette, followed by the current local ones. Then the output area is cleared, some variables get appropriate initial values and a list called *simulation data*, which should hold the simulation results, is cleared (*set <simulation data> to* <list>). This part could well have been put in a separate block, but since we want to experiment with the variable values, it is better to have them "on the table".

In the following, the number of initially vaccinated (the *number of immune normals*) is gradually increased from zero to 100. The control structures for this can be found in the *Control* palette. For each value, a series of simulation runs is performed and the average of the results (here: the *maximum number of infected*) is determined. The variable *number of simulations* determines how often this is done. After each run, the results are entered as a percentage in the *simulation data* list. Finally, it is asked to generate a graph from this data. Another working group can take care of this.



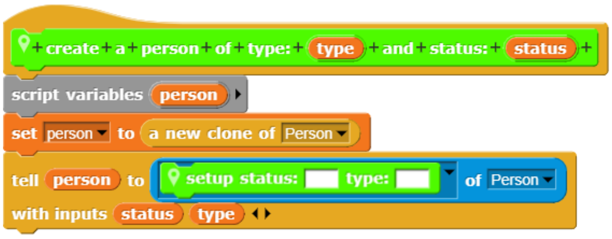
Create Objects

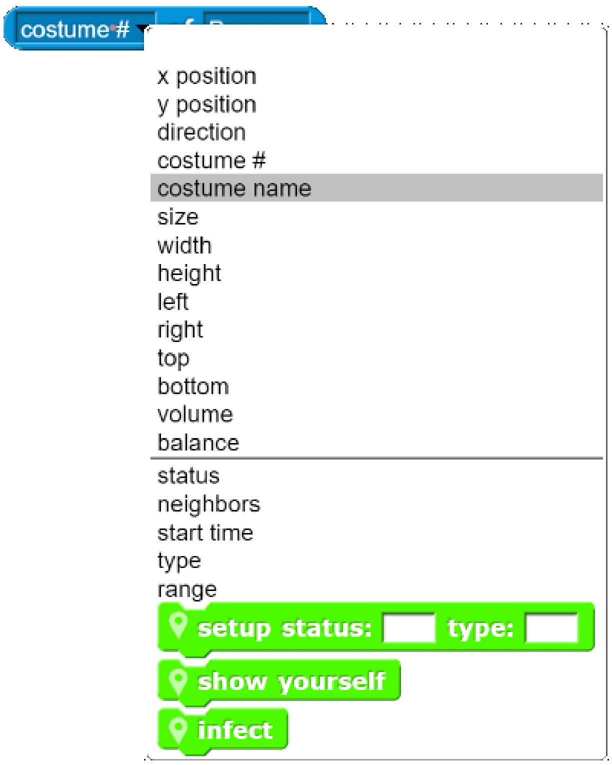
In the control program a method *simulate* is used. In it, some initial values are reset and the corresponding number of individuals is generated, differing in type (*normal*, *multiplier*) and status (*healthy*, *infected*, *immune*). To increase the speed, this is done in a *warp* block. Then the simulation run is started by sending the message "*come on!*" which is "heard" by all objects in the system.



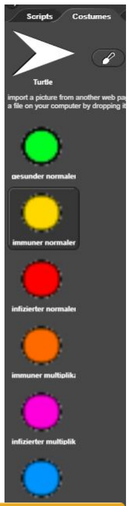
How to create objects?

In the *create a person of type <type> and status <status>* method written for this purpose, we first declare a local script variable to which we assign a reference to a newly created clone of the specified prototype. After that, the clone exists, is visible, and is accessible under the name *person* - very simple. However, the clones should differ in type and status. For this they contain (in this case) a local method inherited from the prototype *setup status: <status> type: <type>*. We have to call this with the parameter values passed. We therefore *tell* the object *person* that it should execute this method. Since this is local for persons, we take the *<attribute> of <object>* block from the *Sensing* palette, select the prototype (in this case: *Person*) in the right field and then the desired method (in this case: *setup* ...) in the left field. Because there are two parameters to be specified, we expand the block with the small arrow keys and specify the status and type behind *with inputs*. The block is to be understood as "*person, please execute in your context of methods and variables the passed method with the given parameters*". The block is equivalent to the well-known dot notation of OOP languages: e.g. `person.setup(status,type);`



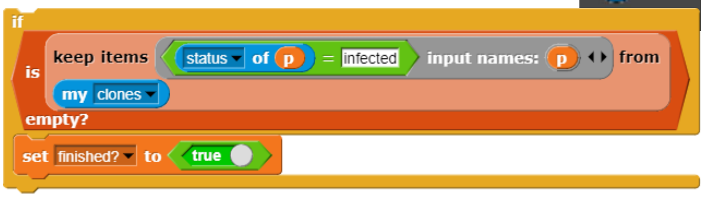


Communicate with Objects



We now come to the actual actors of our flu project: the *persons*. These are symbolized by small circles whose color expresses their status. "Normal" persons scurry around in their environment in a relatively small scale, meeting the neighbors they can infect or who can infect them. After a certain time, the seroconversion period, they become immune and are no longer infectious, nor do they become infected. Vaccinated people are immune from the beginning. Some of the people are "multipliers", that is, they jump around quite wildly and can spread the infection quickly. They are similar to the normals but color-coded slightly differently. We make appropriate costumes in the graphics editor or a drawing program and import them into the *Costumes* area.

After creating the persons, they all receive the message "*come on!*" to which they react because they have a hat block from the *Control* palette that reacts to "*come on!*". After that, they get into a loop that terminates when the global variable *finished?* gets the value *true*. This is the case when there are no more infected, so when the list of clones that are still infected is empty.



In this loop the following actions are executed repeatedly:

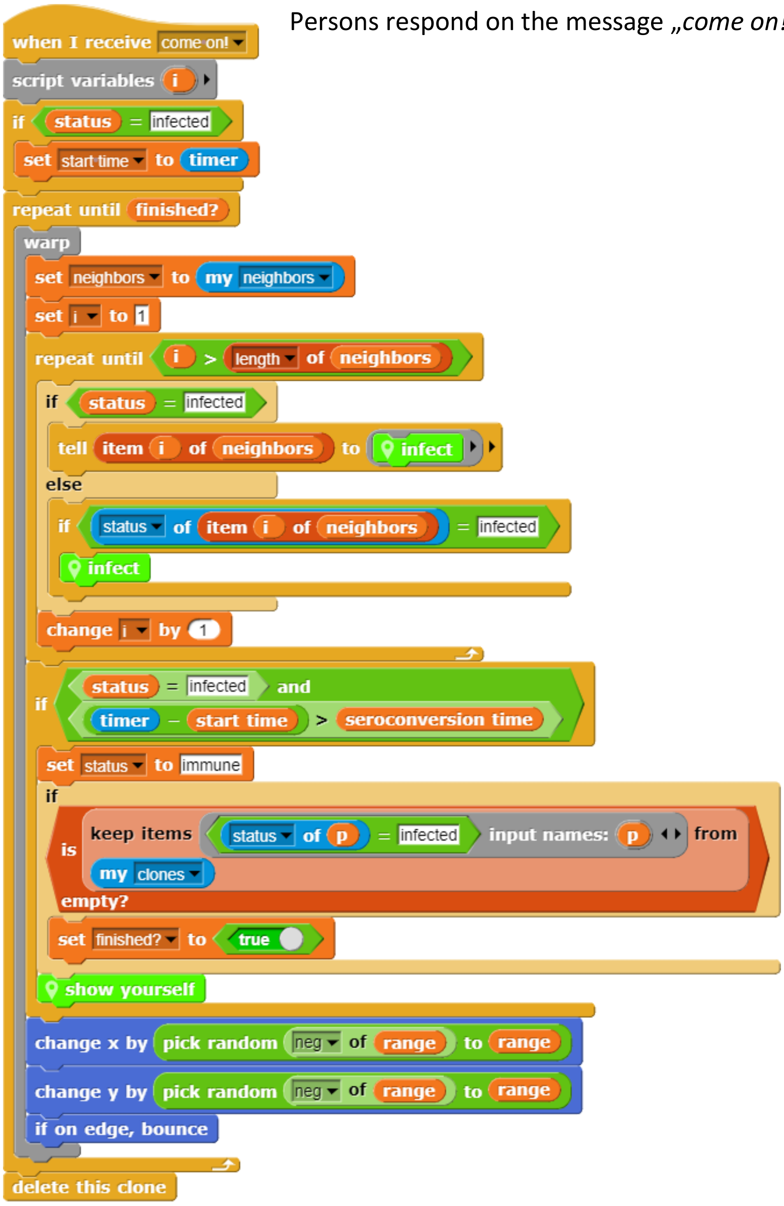
1. Objects near the person are searched and stored in the *neighbors* list.
2. All remaining neighbors are infected if necessary or infect the person if they are ill.
3. It is checked whether the person must be immunized after the seroconversion time has expired. The corresponding variable values are changed.
4. After that, the person moves according to their type.
5. After the loop is finished, the clone deletes itself.

Since these processes involve exchanging data between persons and initiating method calls from the other persons, the example shows some procedures for doing so:

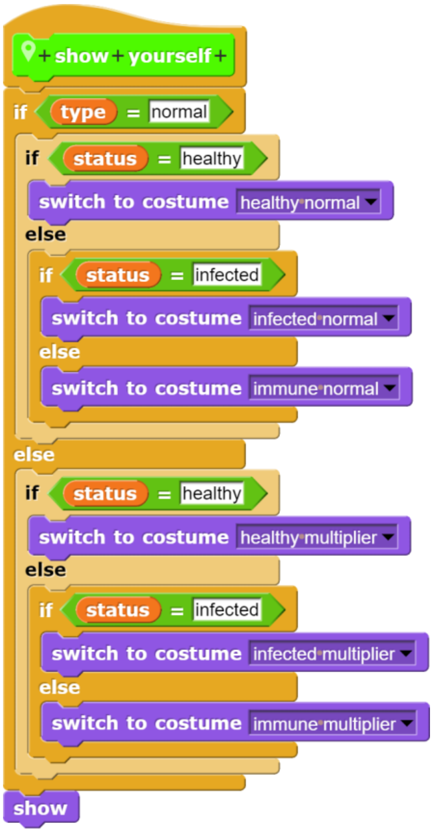
The *tell <object> to <run this script>* block is used to ask a person to get infected. If you call a function (which returns a result) of another object, you use the *ask <object> for <reporter>* block. Attributes and local methods of other objects are obtained via the *my <attribute>* block from the *Sensing* palette, which you have already met. Here we query the state of an object by executing the *<attribute> of <object>* block in the context of the other object. The blocks are surrounded by a gray ring (*ringified*) indicating that the unevaluated code of the block is passed and not its actual result.

In two places below, local methods - shown in green - are executed in the context of the object. This happens "normally" when the block is reached.

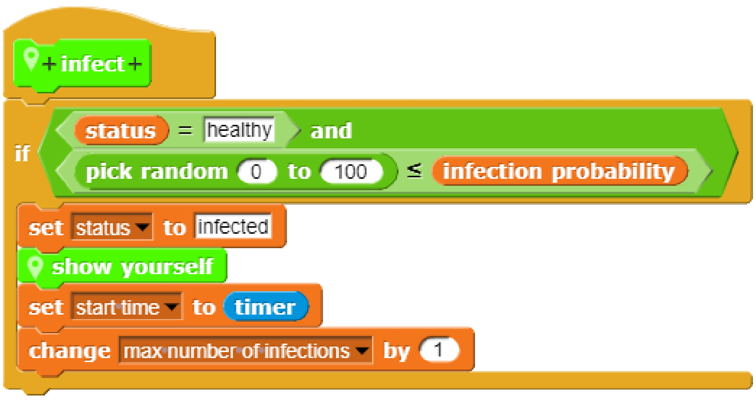
Persons respond on the message *„come on!“*:



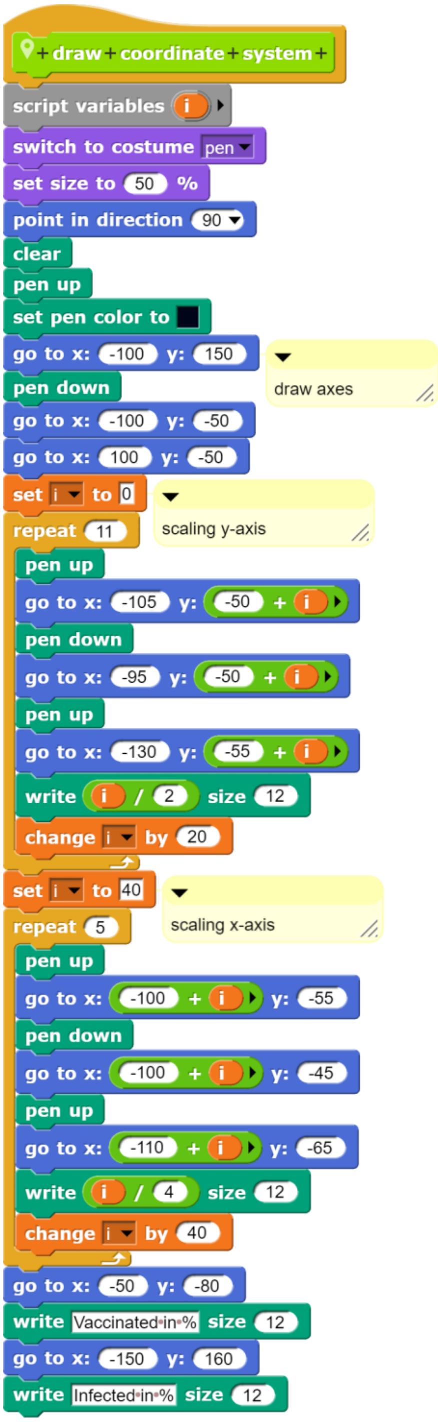
The method *show yourself* selects the appropriate costume.



The method *infect* infects the current object if necessary and enters freshly infected into the corresponding list. After that, the appearance of the object is changed.



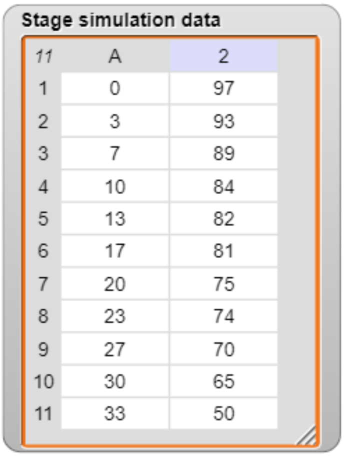
Draw a diagram



Finally, we want to have our results represented in a diagram. We measured the initial number of vaccinated (in %) and the maximum number of infected (in %). For this purpose, we create a second stage called *Graph* with *New scene* from the File menu[26]. On this stage there is a new, second project, which has nothing to do with the first one. Its objects, variables and methods are unknown in the second scene. However, we can use the *export/import* functions to send objects and/or scripts from one project to the other - i.e. via files. In addition, we can switch between scenes, sending data from one project to another. We want to go this "internal" way.

In the second scene, we create an object called *Pen*, which we give a nice pen as a costume. First, we let the pen draw a coordinate system on the screen and label it. We find the blocks for this in the *Pen* palette.

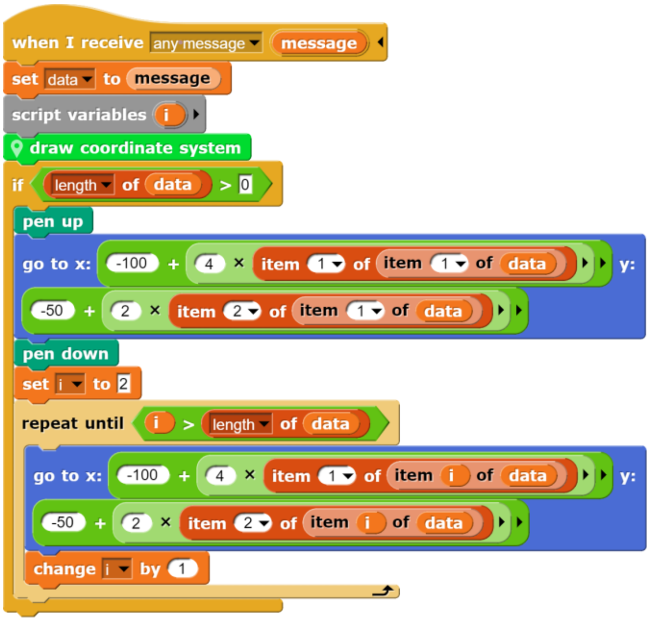
The determined data are available in list as variable *simulation data*. They are sent to the *Graph* scene after completion of the simulations. The *Pen* object retrieves this data from the *message* variable and stores it as *data*.


Stage simulation data

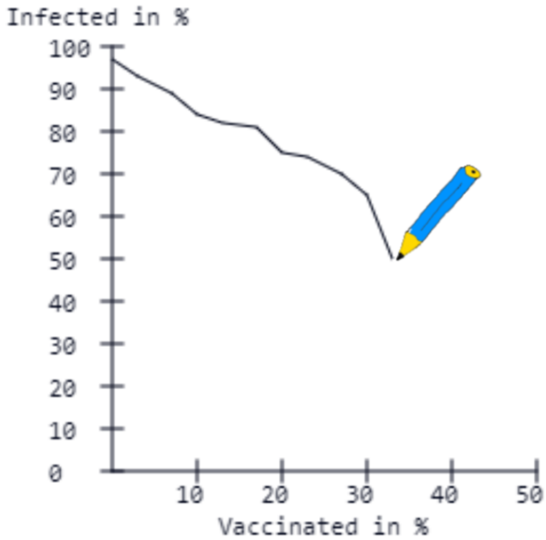
11	A	2
1	0	97
2	3	93
3	7	89
4	10	84
5	13	82
6	17	81
7	20	75
8	23	74
9	27	70
10	30	65
11	33	50

[26] only to show this possibility already here 😉

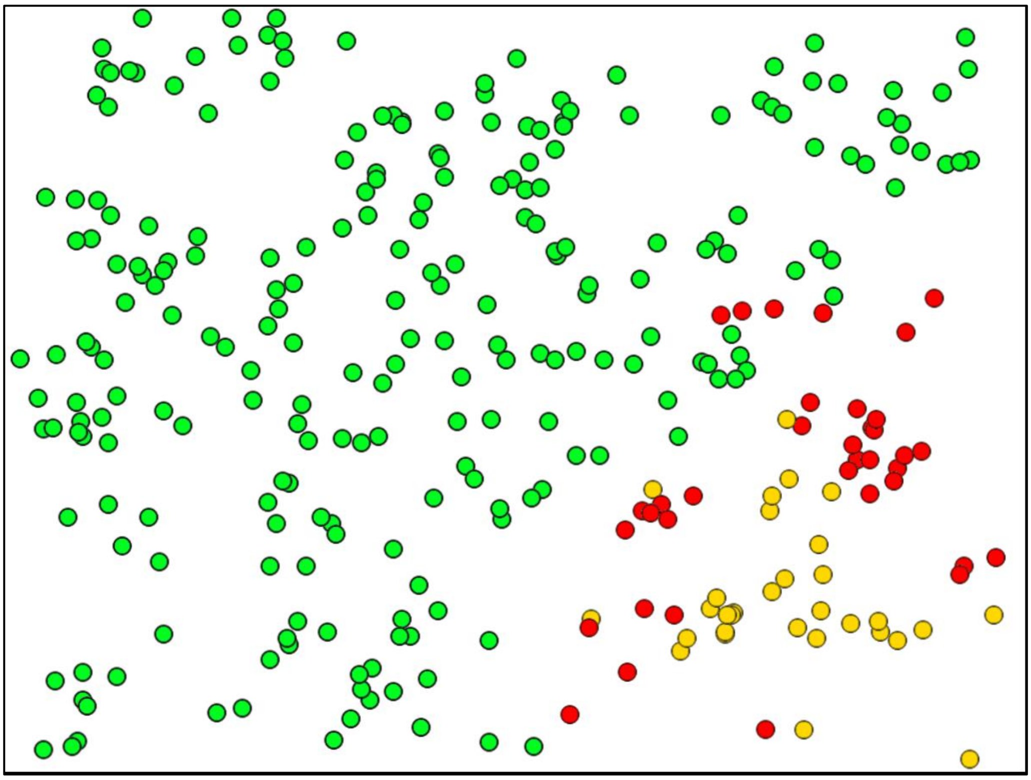
After that, the data points are transferred to the diagram. We send the pen to the first data point given by a list with the two entries mentioned. After that we guide it lowered to the remaining points - with some conversion.



The result can be admired on the output area:



In each case, 300 "persons" without multipliers and with only one initially infected person were used (red: *infected*, yellow: *immune*, green: *healthy*). As can be seen, if half of the population is to remain healthy in this model, then 30% must be vaccinated.



3 Examples for "Data and Information"

3.1 Examples for Communication in a Given Context

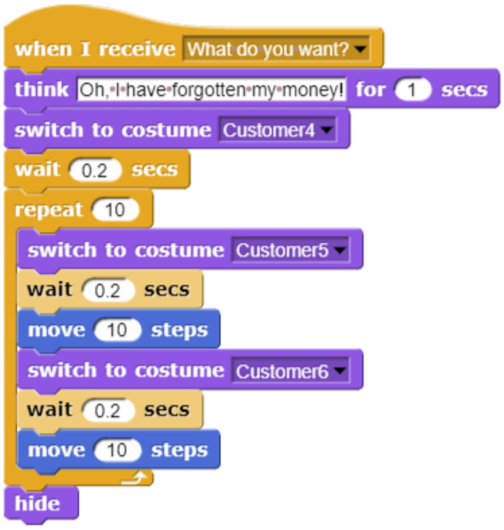
As described in the didactic considerations, we need scenarios in which the computer system only acts as a vehicle for messages that are "understood" by the participants. In the simplest case it only represents the context, e.g. in the programming of a story. However, the information system is not irrelevant, because on the one hand its use is learned and thus later, more "informatic" tasks are prepared. On the other hand, the use of different objects communicating via messages provides an intuitive introduction to object-oriented modeling. The following examples are therefore particularly suitable for the beginning of a programming course.

At the Greengrocers

Level: from middle school Materials: *At the greengrocers*

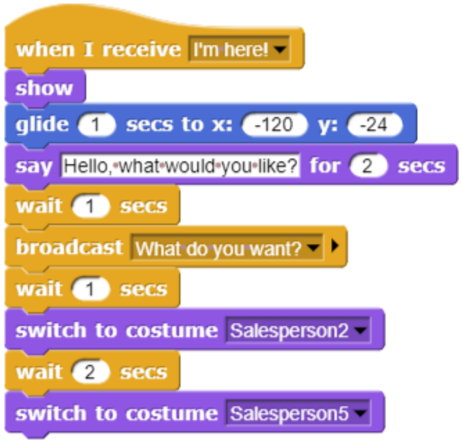
Two people act in a store[27], e.g. by a customer entering the room (with leg movements through costume changes) and then sending a message ("*I'm here!*"). Thereupon the saleswoman appears, asks for the wishes, ... - all controlled by messages. The context in this case is clear and largely given by the background image, and since the objects react only to certain messages, it is also clear what to do in each case. Even if the situation is trivial, there is no doubt about the distribution of roles: Messages in the information system consist of texts which are interpreted by the agents and, if necessary, trigger actions.

Scripts of the customer:



[27] Costumes partly from Scratch and/or own photos.

Vendor scripts:



The given animation program, which the learners should be involved in creating, provides a simple framework for the initial lessons, which is modified and supplemented by the learners. Not to be underestimated is the work with the costumes, e.g. to visualize movements. Working with the built-in graphics editor and other graphics programs, which in turn provide different graphics formats, which is important, for example, for the transparency of the background, motivates some of the learners more than the direct entry via scripts. If different costumes are created, then they should of course also be used - and for this, one needs program scripts. The detour via the graphics leads to the algorithms - however, based on self-created (partial) products, which can greatly increase the motivation.

Why is the graphic representation so important for the learners? The context is not only revealed by the texts, but also by the appearance, posture, etc. of the actors and their environment (see: "*In the Bistro*" below). What is portrayed here does not have to be said anymore, but it is decisive for the interpretation. There is a difference whether the exclamation "*That's cabbage!*" occurs in a greengrocer's store or in a classroom.

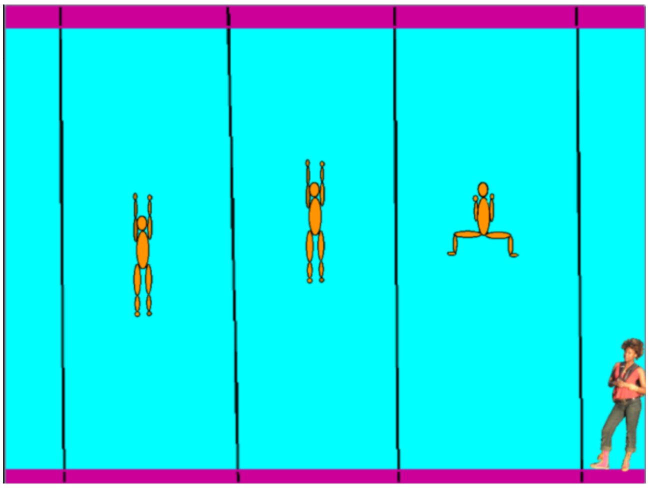
The example leads to different stories that emerge after establishing a defined initial state ("*green flag pressed*") in the interplay of messages sent and events triggered by them (with their treatment) following the given examples. The quality of the results is then expressed in the imagination, complexity and wittiness of the stories.

Although messages, events, possibly states, which can be described by local variables as "*additional attributes to the already existing ones*" are used, the focus of what happens should not be on technical language. Of course, it is necessary to talk about the processes, and then one may as well use the usual terms. But this is only a (quite desirable) side effect. The goal of teaching is to activate the learners and to encourage imaginative action. The use of a formalized way of speaking ("*The attribute x-value of the object customer is reset by calling the method change x by -5.*") is rather not part of this. It is sufficient to state that the customer moves to the left.

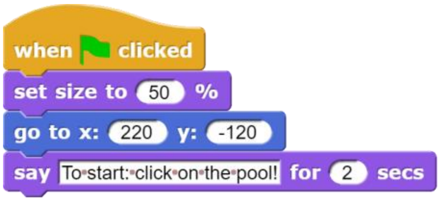
Swimmers

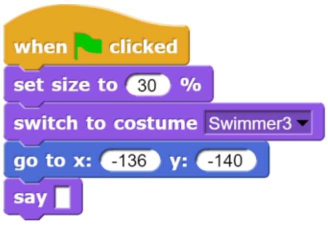
Level: *from middle school* Materials: *Swimmer*

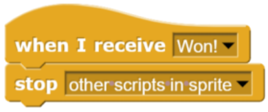
We draw a swimmer in different swimming phases and duplicate it several times. On a message ("*Go!*") the swimmers swim with randomly chosen initially variable speeds to the other end of the swimming lane. When one reaches the edge, he is happy and stops all other swimmers (and himself) by sending a message.



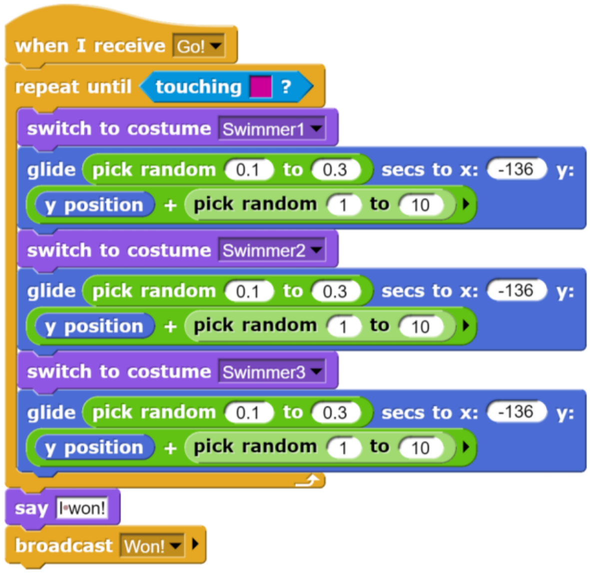
The trainer's mini scripts are trivial ...







... and also, the swimmers have not much more to do than swim.



This example, whose background image defines the context without further explanation, also serves only to make clear the differences between the message and the information conveyed. However, it can easily be extended, e.g. by assigning fixed speeds to the swimmers (which requires a new attribute, i.e. local variable) or by having them turn around at the end of the lane to swim back to the starting point. Other swimming styles can be easily represented, success statistics can be kept, and the stop at the finish can be greatly improved. However, the connection between the victory message and the coach's statement "*I don't believe it!*" remains the secret of those involved.

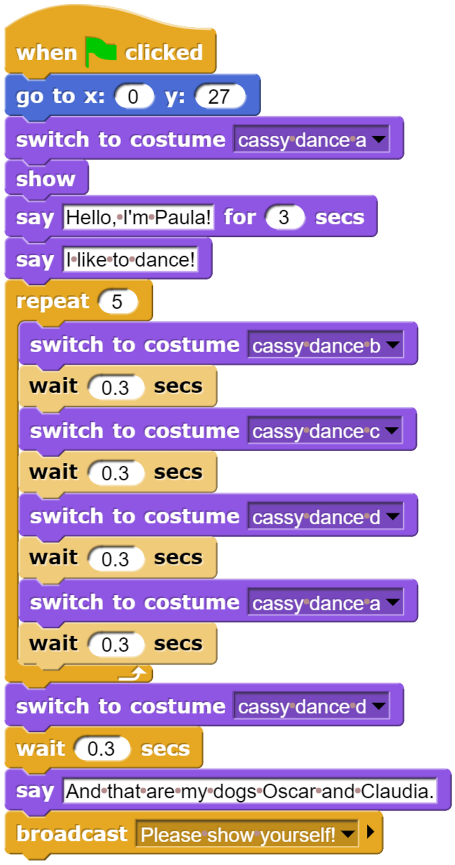
Self Portrait

Level: *from middle school* Materials: *Self-portrait*

The students introduce themselves: where they live, their way to school, their hobbies, …

In this case, the distribution of roles is even clearer: the computer system provides images and texts, i.e. data. The person portrayed was responsible for selecting them, and this selection is supposed to make Paula appear sympathetic to the recipients of the data. Does that work?

Script of Paula:



Paula knows nothing about the recipients of her data. If they like dogs, the sympathy will work out: Almost everyone thinks puppies are cute. But if they have a cat that is chased by the neighbor's dog, they will find that Paula's dogs will have to be treated with caution in a few months. In that case, Paula should rather use pictures of equally cute kittens. So, it would be in Paula's interest to know as much as possible about the recipients and to adapt her self-presentation to these interests if she wants to appear sympathetic to everyone.

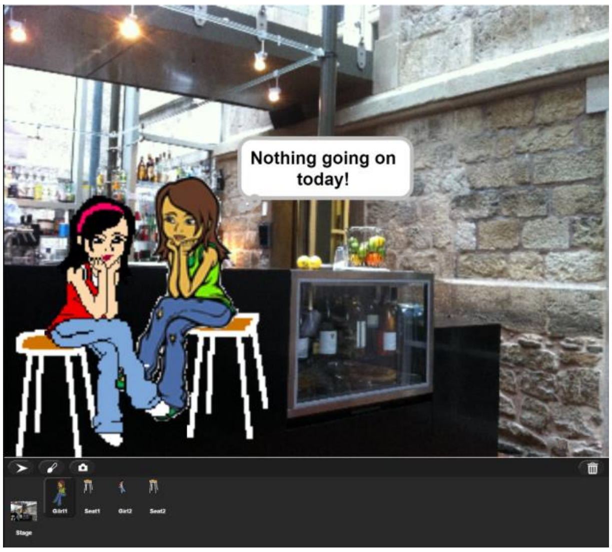
If Paula were a politician, a company, or any other institution, she would have a massive interest in gaining data from her "readers". Even trivial data like dog/cat preferences can be deduced from the purchase of pet food - not always correctly, but mostly. For normal citizens they are probably uninteresting, for the politician however not, because he must decide whether he hugs on his press photos more children, dogs, or cats. If he can send the right picture to every interested party, then the resulting wave of sympathy will be able to deliver some other political content as well.

We can learn from this example that even a trivial project like a self-portrait provides starting points for the discussion of socially relevant issues, in this case: for the motivation to collect personal data.

Scripts oft the dogs:

In the Bistro

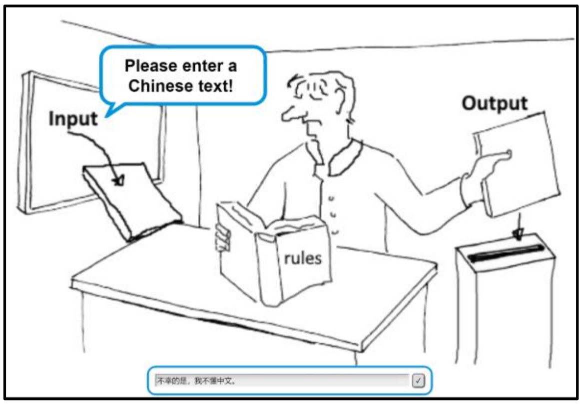
Level: *high school* Materials: *In the bistro*

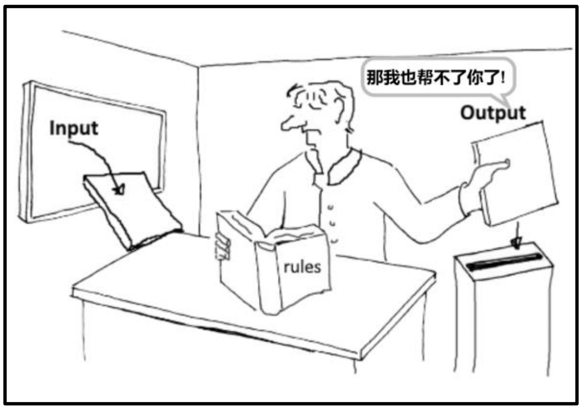


The example is directly equivalent to the vegetable store, perhaps for slightly older students. Using *Snap!* opens up some possibilities, for example, in animating the sprites. For example, one could move the limbs in a controlled way on the common object ("attached parts"). Most importantly, using *Snap!* even for simple animations eliminates the need to change tools when working on more complex problems later. And the role of body language becomes more than clear. If the conversation shifts to the realm of irony, for example, then many utterances will only be properly interpretable against the pictorial background. And of course: the story is just getting started. What happens next? One does not know!

Searle's Chinese Room

Level: *high school* Materials: *Chinese room*





Searle’s example serves to discuss possible artificial intelligence. There is a person in a room who does not know Chinese but has a book that contains rules in their language for modifying Chinese texts. The person is given Chinese texts, applies the rules, and passes the results back to the outside world. [28] People outside the room believe that the person inside knows Chinese.

The example is topical in view of the discussion about artificial intelligence. But it is of course also an excellent example of the relationship between information and data. The "data-processing system" inside understands nothing at all but produces results that are interpreted by users as manifestations of intelligence.

Note: Since I unfortunately do not know Chinese, the texts in the example shown were translated by a program. The results were then copied into the input fields.

[28] https://de.wikipedia.org/wiki/Chinesisches_Zimmer

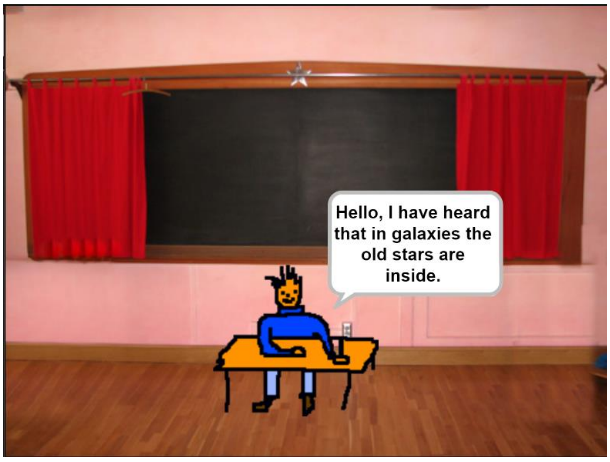
3.2 Examples for Communication with an Open Question

In this scenario, two human partners who do not know each other communicate with the help of an IT system. One asks a question, the other helps him with an answer - hopefully to the best of his knowledge and belief.

Distance Learning Astrophysics

Level: *high school* Materials: *Distance learning*

We take advantage of *Snap!'s* ability to work with multiple stages and thus manage two projects that can communicate with each other. A student asks a question to the distant astrophysicist, who provides him with material[29], which he hopes the student will be able to infer the answer. In this case, the material is some galaxy images. By the question a context is produced, for which the answerer compiles and transmits data, from which he believes that the information searched for opens itself to the questioner from it. The questioner then interprets the data material as assistance related to his question - and not as decorative material for the classroom.



The partners communicate via the block to change the scene and can transmit texts and/or other data to each other - in analogy to the block *broadcast* ….

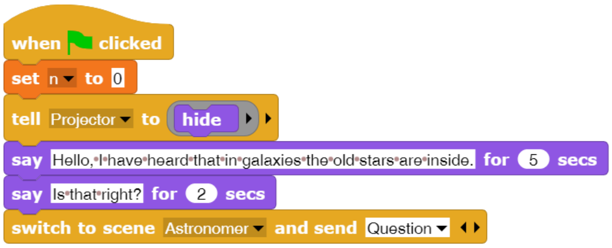
```
switch to scene [Astronomer ▾] and send [Question ▾] ◂▸
```

In the case of somewhat more complicated questions, the data must of course first be compiled, evaluated, presented, and interpreted before deciding whether the original question can be answered with it (see next example). This can also be used for communication with the teacher.

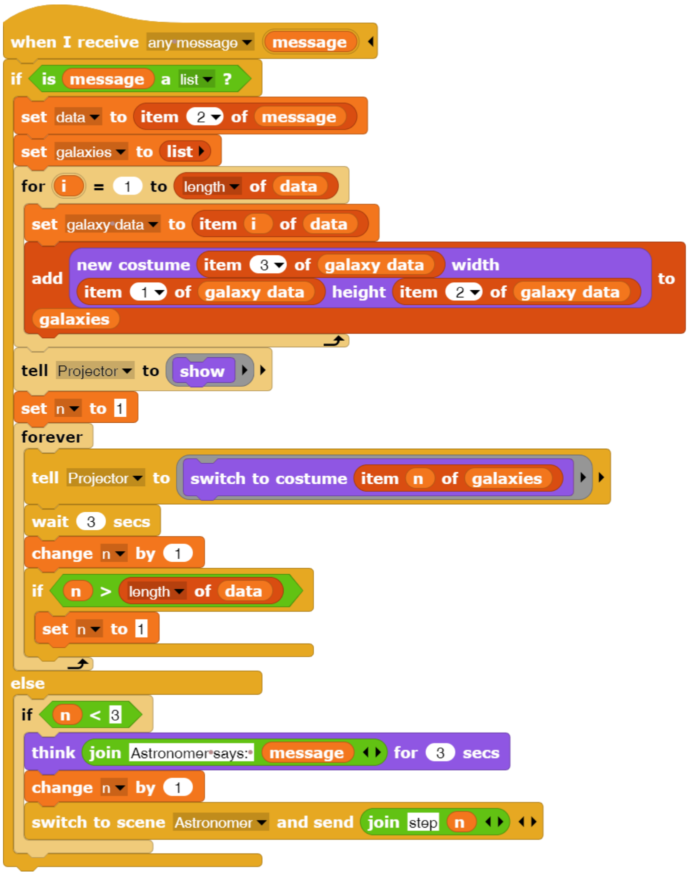
The scripts within the projects of the participants are simple. To be understood is the compilation of image data for transmission on the part of the astronomer and the inverse generation of images on the part of the student, the communication between the scenes. Central is thus the data exchange between remote partners.

[29] From https://en.wikipedia.org/wiki/Galaxy

The student asks his initial question. Then he causes the scene to switch to the astronomer's scene.

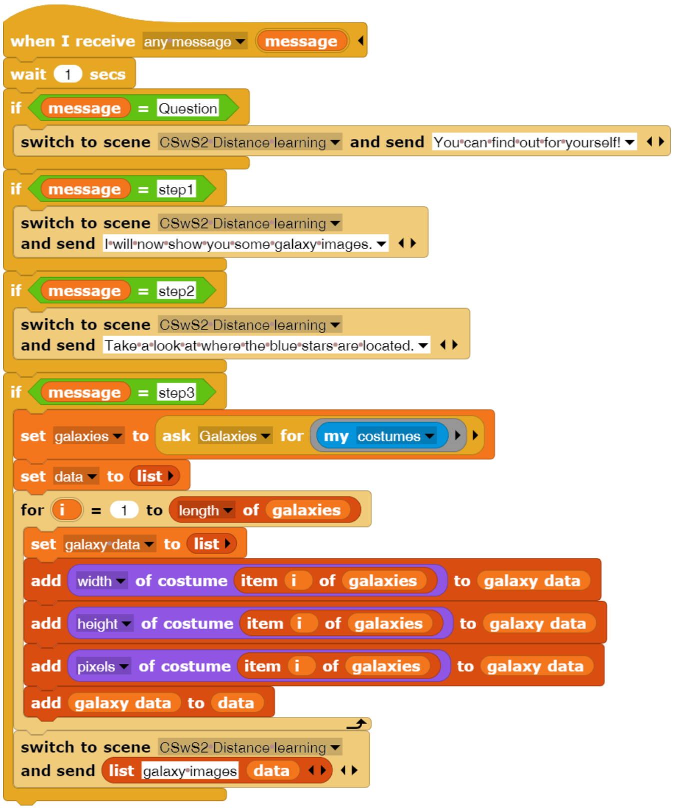


If he receives an answer, he first looks to see if it is a list (i.e. data). If this is the case, then he assembles the galaxy images from the data and displays them on the projector in an endless loop. Of course, he has to know how the data are structured.



If he does not receive a list from the astronomer, he displays the astronomer's answer for 3 seconds and switches back to the astronomer. Thereby he counts through the dialog steps (*step1, step2, …*).

The astronomer wants the student to find the answer to his question himself. He shows this step by step by some remarks he transmits to the student's scene.



Then he generates transferable data from the galaxy images, i.e. lists of width, height and pixels of the image, which he combines to the list *data*. Then he sends the whole package to the student.

The proceeding of the astronomer is to be understood only if he takes from the questions of the pupil that he should be guided as a newcomer to own realizations. Otherwise, he could simply answer with "yes". So, he doesn't suspect a colleague, journalist researching for an article, wallpaper designer looking for pictures, ... on the other side of the line. This can be true - or not. If it is not true, then the misinterpreted context can lead to some trouble. Examples of this can easily be found.

Calculation of the Distances of the red and blue Pixels from the Center of the Galaxy

Level: *high school* Materials: *Distance learning II*

We want to continue our astronomy example and enable the student to measure the average distances of the red or blue pixels from the center of the galaxy. For this, of course, we need a tool that on the one hand can read and rewrite RGB values at all, and that then evaluates the processed image data. This is a typical task for the expert, our astronomer. He writes two functions, which select and amplify the "predominantly" red or blue pixels from a galaxy image and then calculate the average distances of the red or blue pixels to the center of the image, i.e. approximately to the center of the galaxy. He sends these functions "by mail", i.e. via file export and import to the student.[30]

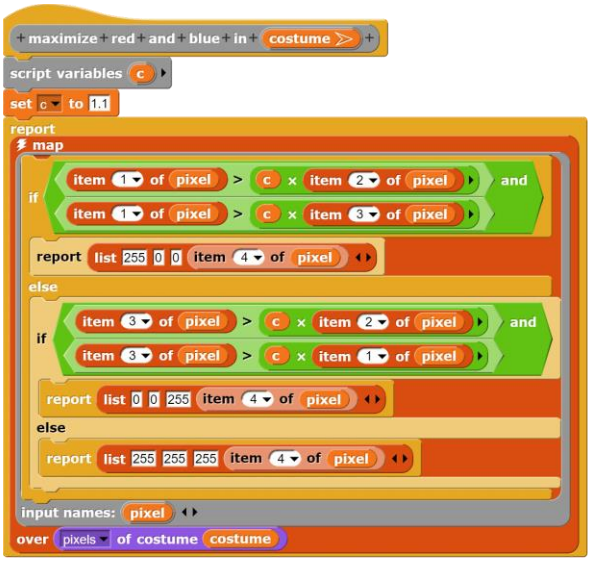
One can disagree on what one means by "predominantly" red or blue. The version of our astronomer for *maximize red and blue* is:

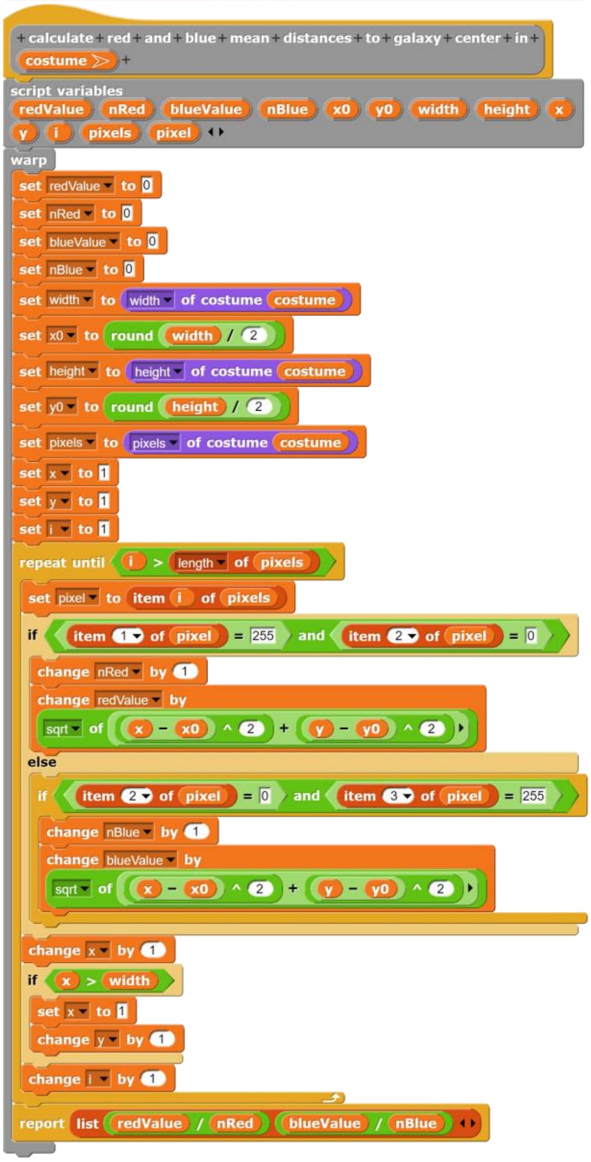
If the red value (item 1) of a pixel is both greater than the green value (item 2) and the blue value (item 3) (including a factor), then return a pixel in full red, correspondingly a blue pixel, and otherwise white.

One of the higher functions of *Snap!* (*map ... over ...*) is used in precompiled form. The result is thus determined very quickly.

The function *calculate red and blue mean distances ...* first determines some initial values as well as the image dimensions and the pixels of the image. Then it calculates the distances to the center for all pixels of the image. It returns as result a list with the two mean values.

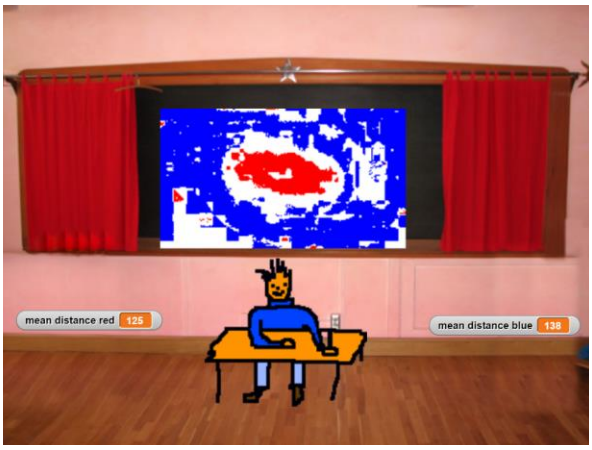
The astronomer transmits these two functions to the student "by mail".



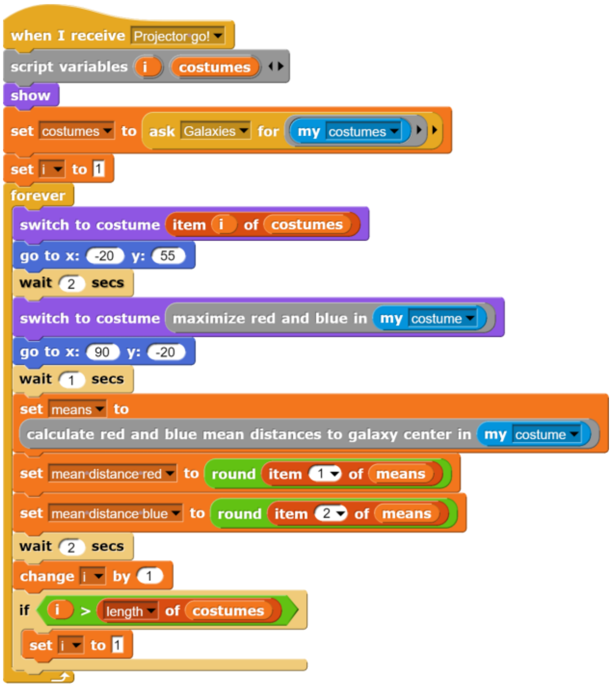


[30] Because Snap! currently cannot exchange scripts directly between scenes.

After receiving the mail, our student asks the projector to apply the two operations to all galaxy images and to display the results. This is what he does.



So far, the informatics part.

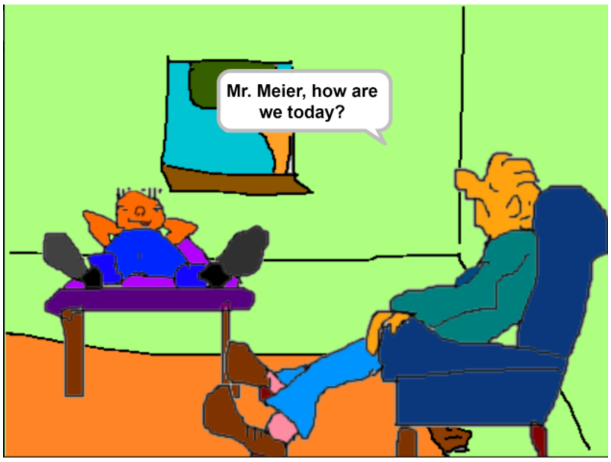


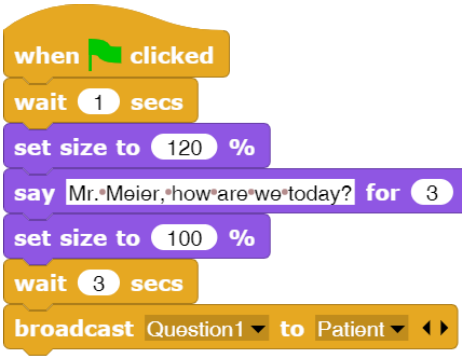
If we remember that the astronomer sent the pictures with the small additional tip as an answer to the question about the old stars, then this question was not answered so far. The student has indeed found out, first by eye, then confirmed by a small program, that there are more red than blue shining stars inside the galaxies - but he didn't want to know that. He can now react to the situation in (at least) two ways: either he thinks the astronomer is an incompetent teacher who does not take him and his questions seriously, and leaves the distance learning program in a huff, or he trusts the astronomer and concludes that the old stars are the red ones. But this additional conclusion has only partly to do with the transmitted data, facts about it are completely missing. It results essentially from the context and the situation of the involved persons.

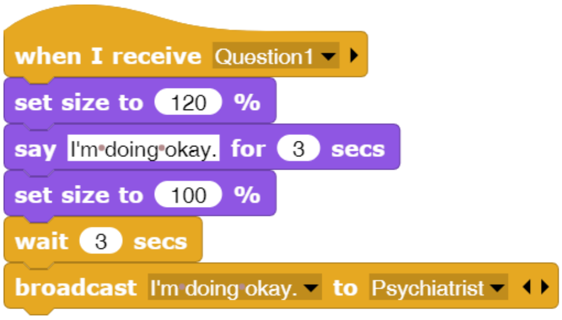
Weizenbaum's Eliza[31]

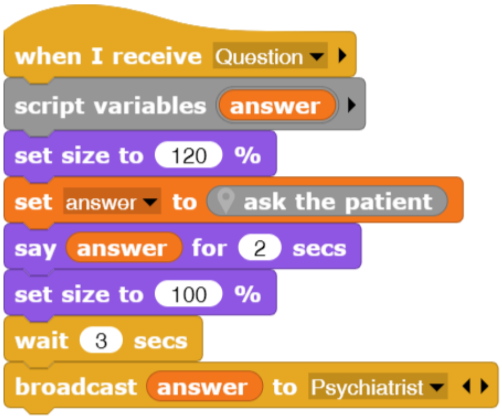
Level: *high school* Materials: *Eliza*

The famous example describes the communication between psychiatrist and patient, where (in this case) both randomly spout platitudes. The coordination of the "conversation" happens again by corresponding messages.









Besides the playful character of the example, the information content of the messages is also interesting. Patient and doctor do not react to each other at all in terms of content, but they send data at the right time. So, what is the transmitted information? If anything, the patient learns that someone is there to talk to. Maybe that helps him. But this information is not represented by the transmitted data, but by the presence of data (garbage). The data itself is irrelevant. Or, to put it another way: any data can be transmitted, because it are not they that carry the information, but the context that gives them all the same meaning.

[31] https://de.wikipedia.org/wiki/ELIZA

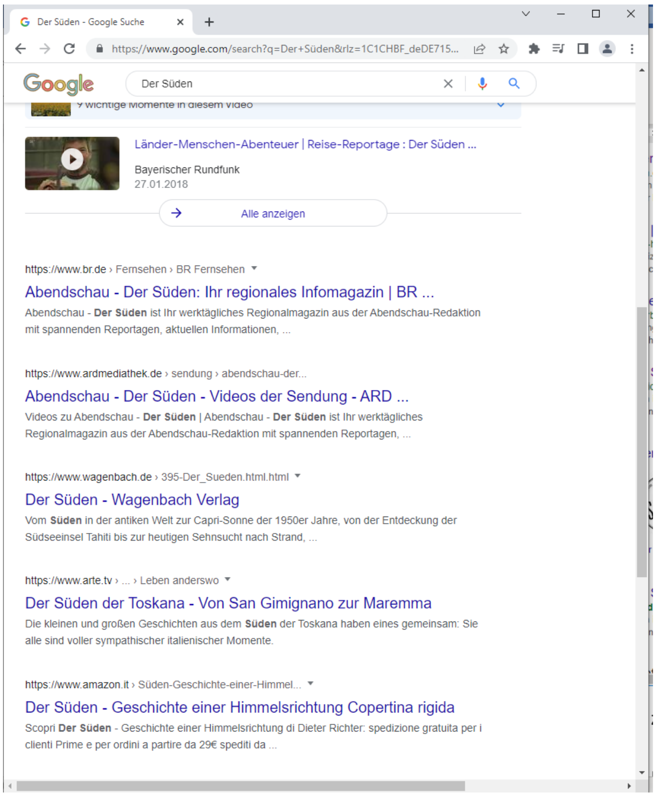
Reporter to determine random responses of psychiatrist and patient:

3.2 Examples for Communication with a Clear Question

In this scenario, a human partner communicates with an IT system. If he wants to have appropriate answers, then he must formulate his questions accordingly.

The Knowledge Society

Level: *high school*
Materials: *Results of the research* [32]



We often hear that it is no longer necessary to acquire knowledge because it is "on the net" and can be accessed at any time.

Is that right?

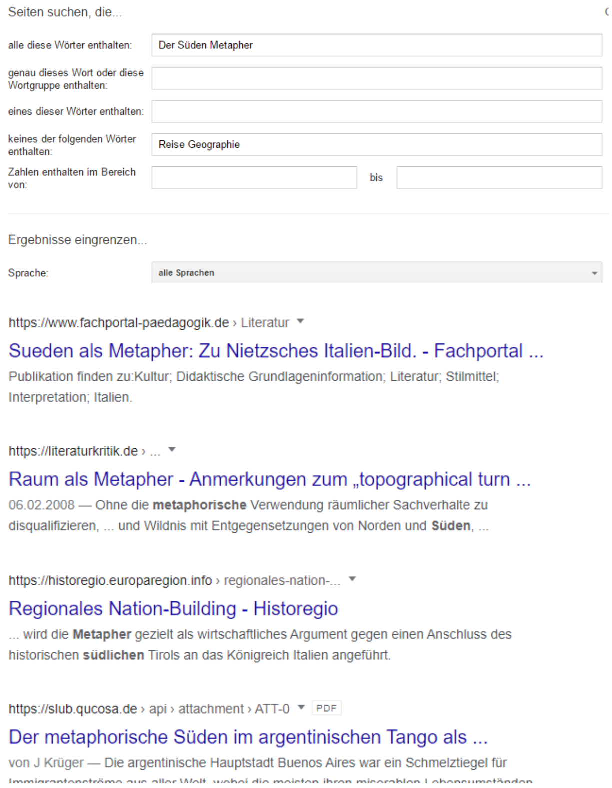
As always, we take an experimental approach and try to find out about a non-trivial concept: the South. The answer is the "most relevant" of the 47,400,000 results. The usual Wikipedia entries are supplied, e.g. on the book by Borges ("*El Sur*"), on television programs, films and the cardinal direction, as well as references to travel literature - and to other books on the subject. We don't find much more on the following pages either. So, we have to conclude that "the south" is to be understood geographically alone - or we have to bite the bitter apple and read real books. There seem to be a few of them according to Google's opinion. If we reject this evil suggestion, then we are left with the geographical South. References to the South as a metaphor, social or economic phenomenon, narrative element, place of longing, literary category, theme of visual art, etc. are missing, and we won't even miss them unless we knew they existed.

We can find facts "on the Net": the population of Hamburg or the gross national product of Burkina Faso, the recipe for Frutti di Mare or for repairing the vacuum cleaner. Information can be gained from these facts if we evaluate and classify them appropriately. But what do we "classify"? Only existing knowledge comes into question for this, knowledge that exists in the mind, and that must first be acquired before "the network" can be used appropriately.

If we ask for the results of such ordering processes, i.e. the evaluated data, then we naturally also receive answers: the opinions of others. But we can only evaluate these opinions, i.e. we can only classify them ourselves, if we have the corresponding abilities (see above). If these are missing, then other evaluation criteria remain: that we believe the opinion leaders (or not), that they are sympathetic to us (or not), that they are like us (or not), that others believe them (or not) ... - if we believe that the others are who they claim to be (or not). This has little to do with rationality.

[32] The screenshots shown are from Google Chrome from 2022/5/1.

If we know that there are other ways of answering our questions than those first supplied by the search engine, then we are off the hook. If we expand our search for "*the South*" to include the term "*metaphor*", we get a completely different spectrum of answers - and there are only 200,000 results. What impoverishment! Even "*the South as a space of desire*" yields 609 answers that have almost nothing in common with the previous ones. Only the combination with the fine arts yields again some hundred thousand hits. If we specify our query by using advanced setting options or already know how to exclude terms, for example, then the search results slowly come closer to what we expected from them. Again, existing knowledge provides access to new knowledge.

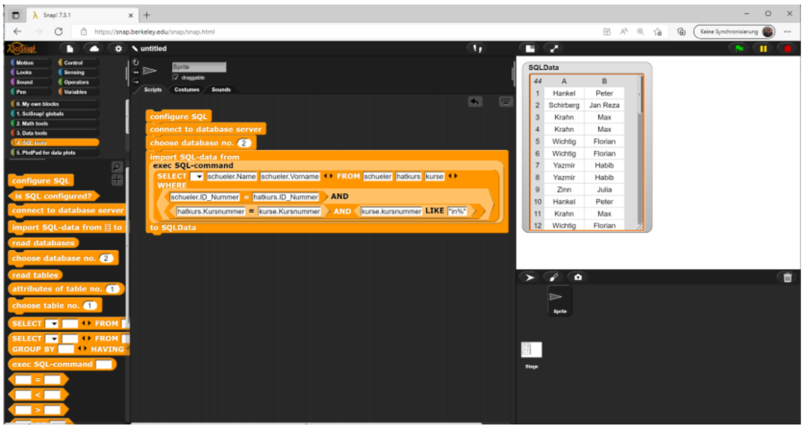


Search engines are not spiteful, they just work "as intended". If we ask precise questions, they usually provide precise answers. If we do not ask precise questions, then they need additional criteria to find "the best" answers. These criteria can be paid rankings, but most often they are "ratings" of the answers by other users who have asked the same or similar search queries, or other stored data. The rating is known to take place in the form of a click on the answer line.

This behavior has consequences. No one can sift through the 47,400,000 answers mentioned at the beginning, and even the 609 hits of the longing space are almost unmanageable. So almost all clicks will be on the first one or two pages of the search results - and thus we have a self-reinforcing process: the most clicked pages will again be the most clicked, further consolidating their ranking. The others are present, but practically invisible. "On the net", users will permanently see only pages whose content corresponds to those that were initially offered to them. New ones will hardly be added. If, for example, the initial search queries have been filtered by the provider, supplying him with the results of "similar" users who can be easily found based on his previous use of the system (or, increasingly, "the net" as a whole), then the user will hardly leave this "information space" again. He simply does not see anything else. On the part of the provider, this behavior is understandable, because he wants to deliver search results that are highly likely to be clicked on - only then he will get paid. But the resulting "echo chambers" are politically explosive, for example, because they divide society into disjunctive groups that are hardly capable of discourse, but also impoverish the spectrum of "information" that would otherwise be provided.

The result of our considerations is quite clear: "The net" does not contain knowledge, but data. These can enrich our knowledge if we have the knowledge to use them appropriately, to evaluate them, to classify them or to discard them. Appropriate education forms the basis for wonderful new possibilities. If it is missing, we become manipulable objects.

Access to Databases



Level: *from middle school*
Tool: *SciSnap*[33]
Materials: *SQL example*

We use one of the libraries of *Snap! (SciSnap!)*, which among other things offers the possibility to access databases. This allows us to compose the usual database queries from the associated relations, attributes, etc. using blocks and have them executed. The result in each case is a list. In the example shown, we get the participants of basic computer science courses. Quite simple.

Why is it so easy - and for whom? The user must know the SQL syntax and stick to it. Above all, he must describe the desired data completely unambiguously, there must be no possibility of interpretation. This is not so easy. However, the evaluation of such queries is then easy for the machines, they have been able to do that for years.

Our user describes with his request the data he wants to receive from the system. He usually does not know exactly which data he will receive, but he knows their meaning. He knows them, not the machine. The machine cannot know them at all because it cannot know which meaning the user gives to these data, which information he will take from them. The list of our students, for example, can mean many things: maybe they have to be excused from the rest of the day's classes because of a field trip, maybe they check which small courses can be cancelled, maybe they check if the books are enough for the course. One does not know ...

SQL queries are easy for machines to evaluate because they provide a clear basis for decision-making: Data either belongs to the requested category - or not. It gets interesting when questions are asked that do not provide a clear basis for decision-making. Should such fuzzy questions as *"Are boys (or girls) disadvantaged at school?", "Do urban children (rural children, children from middle-class homes, members of sports clubs, kindergarten children, children with a migration background, ...) have it easier (harder) at school?" "Is the evaluation of different competencies at school fair?", "Who is the best teacher?"* are answered with the help of (e.g.) SQL queries, then a reformulation and thus an interpretation must inevitably take place, which leads to answers for which it is at least questionable whether they answer the original question or just the interpretation. There is always an answer when a query has been formulated, even if the original question cannot be answered based on the data.

---

[33] SciSnap! is a Snap! -library.

ACCESS to JSON[34]-Data

Level: *high school*

Materials: *JSON example*

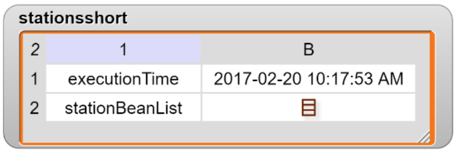
stations.json (contains the data of bicycle rental stations in New York)

stationsshort.json (shortened version of the file)

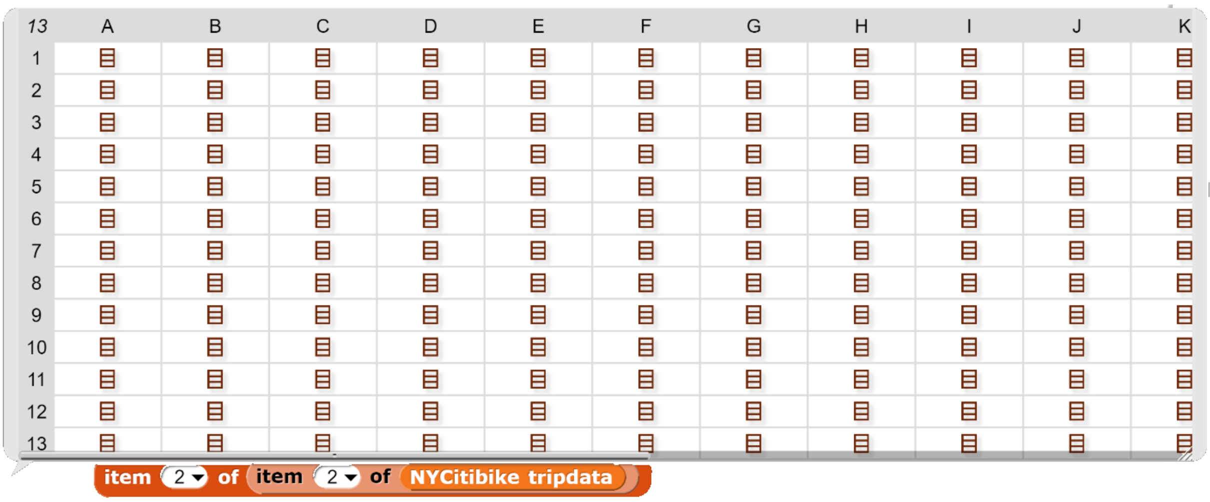
JSON is a string data structure that is stored as a file. *Snap!* can import such files and convert them into a nested list, as well as convert lists into *JSON* format and export them again using the *<length> of <list>* block, if necessary. So, we load the *JSON* data on our computer and import it directly into a variable we created before. It is even easier if we simply drag the *JSON* file onto the *Snap!* window. In this case, a variable is created with the file name and filled with the file contents.

In the project, current and freely accessible data that have been saved as *JSON* files are to be evaluated. To do this, the learners must first research what *JSON* is, what structure the files have, what data types can be represented in them. As an example we choose the data of the bicycle rental stations in New York, which are available in a file *stations.json*.[35] In this case, the target of the transformation is a structure that contains either an atomic quantity (*logical value, number, string, ...*) or a list that consists of atomic quantities and/or partial lists that contain the type of the original data (list or dictionary) as the first entry. In dictionaries, two-element lists with key/value pairs follow as further elements.

In our case the result of the data import looks a bit disappointing:



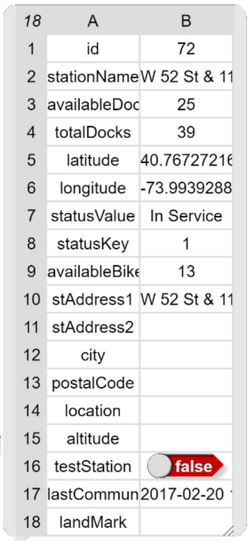
We therefore take a closer look at the second element of the second line of the list - there seems to be something hidden.



34 JavaScript Object Notation

35 https://catalog.data.gov/dataset/citi-bike-live-station-feed-json-d1c27

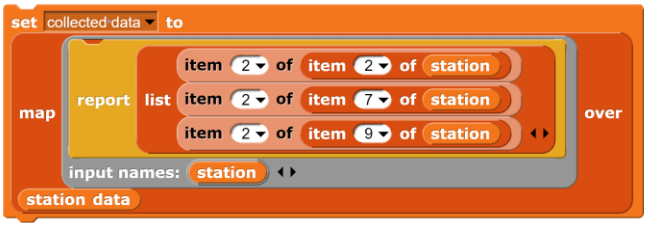
This already looks more interesting. Let's see what the first line contains:

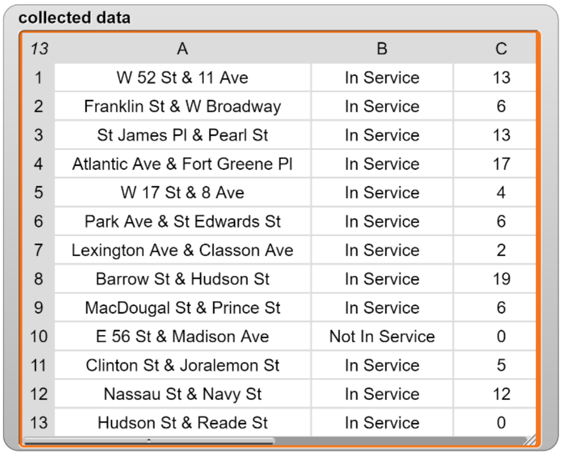


18	A	B
1	id	72
2	stationName	W 52 St & 11
3	availableDoc	25
4	totalDocks	39
5	latitude	40.76727216
6	longitude	-73.9939288
7	statusValue	In Service
8	statusKey	1
9	availableBike	13
10	stAddress1	W 52 St & 11
11	stAddress2	
12	city	
13	postalCode	
14	location	
15	altitude	
16	testStation	false
17	lastCommun	2017-02-20
18	landMark	

So, these are the station data we are looking for. We therefore store these elements in a new variable.

From this data, a table is now to be created that contains only the columns *Station name*, *Status* and *Available bikes*.





collected data

13	A	B	C
1	W 52 St & 11 Ave	In Service	13
2	Franklin St & W Broadway	In Service	6
3	St James Pl & Pearl St	In Service	13
4	Atlantic Ave & Fort Greene Pl	In Service	17
5	W 17 St & 8 Ave	In Service	4
6	Park Ave & St Edwards St	In Service	6
7	Lexington Ave & Classon Ave	In Service	2
8	Barrow St & Hudson St	In Service	19
9	MacDougal St & Prince St	In Service	6
10	E 56 St & Madison Ave	Not In Service	0
11	Clinton St & Joralemon St	In Service	5
12	Nassau St & Navy St	In Service	12
13	Hudson St & Reade St	In Service	0

If the data are available in list form, as is the case here, then their relevant part can easily be extracted and evaluated - but what is "relevant" in this case? Of course, this depends on what is to be done with the data, what information is sought. If we are interested in the number of available bicycles during the week, then the evaluation will be different than if we want to show the distribution of bicycles over the city in a city map. And maybe we are just looking for a free bike near the hotel.

And couldn't we have just left the whole thing to an SQL query? We would have if the data were present in an SQL database. But in this case, they are not. So instead of giving an exact SQL description of the data we are looking for and leaving its evaluation to the SQL server, we take action in this case by formulating the exact description algorithmically. The result is the same. In any case, it is up to the human user to describe his or her wishes so precisely that the machine receives a unique sequence of instructions that it needs to compile the answer.

3.4 Communication without Human Partners

For this scenario, we need examples where data is collected by one system, transmitted to another, which may well be running in the same computer, and evaluated there. The results of this evaluation are discussed.

License Plate Detection

Level: *from middle school* Materials: *License plate detection*

We want to deal with the wide field of character recognition, i.e. extracting text from an image. As an example, we choose license plate recognition, as it is practiced e.g. in the toll barriers at highways. We choose here the simple case, suitable for the intermediate level, that we are only interested in the nationalities of the vehicles.[36] Thus, we only need to recognize the characters in the blue area of the European license plate. Images for such tasks can be generated quickly and easily on the Internet.[37]

We choose a very simple approach and hope that the numbers of pixels to represent these symbols differ. This reduces the problem to the task of finding the blue area and counting the non-blue pixels in it. To be independent of the size of the representation, we compare the pixels in the upper area ("Euro stars") with those in the lower one. For simplicity, we work with global variables to transport data between the different blocks.

First of all, we generate some license plates of different nationalities, import them as costumes and write some methods that solve the subtasks.

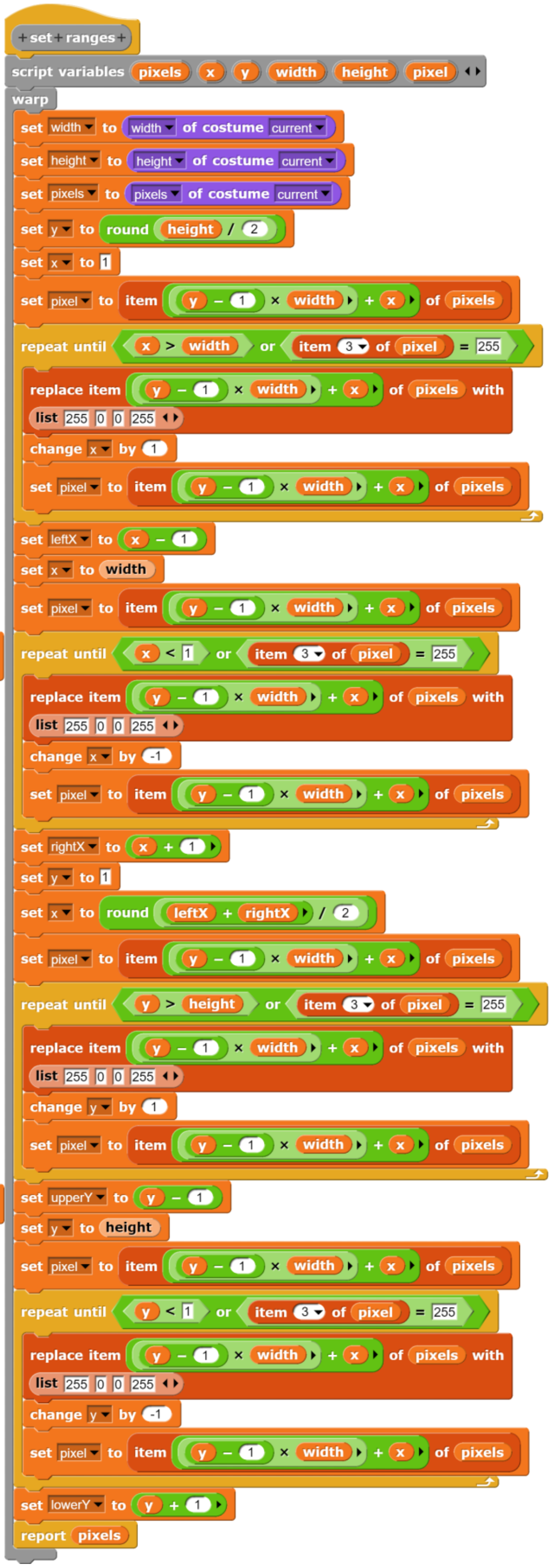
We have to find at least the blue area in the license plate. We have already done something similar with the galaxy images, for example. With the RGB limits we have to experiment a bit, then it works fine. Since an image has quite a lot of pixels, we work with the compiled version of *map...over*. Then the recognition is very fast.

So far, we have the following partial result.

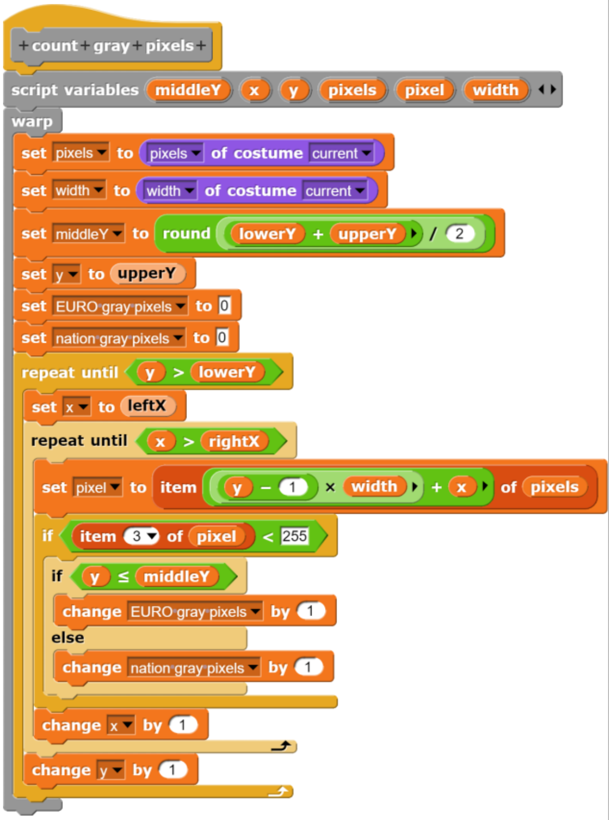
[36] We can find a more detailed example on http://snapextensions.uni-goettingen.de/beispielsupermarkt.pdf. There, in addition to character recognition, simple approaches to face recognition, etc. are also implemented.

[37] For example, you can simply search the keyword "license plate".

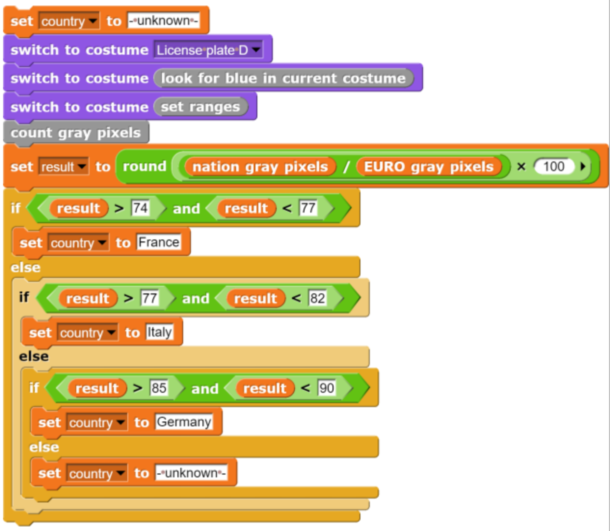
The boundaries of the blue area can be easily found if we search it starting from the left or right, top and bottom. For clarity we mark the examined pixels in red. The corresponding script is simple.



After that the borders of the blue area are known as global variable *leftX*, *rightX*, *upperY* and *lowerY*. In this area we can now count the inner non-blue pixels, here separated into upper EURO- and lower national-area.



We receive as a total script for the recognition of the nationality of EURO license plates:



With these results, we can now investigate, on the one hand, whether the initial solution approach was useful at all, and if so, from which country the license plate under investigation originated.

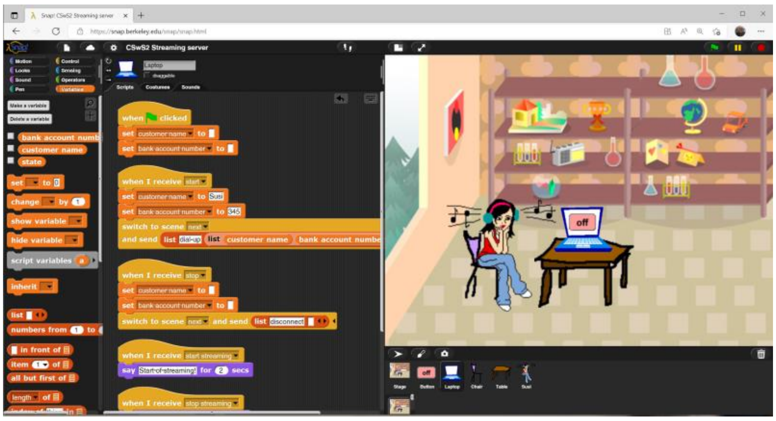
So much for the "technical" part. We can now easily imagine that the remaining part of a license plate can also be determined with a little effort. The result of this process is then transmitted to another location and evaluated there. This can be toll booths, police computers, ... We will first deal with the rather "uncritical" case of a toll station.

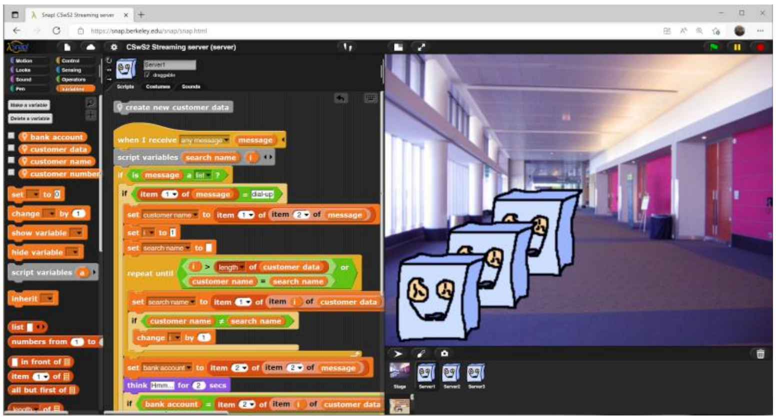
Our license plate reader therefore reads license plates and transmits the result to the control center - as data, e.g. "ABC DE 123". This data is evaluated there by the running program, as if it were information. In this case, it is assumed, for example, that the vehicle with the specified license plate number is on the Brenner freeway. If the relevant conditions are met, then the toll can be debited from the car owner's account. What happens if the owner objects to the debit because he allegedly did not drive over the Brenner, but was swimming at the Kochel lake? Human eyes for both are not found, only "the computer" thinks to have identified the license plate on a picture. Is this case justiciable? Probably not, because computers are not recognized as witnesses. Probably, in this case, the original image that the computer evaluated will also have been stored, so that human experts can check whether the machine was mistaken. However, it is easy to give scenarios where this verification does not take place or is not even possible, e.g., because the person concerned does not know about his "identification". The movement data of a cell phone may serve as an example, for which there are very different interested parties.

So, even in this case, the data transmitted by the image analysis has little to do with the information that is derived from it. If this interpretation is outsourced to machines, then we quickly arrive at scenarios that are to be located in the area of "computer science and society".

Streaming

Level: *from middle school*
Materials: *Streaming*

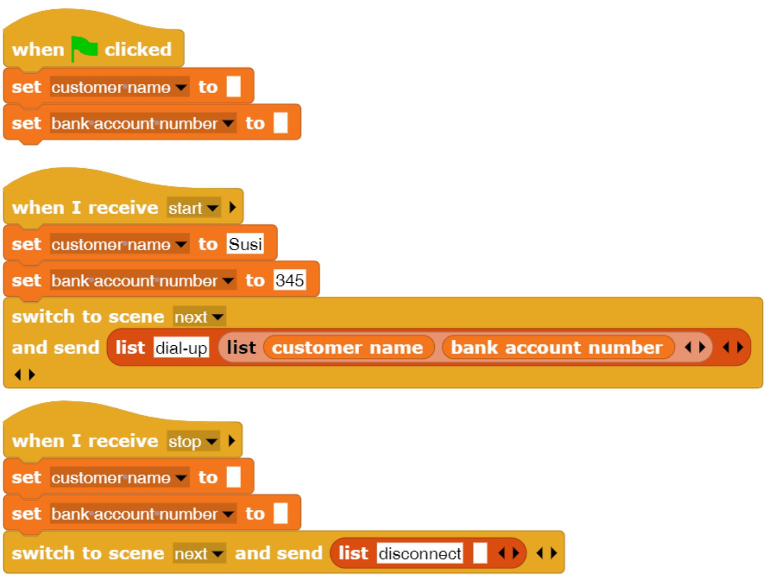


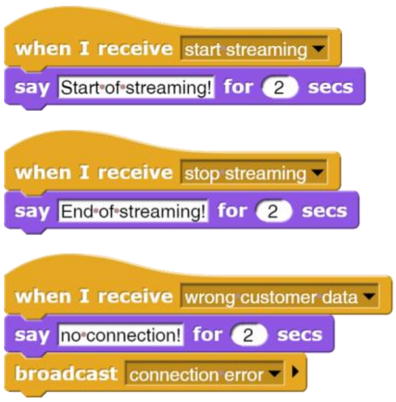


We use a project with two scenes, one serving as the room where *Susi* wants to listen to music, and the other as the server room of the streaming service provider. On the server side, we maintain a list of customer data that allows logging in and perform billing (not realized in the example) that depends on the usage time. If the user account is empty, then the connection is shut down. On the client side there is *Susi* and her laptop. The laptop establishes the connection when the power button is pressed and terminates it when the button is clicked a second time.

Of course, the students have to set up the accounting system on the server side first. However, their main problem should be to establish a secure connection between the server and the client, where the transmitted data cannot be read. Since there are very different solutions for this, this is a highly differentiating task.

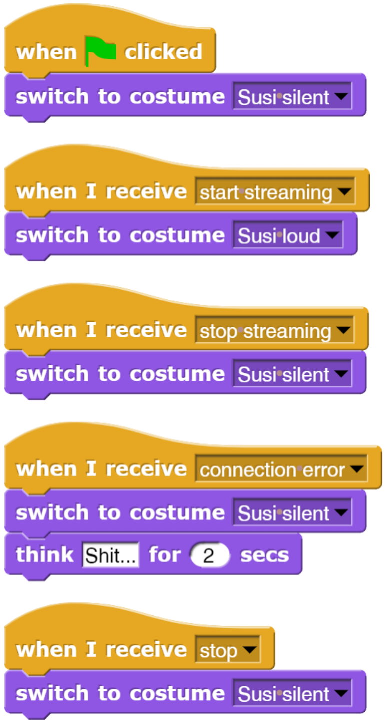
We implement the solution here very simply without encryption via messages and scene changes. The laptop is responsible for establishing the connection, for example. It does this via scene changes to which it attaches a message (as a list).

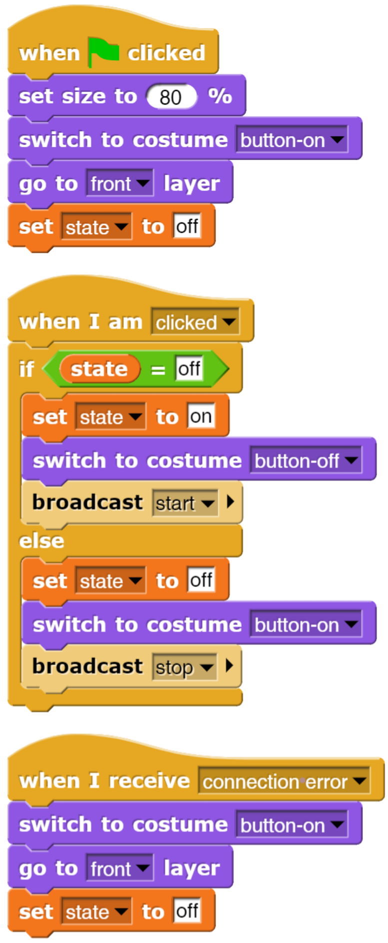




The button on the laptop simply controls switching on and off.

And *Susi*? She doesn't really do anything - except change costumes. She doesn't want to do anything; she wants to chill!





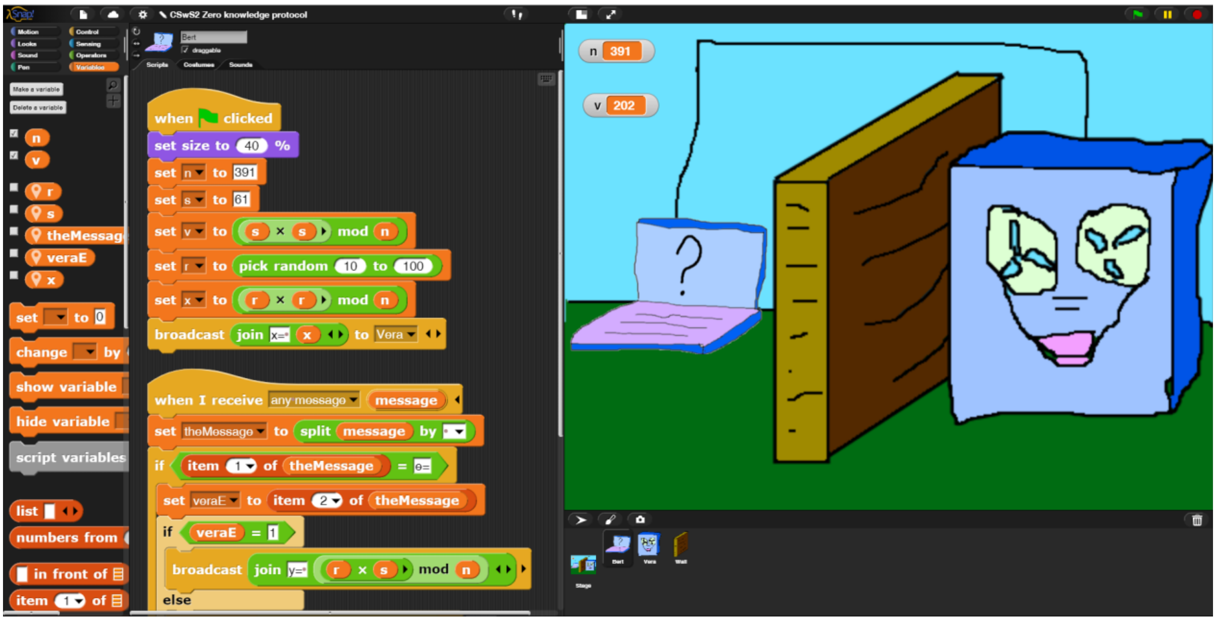
The streaming server receives a message in the form of a list when the scene changes. The first entry contains either the user data, which is then evaluated by the server, or the request to stop streaming. And initially it sets up a list with customer data.

As you can see, the previous scripts are trivial. However, the solution can be greatly extended in very different ways.

The data transmitted during the LogIn process should contain the information that the user (in this case) is *Susi*. Obviously, this information may or may not be correct. So, the data alone does not determine the information content, but the whole context is important, e.g. its security aspect, on which depends how far the data is to be trusted.

Zero Knowledge Authentication

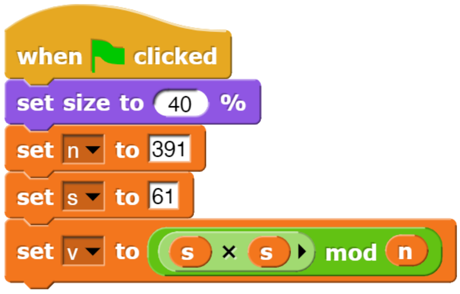
Level: *high school* Materials: *Zero knowledge protocol*



The idea of the zero-knowledge protocol[38] is that a prover ("*Bert*") has to prove to a verifier ("*Vera*") that he has certain information (the *key*) without the prover communicating this key over the network. For this purpose, Vera poses tasks to the prover, the solution of which can only be guessed with a certain probability *p*. The verifier has to prove that the verifier has the key. If $p = 0.5$ and the number of questions $n = 10$, then this question chain could only be answered correctly by guessing with a probability of $(0.5)^{10} = 0.00097$.. If we choose *n* higher, then the answering system can be authenticated with virtually any degree of certainty. We choose a simple version of the *Fiat-Shamir protocol*, which runs as follows:

Presupposition:

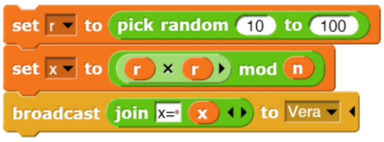
Bert determines a large number n as the product of two large prime numbers: $n = p*q$. Then he chooses a number *s* which is partially alien to *n* and calculates $v = s^2 \bmod n$. He publishes *n* and *v*. In our case, this is done by assigning values to two global variables.



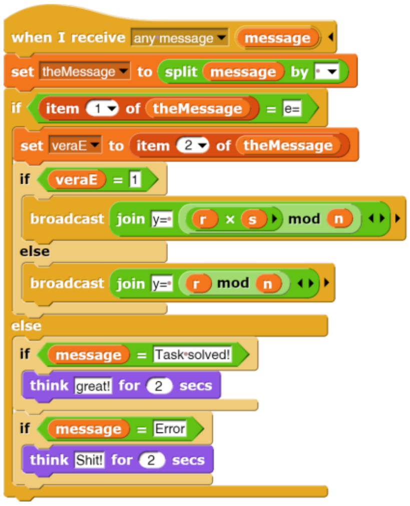
[38] https://de.wikipedia.org/wiki/Fiat-Shamir-Protokoll

For authentication, the following steps are then run through several times:

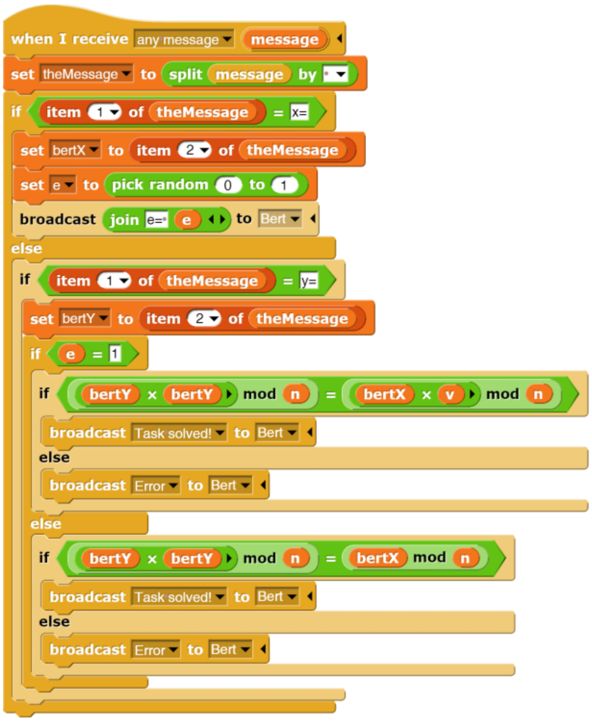
1. Bert determines a random number *r* and sends $x = r^2 \ mod\ n$ to Vera.



2. Vera remembers *x*, determines a random bit *e* (0 or 1) and sends it to Bert.
3. Bert calculates $y = r*s^e \ mod\ n$ and sends *y* to Vera.



4. Vera checks if $y^2 \ mod\ n = x*v^e \ mod\ n$ and reports the success or failure.



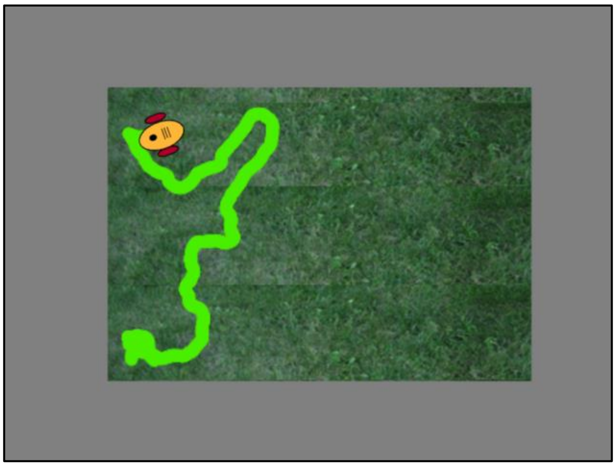
In this case, too, data is transmitted between the communication partners. However, their content does not result from the transmitted values, but from their coherence within the framework of the protocol, which goes beyond the pure exchange of data. Thus, it is not about the data itself, but about its property of being "correct".

4 Simple Examples

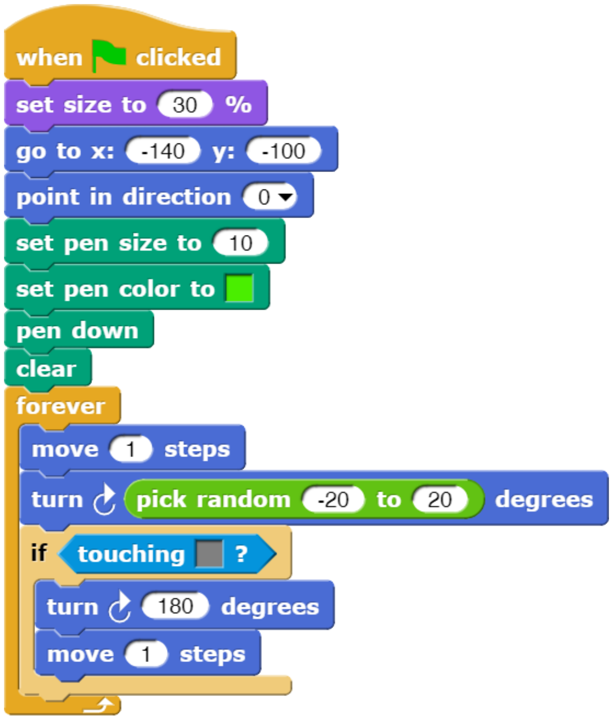
The following examples each demonstrate a few aspects of *Snap!*. They are quick to implement and should encourage modifications and extensions. Above all, they show how easy visualization is in *Snap!*.

4.1 A Lawn Mower

Level: *from middle school* Materials: *Lawnmower*



We provide the stage with the costume of a lawn with a gray border, so we turn it into a modern lava stone garden. For the sprite, we draw a lawn mowing robot as a new costume. The robot is supposed to mow the lawn, and it can do that in very different ways. We realize a simple one that runs only randomly. The robot overwrites the lawn background with the light green color of a freshly mowed lawn.



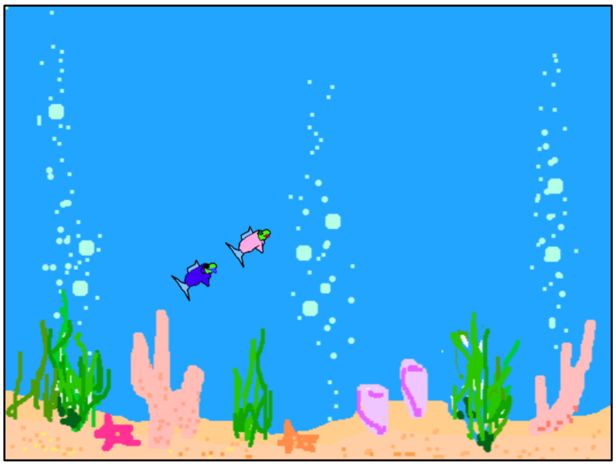
Nevertheless, the task is not trivial. For example, is the lawn always completely mowed on the inside? What about the not mowed strip at the edge? Are there more suitable robot movements for mowing? Which one does it the fastest? Is it possible to install a timing system? Where is the robot's charging station and when should it approach it? What happens to the plants in the lawn, e.g. spring tulips? Will the lawn grow back?

It can get as complicated as you like - and there are very different possible solutions to all questions.

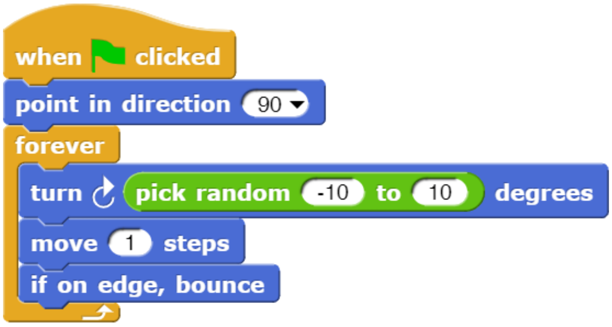
4.2 In the Aquarium

Level: *from middle school* Materials: *Aquarium*

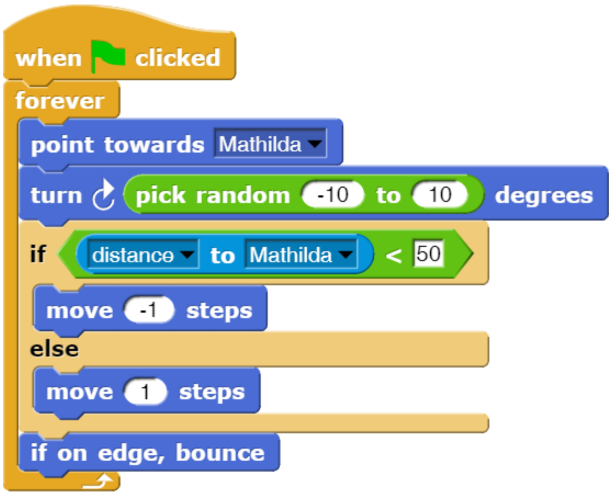
We will look for a nice background image for the stage (or draw one) and import it as the costume of the stage. Then we create two sprites using the button Sprite-Coral and name them *Mathilda* and *Joe* - whatever else. For each of them we draw a fish costume.



Mathilda has little interest in other fish and swims around the aquarium independently of them. If she meets a wall, then she turns back.



Joe is more interested in Mathilda than the rest of the aquarium. He constantly looks in her direction and swims towards her. If he gets too close to her, he carefully keeps his distance. He has his own experiences.



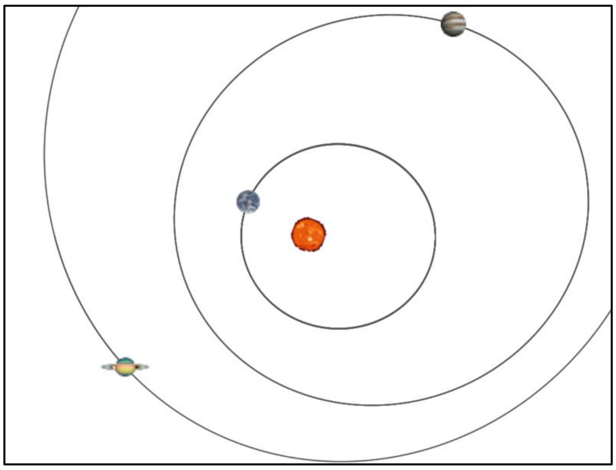
We can very easily create more fish, forming a chain altogether, as can sometimes be observed in large marine aquariums. The introduction of a shark is also simple. It swims towards other fish, but if it gets too close to them, then they quickly run away.

More demanding is a real schooling, where many fish build a common structure. The strategies for this can be found in the net[39] and are also well realizable.

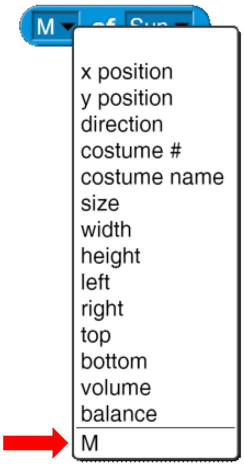
[39] e. g. in https://de.wikipedia.org/wiki/Schwarmverhalten

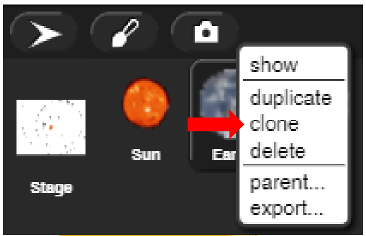
4.3 The Sun System[40]

Level: *high school* Materials: *Solar system*



If there is a sun of mass *M* in the origin of the coordinate system, the gravitational force on a planet of mass *m* is $\vec{F} = -G \cdot \frac{m \cdot M}{r^3} \cdot \vec{r}$, so $\vec{a} = -G \cdot \frac{M}{r^3} \cdot \vec{r}$. The value of the local sun variable *M* is obtained by other sprites using the *... of ...* block (see figure).



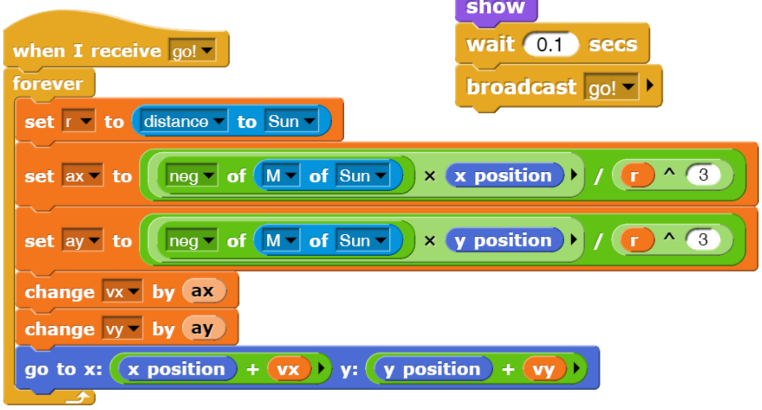


We get an image of the sun and some planet images from the web and scale them down a lot. Then we load them as costumes into a planet prototype sprite called *Earth*. A second sprite called *Sun* clears the screen and starts the simulation. Other planets are created by cloning *Earth*.

Our Earth has a set of local variables that describe its state. These include the velocity components *vx* and *vy*, the acceleration components *ax* and *ay*, and the distance from the sun *r*. The velocity values are each set appropriately at the start of the simulation by clicking the green flag.

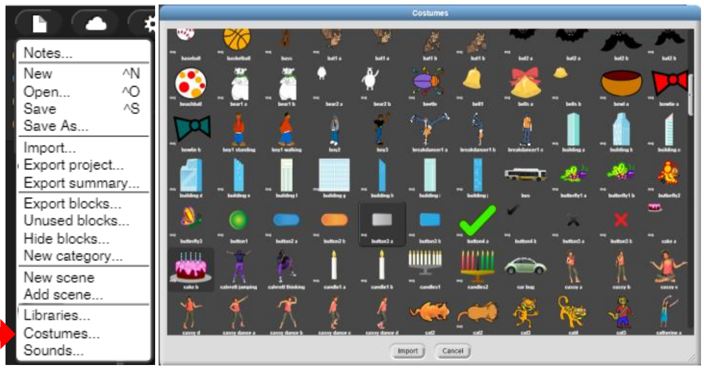
The Sun clears the screen, sets its mass, and shows itself in the center. Then it waits for a moment so that the planets have enough time with the self-initialization and starts the simulation by sending the message *"go!"*. After that the Sun stops its activity.

The planets react to the message by measuring their current distance to the sun. Then calculate the acceleration components *ax* and *ay*. These change the corresponding velocity components *vx* and *vy*, and from this the new planet position can be calculated. These processes are repeated continuously and result in the planetary orbits. All values were chosen in such a way that the orbital curves fit at least partially on the screen.



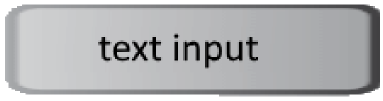
[40] In a rather simplified version: the sun is nailed to the center and the planets do not influence each other.

4.4 Caesar-Encryption



Level: *from middle school*
Materials: *Caesar encryption*

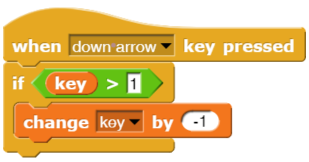
We want to encrypt and decrypt simple strings using the Caesar method. Since this is hardcore computer science, we need a very serious, somewhat boring interface with a few buttons on it. We import these from the *Costumes* library using the File menu.[41] We export the button image to a file. Using a graphics program, we stretch it a bit and label it differently. We import the resulting costumes again.

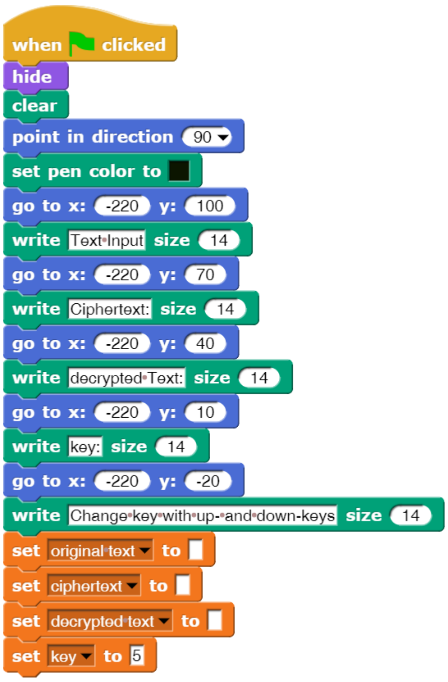


We create three new empty blocks named *text input*, *encryption* and *decryption* and make our buttons react by calling one of these blocks when clicked. To do this, we copy the button twice using the context menu in the sprite area and change the costumes and called blocks accordingly. We drag the buttons to the right place, change their names to *bTextinput*, for example, and uncheck the *draggable* box. Now the button is fixed.

Then we create four global variables called *original text*, *ciphertext*, *decrypted text*, and *key*, as well as a new sprite called *Control*, which makes for a very serious interface. To do this, it writes some text on the stage. We display the four variables on the screen with monitors (put a check mark in front of the variable names) and switch to the large display using the context menus in the display area. Then we drag them to suitable places behind the texts.

Lastly, we enable the change of the key with the help of keystrokes.

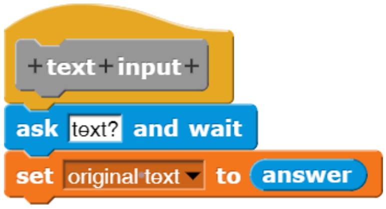




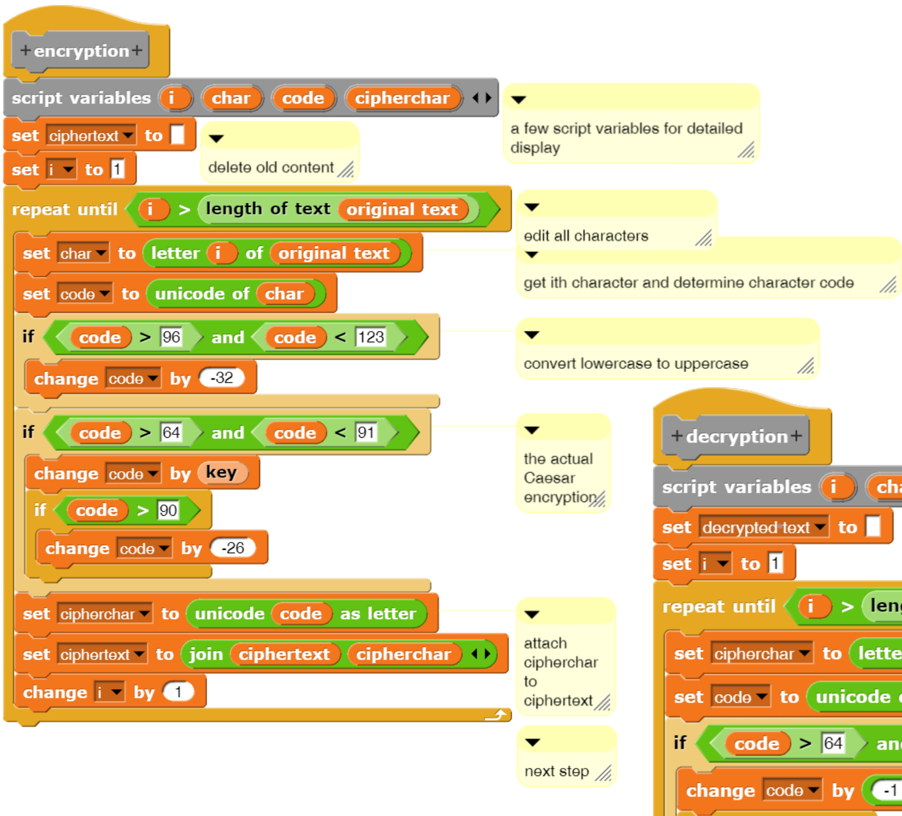
[41] As you can see, there are also far more "interesting" costumes in the library!

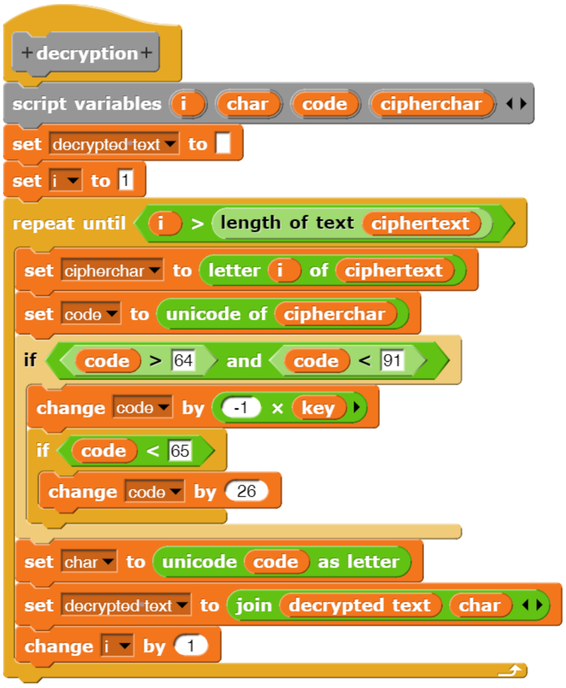
With this, our "user interface" is ready.

We now come to the actual functionality, which can be developed independently. The text input is simple: we just ask for the original text. The output can also be made more beautiful in the process.



Caesar encryption consists of shifting all characters in the code (here: in Unicode) by the key length. The last characters are shifted cyclically to the front. In the script below, this is done very verbosely, but - hopefully - readable. Note that the green *length of text <string>* block from the *Operators* palette works with strings, while the brown *<length> of <list>* version from the *Variables* palette works with lists.

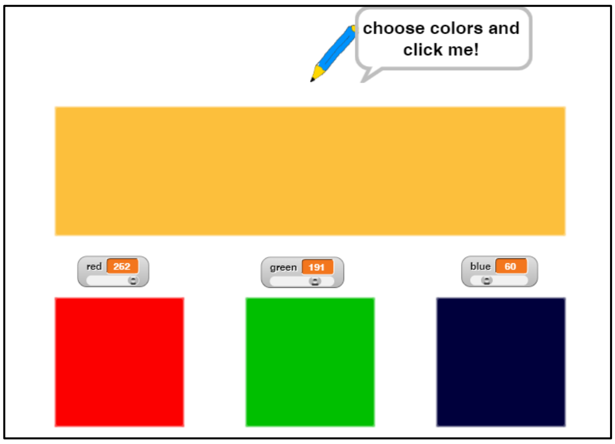




Decryption is performed directly inverse to encryption.

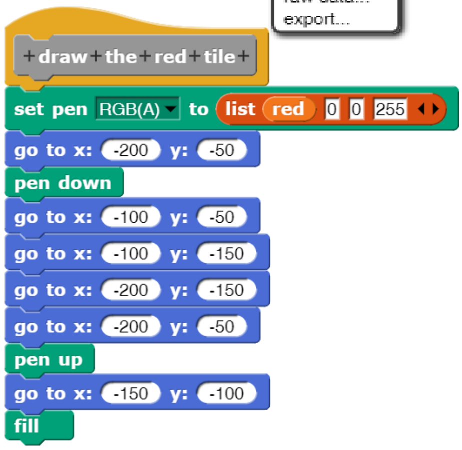
4.5 Color Mixer

Level: *from middle school* Materials: *Color mixer*



We want to create and display mixed colors from the primary colors red, green and blue in the usual way. For this we have to know that colors in the RGB model are represented by 4-element lists, which contain the three color values and additionally the "transparency" of the mixed colors, i.e. their opacity. All four values come from the number range 0 to 255, so each can be stored by one byte. If the pen is to draw in full red, then this is achieved by the adjacent block.

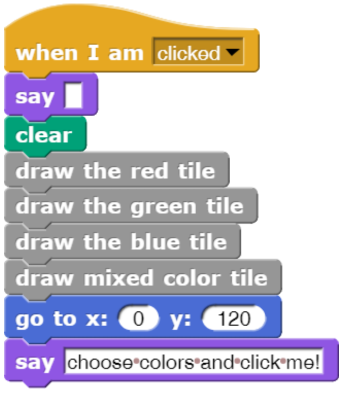
So, we create three variables with the identifiers *red*, *green* and *blue*, display them on the stage (place a check mark in front of the variable name) and place them in a suitable position. Then we select the *slider* item from their context menus so that a slider appears below the variable value, and then set the value for *slider max* to *255*. This allows us to select the desired color values simply by moving the slider.



After these preparations we can have an appropriately colored square drawn under the red variable, and with slightly changed coordinates also the green and the blue. And of course, as the crowning glory of the project, a larger rectangle with the mixed color belongs over everything.

The drawing process is started by clicking on a drawing pen, to which we also assign a suitable costume for this purpose.

If we work together on the basic framework of this project, which includes a red rectangle and the reaction to the pen's *OnClick* event, then the missing three color areas can be created by the learners themselves in direct analogy. And of course, there is room for improvement:



- The colors should change immediately when a color value is changed, not when the pen is clicked.
- The help is also kept very sparse. This can be done better!
- From physics lessons you know color circles, possibly with different transparency, which show the mixed colors. Is this also possible in *Snap!*?
- Are there other color models than the RGB model? Which ones? How do they work?

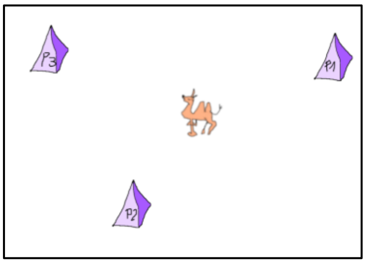
4.6 Tasks

1. a: Find out about **XOR encryption**. Implement the procedure.

   b: Find out about **offset procedures for encryption**. Implement such a procedure.

   c: Learn about **cryptanalysis**. Implement a frequency analysis.

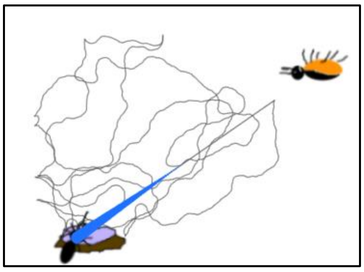
2. In the **camel problem**, the animal gets into a terrible situation between three pyramids. It moves purposefully towards a randomly selected pyramid. When it has covered exactly half the distance to the pyramid, a spiteful desert ghost comes and whirls the poor creature around so that it no longer knows which pyramid it was heading for. The movement, of course, leaves an imprint on the screen, and the procedure starts all over again.



3. The **goat problem** pops up in the media every now and then. It goes like this: In a lottery, there are three doors, behind two of which is a goat, and behind the third is the main prize. The game master, who knows the positions, asks the player to guess one door. Then he opens one of the remaining doors, behind which there is a goat, and offers the player to change his mind - or not. The question is: should the do that? Realize the game and decide the question empirically.

4. a: **Desert ants** live alone in the desert. When they leave the burrow, they search the surrounding area for something to eat. When they have found it, they run directly back to the burrow. Obviously they remember which movements they have made in total. From these they "calculate" the direct way back. Realize the procedure.

   b: On their way to the burrow, the ants should leave a **pheromone trail** that slowly evaporates. On this trail they find their way back to the prey, get another piece and run back to the burrow, leaving a new pheromone trail. If they have found nothing more, they leave no new trail.



5 Simulation of a Spring Pendulum

Level: *high school* Materials: *Spring pendulum*

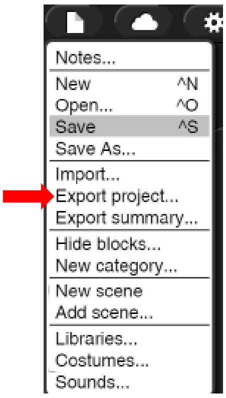
In addition to the extensive freedom from syntax, the excellent visualization possibilities and the good-natured behavior of *Snap!* in case of errors are an incentive for the learners to proceed experimentally and thus to try out their own ideas. An experimental approach opens up possibilities for independent problem solving instead of reproducing predefined results, especially at the beginning.

In the area of simulation, to which we can also count many of the usual games, we find enough simple, but not trivial problems, which can already be worked on by beginners with a little good will. Experimental work of course requires an interest in developing own ideas. So, we need problems that generate enough motivation. As an example, we choose the simulation of a simple spring pendulum, which is suspended from a periodically oscillating exciter. I know that an example from physics is not very motivating for all learners - rather the opposite. But I do not give up hope!

Organization of Cooperation

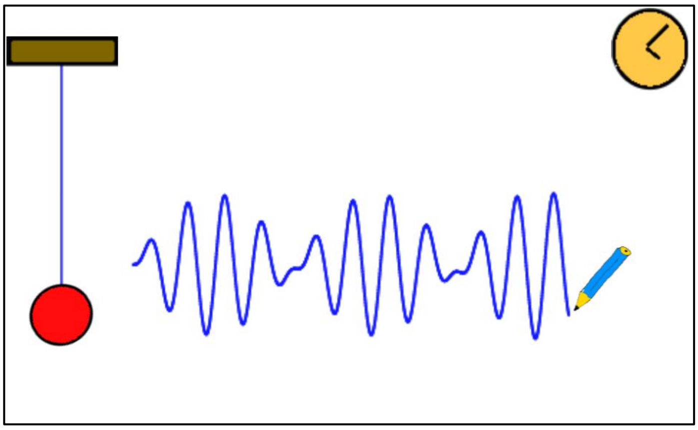
If groups work largely independently of each other, it must be clear on the one hand in which framework work is done, and on the other hand how the results can be compiled later.

To create a frame, you can create empty blocks with the right names as "dummies". These can be used immediately in scripts, still without the functionality sought. The required objects can also be created and given rudimentary behavior, such as responding to events: For example, you can output a speech bubble with an explanatory text: *"Now actually this and this should happen!"* This program frame can be exported or imported as a whole or in parts:



- The project can be exported with all its parts using the File menu. It will then appear at the bottom of the *Snap!* window. Clicking the arrow to the right of it will take you to the download folder where it was saved. From there it can be sent, imported (see picture) or dragged into any *Snap!* window and opened again.
- If there are global methods (blocks *"for all sprites"*) in the project, then another item *"Export blocks..."* appears in the same menu. If this is clicked, then the blocks to be exported can be selected in the window that appears. The additional blocks used in these blocks are automatically added, so that they are available again when the new blocks are imported. The saved blocks can be dragged into open *Snap!* windows, imported or sent like projects.

- Sprites can be exported as a whole with their local methods by selecting the *"export..."* item in their context menu in the sprite area. The re-import is done as described above.
- If images of scripts are saved, then they contain the code of the mapped blocks. If such an image is imported as a costume, then the mapped blocks can be recreated with *"get blocks"* from the context menu of the costume in the script area.
- Within a project, scripts can be transferred from one object to the next by dragging them from the sprite in which they are located at the script level to the sprite in the sprite area that is to be supplied with the script. The addressee will be highlighted a bit when "dragged" if it has noticed that it is meant.



The example of a spring pendulum contains several parts that work largely independently, so that group work with division of the tasks is almost inevitable.

We identify

- an *exciter*, the dark plate at the top-left, which periodically oscillates vertically. Its frequency ω is an instance variable and can be changed in the variable display.
- a *ball* hanging relatively stupidly on a thread, but at least it knows the basic equation of mechanics.
- a *thread* that has to redraw itself again and again so that we don't see any protruding ends on the screen.
- a *pen* that records the diagram of the movement.
- a *clock* for the common time.

The Clock

We create a new sprite and draw a simple clock as its costume. Clicking on the green flag we choose this costume for the clock and send it to the right-top corner. After the clock is started using the *start* message (green flag clicked), it resets the *timer* built into *Snap!* and continuously remembers the current time in the variable *t*, which we also display.[42] Since the time *t* logically belongs to the clock, we declare it as a local variable. Local variable is accessed from other objects via the *<attribute> of <object>* block of the *Sensing* palette. We export the clock sprite to the *Clock.xml* file as specified.

Extension: Let the sprite show the time by moving the hands correctly.

The Exciter

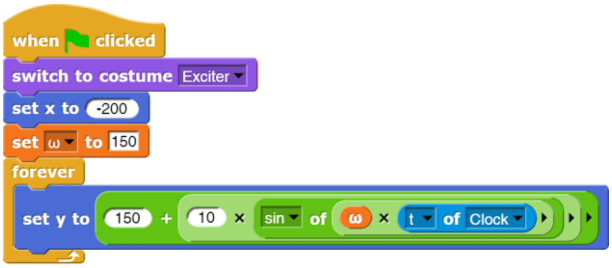
We draw a simple rectangle symbolizing a plate suspended somewhere. Since the plate should only oscillate vertically, it needs a fixed x-coordinate on the screen (here: -200) as well as a zero position in y-direction (here: 150). Around these it oscillates with a fixed amplitude (here: 10) with a variable angular frequency ω (here: 150). In the course of time t, which initially has the value zero, the y-coordinate is then calculated to be

$$y = 150 + 10 * sin(\omega t).$$

---

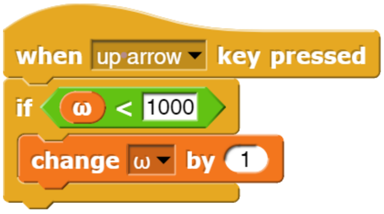
[42] Of course, we could have accessed the timer time directly instead. But I want to show access to local variables from other objects.

This information can be translated directly into a script.



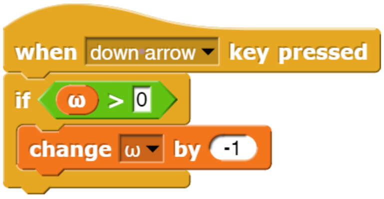
The script starts its work when the *start* message is sent. Since the scripts of the other parts have to be started at the same time, this option is suitable.

More interesting are the variables used. The time is imported from the clock. The frequency is not needed in any other script and therefore should be declared locally. You can change it using the arrow keys.



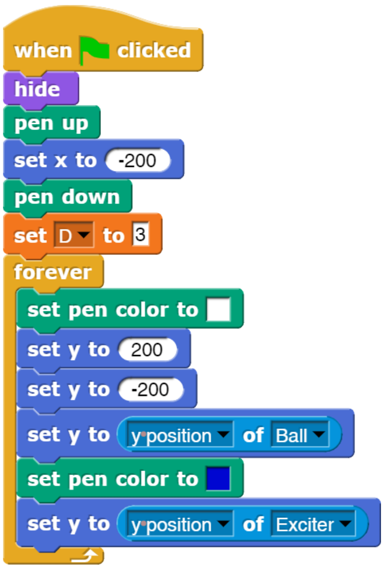
We export the sprite as described as *Exciter.xml*.

Extension: Also have the "laboratory ceiling" drawn against which the exciter oscillates. Alternatively, a shaft can rotate, which leads to a vertical periodic motion via a deflection roller.



The Thread

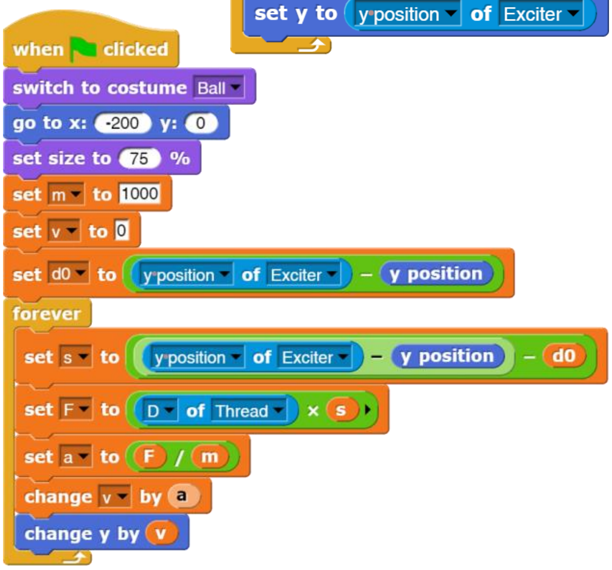
The thread replaces the spring. It has only one property, the spring constant *D*. This is set once to a fixed value, then a bright vertical line is drawn at the location of the thread, which deletes its old representation (of course, this can also be done more elegantly). After that, the current thread deflection is drawn. We export the object as *Thread.xml*.



Extension: Instead of the simple thread, draw a spiral spring with a constant number of turns that stretches and contracts again.

The Ball

Our knowledge of physics, which can be quite poor, is "built into" the ball: We know the basic equation of mechanics *F = m*a* as well as Hooke's law *F = D*s* where *s* is the deflection from the zero position. Furthermore, the acceleration *a* is known as velocity change per time unit and the velocity *v* as displacement change per time unit. Nothing else. As local variables we need the quantities to be calculated as well as the mass *m*. We convert this knowledge into a sequence of commands: we determine the momentary displacement *s*, from it *F*, from it *a*, from it *v* and from it the new position.



We export the ball as *Ball.xml*.

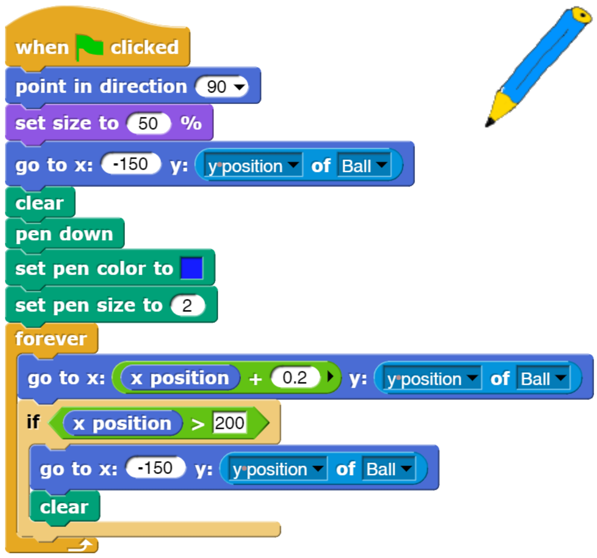
Extension: Introduce a friction constant *R* that reduces the speed by a certain (small) percentage. *R* should also be interactively changeable within a reasonable range.

The Pen

The pen has no local variables. It moves slowly from left to right and moves in y-direction to the y-position of the ball. Thereby it writes. As a little treat, we add the function that it starts writing again when it has reached the right edge.

We export the pen as *Pen.xml*.

Extension: Introduce a way for the pen to run at different speeds.



Why is this a simulation?

Our example contains some basic physics, but there is nothing about resonance, beat etc. in it. Nevertheless, they appear in the simulation. We check with the program whether the consequences „necessary from thinking" (acc. to Heinrich Hertz) of the basic knowledge agree with the observations in the experiment, whether our conceptions of the physical relations thus result in the observed behavior. We simulate a system to check our ideas. As a tool for this, instead of mathematics, we use an algorithm that tracks the system behavior over a sequence of small temporal changes. So instead of integrating "mathematically," we iterate "informatically." Except in the simple cases, a tool for integrating a differential equation system does nothing else either.

Something completely different is an animation into which the observed behavior is programmed. Here no new phenomena can arise because everything is known. *Animations* represent something, *simulations* can lead to real surprises.

6 Troubleshooting in *Snap!*

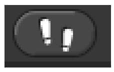
Level: *high school* Materials: *Towers of Hanoi*

Snap! visualizes the program flow without requiring any special activities from the learners. This alone makes many errors "visible" that would otherwise require tedious analysis of code to find. For example, if a body moves in the wrong direction, then it is pretty clear what to look for.

Since global and local variables can be displayed in a *monitor* on stage by setting the checkmarks in front of the variable name, their change is also directly observable. Script variables can be displayed in the same way if the blocks *show variable <name>* or *hide variable <name>* are included in the script. An important aspect of troubleshooting is the "freezing" of variable assignments when the program is stopped: if the program is interrupted or terminated, the current values of the variables are retained and can be inspected.

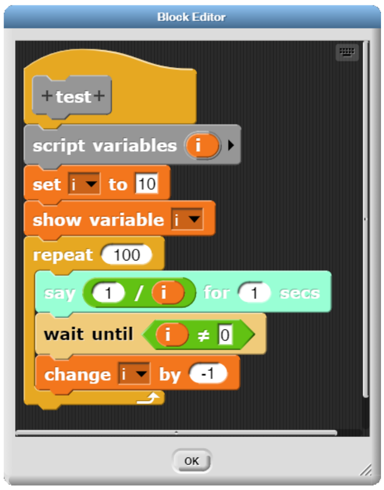
Control output during program run can be easily achieved with the blocks of the *Looks* palette: say <something> for <n> secs and its relatives also allow the output of more complex expressions so that those can be followed on the screen. The wait <n> secs and wait until <condition> blocks allow pauses in the program flow at certain points and/or when certain conditions occur.

If the sequence of the entire program is to be followed step by step, then the *Visible Stepping* must be switched on at the top next to the output window.

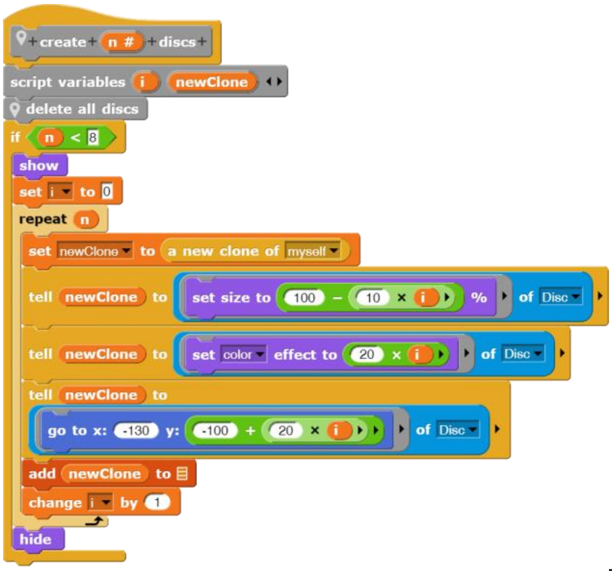


After that, the footsteps appear in light green, and a slider appears next to them that determines the stepping speed. A button for interrupting or starting stepping appears between the green flag and the red stop button. If the speed slider is on the far left, then the program can be stepped through in single steps. The currently executed block appears in light green.

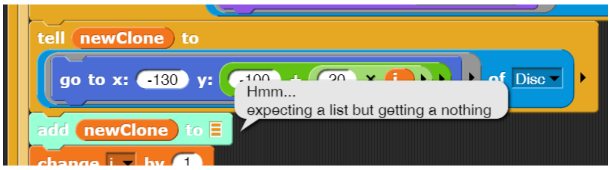
If the program flow is to be followed also within the own blocks, then these must be opened before the start of the program in the editor. The blocks can also be nested.



We want to follow the processes by means of a small example. For whatever reason - the problem of the "Towers of Hanoi" is to be processed. For this we draw a disk and assign this costume to a sprite *Disk*. Further disks are to be created by cloning. For this we wrote a method *create <n> discs* - but it doesn't work. Too bad!



To locate the error, we open the method in the editor, click the *Visible Stepping* button, set the desired speed, and then click the new block again. In the editor we can follow how the commands are called - and where it goes wrong.



Something is missing! We add the missing list variable *stackA* in the block, and this part at least runs fine.

Further blocks that can be helpful for troubleshooting can be found in the libraries. They are described by their own help pages, which are called via their context menus.

However, the - for me - most essential possibility for troubleshooting is to take blocks out of the scripts and "just leave them" next to them. If a script works afterwards, the blocks can be inserted again one after the other. Mostly the error can be narrowed down quickly in this way.

7 Lists and Related Structures

Snap! knows beside atomic data types like *numbers*, *logical values* and *characters* the structured types *string* and *list*. The strings follow later because they allow many special applications. In this section we will first discuss lists, which are practically always needed. From them all higher structures can be built easily. The use of lists will first be shown in a simple case - sorting - followed by more complex applications.

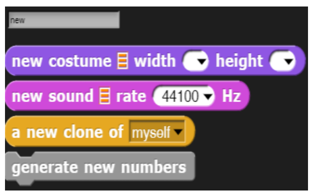
Lists are so-called *references*, i.e. addresses that "point" to a certain memory area where the actual data is located. If this is not taken into account, annoying errors can occur. For example, several list variables can point to the same data area. If, for example, you change this data by accessing it via a list variable, the changes also affect the other variables, as long as they refer to the same data. Such errors can be avoided, for example, by creating copies of the lists. Then we can work with them without interfering with the remaining operations.

7.1 Sorting with Lists - by Selection

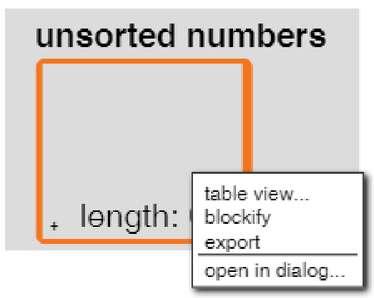
Level: *from middle school* Materials: *Selection sort*

The example is kept extremely simple: it uses only global variables and blocks without parameters, i.e. macros that serve to summarize a command sequence under a new name. Since it additionally exploits the visualization possibilities of *Snap!* it is well suited as an introductory example.

We start with an empty *Snap!* project. If we want to sort something, then the elements to be sorted have to be stored somewhere. For this purpose, there are variables, which can be thought of as "boxes" that can hold any content. For the storage of several elements there are lists, a kind of "box series". The blocks for editing variables and lists can be found in the *Variables* palette in brown.



By the way: The magnifying glass for searching in the top-right corner of the palettes shows us candidates for blocks that match the search pattern. Among them we also find blocks written by ourselves and some that are not in the palettes at all.

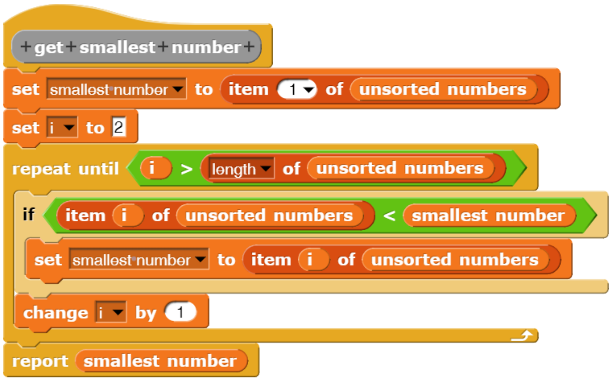


So, we create a variable called *unsorted numbers* and assign an empty list to it. (Using the arrow keys in the list block, we could also specify initial values in the free spaces that would be created. The type of the inserted variables does not matter: lists can hold anything, and in any order). If the variable is created, it appears as a *watcher* on the stage. There we can choose different display formats in the context menu or position the list as a *dialog* arbitrarily in the *Snap!* window.

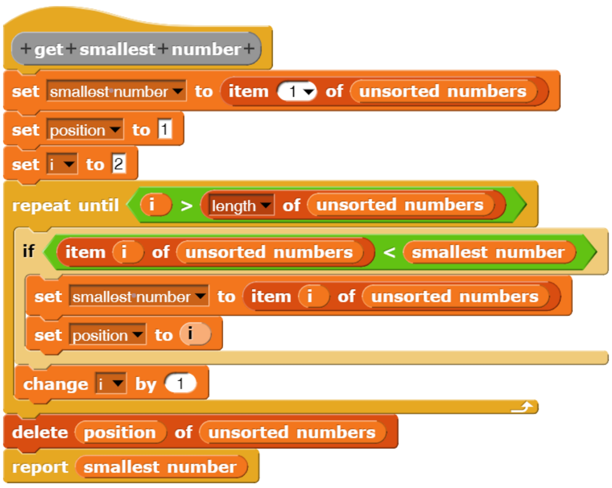
In the same way we create a second list of *sorted numbers*, which will later contain the sorted data. First of all, we need unsorted data - as usual random numbers. We create them with a small script, where the number of numbers results from the number of repetitions in the loop.

We try the script several times - we always get a new list of numbers. Great! Full of pride we form a new block called *generate new numbers*. (Right click on the script layer.) In this one we simply append our script to the "hat" with the block name. Done - we have written a new command! We can find it at the bottom of the *Variables* palette - if we have not specified anything else.

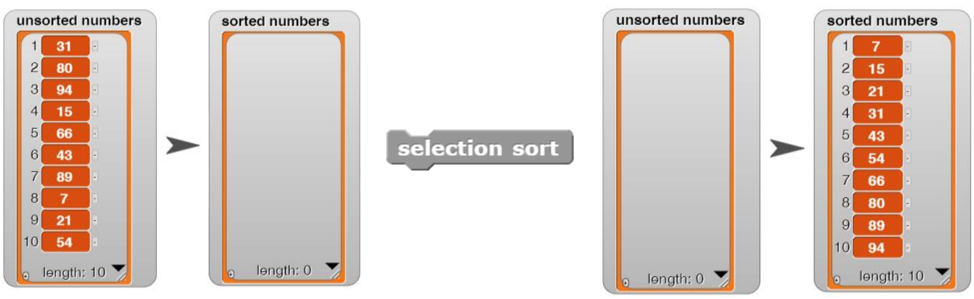
From this list of numbers, we now want to pick out the smallest number. Let's assume that the first number is already the smallest. Then we look at all the following numbers. If one is smaller than the previous smallest number, we remember it. If we are through, then we "report" the result - we write a function *get smallest number*.



That also works fine. But only once because we can't find the next smallest number this way. This is only possible if we remove the smallest number from the list each time. Because we only know which was the smallest number after the entire run, we remember its value as well as its place - and throw it out after the run through the list.



Sorting a list is now quite simple: we take the smallest number from the unsorted list one by one and put it into the sorted one. That's it. We wrap the script again in a new block, which we call *selection sort*.



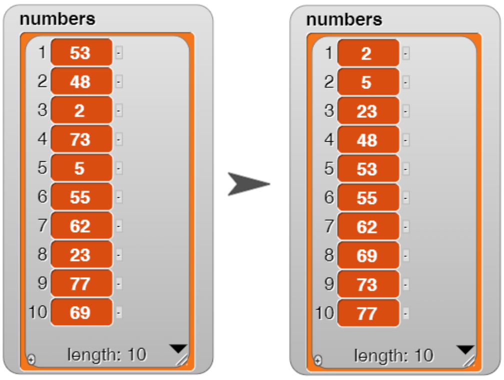
7.2 Sorting with Lists - Quicksort

Level: *high school* Materials: *Quicksort*

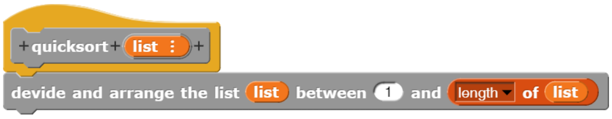
As a second, recursive, example we want to implement Quicksort[43] in the same environment as above. For this we first write a more elegant method for generating new numbers, which uses a *parameter* and local *script variable*. With this we can specify how many numbers we want to have. To be able to handle larger sets of numbers, we wrap everything in a *warp* block.

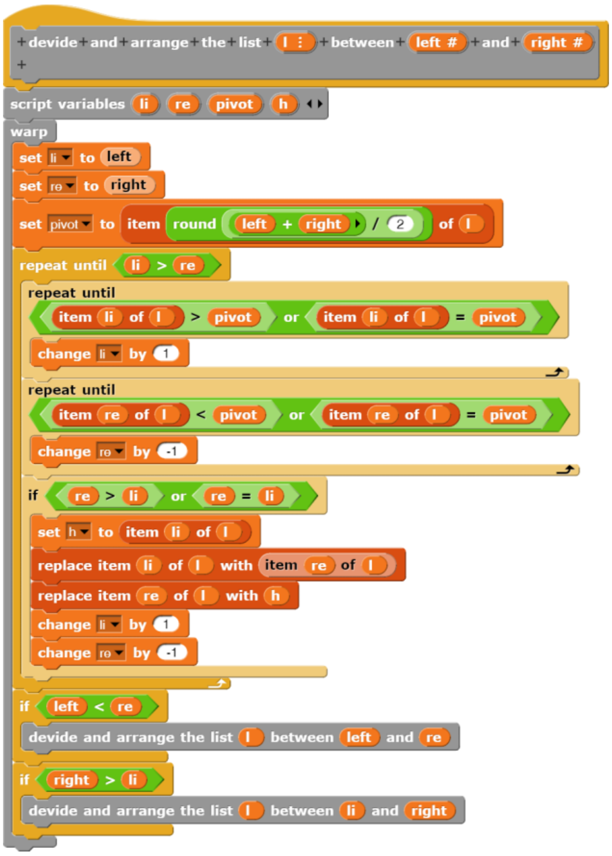
Quicksort is started by specifying the list to be sorted.

The actual work is done in the block *devide and arrange the list <l> between <left> and <right>*. As *pivot element* we select there the middle of the respective sub-list.



We can sort 10,000 random numbers with it in about 2 seconds.





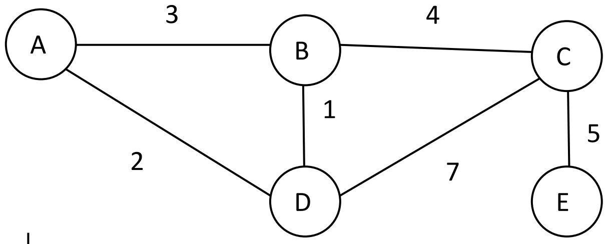
---

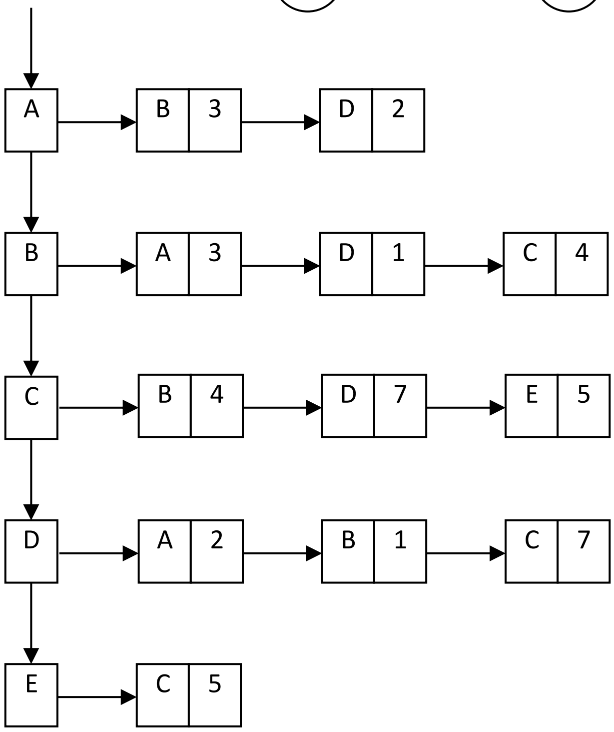
[43] The procedure can be found in various versions on the Internet, e.g. at http://de.wikipedia.org/wiki/Quick-sort. Here, an in-place implementation was chosen.

7.3 Shortest paths with the Dijkstra method

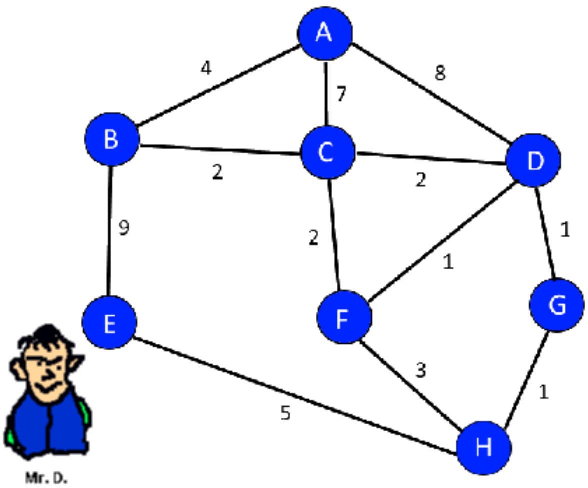
Level: *high school* Materials: *Dijkstra routing*

Let a graph be given by an *adjacency list*. In this list all nodes of the graph are listed, from each of which lists "go out", in which the neighboring nodes with the respective distances are entered: i.e. those nodes to which a direct connection exists. As examples a very simple graph and its adjacency list are given.

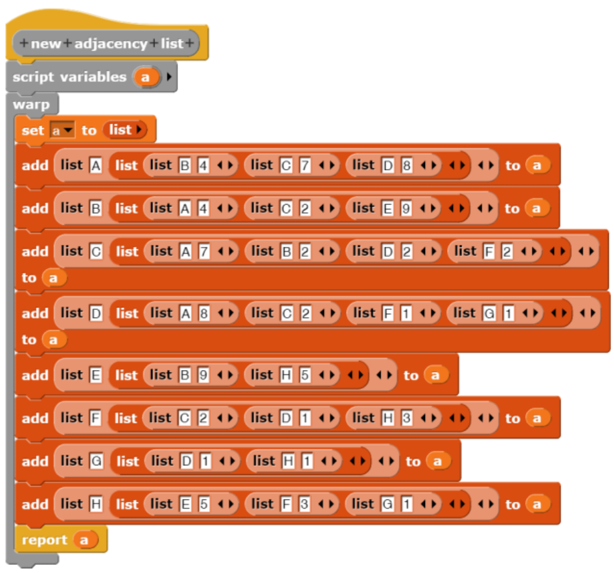




To work on the problem we need a specialist, of course: we draw *Mr. D.* He must be able to generate the adjacency list of a given graph. We simply draw the graph on the background - here done very tastefully.

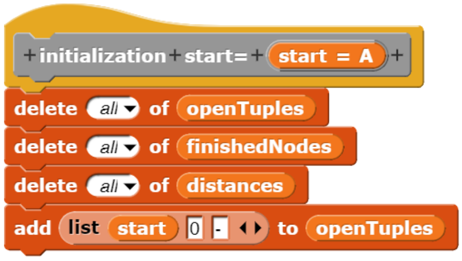


We create the list statically by inserting the appropriate elements into a local list, which we return as the result of the operation.

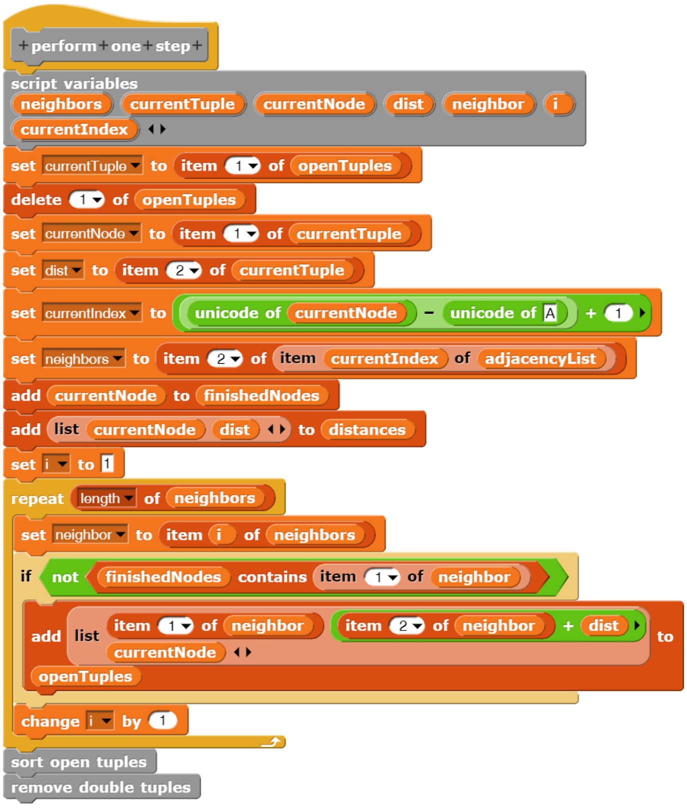


The global variable *adjacencyList* then receives these values via a simple assignment.

For further processing we need three other lists: the list of *openTuples* takes tuples containing the name of the node, its total distance from the start node and the name of the predecessor node; the list *distances* takes tuples containing the name of the node and its total distance from the start node, it is re-sorted for new entries so that the node with the shortest distance from the start is in front; the list *finishedNodes* contains the names of the nodes that have already been finished. We summarize the setup of these lists for the start in a method *initialization*, which is also passed the name of the start node. After its call the following picture results.



The path search is relatively simple in this version, because most of the "intelligence" was put into the handling of the lists. This is done in the method *perform one step*.



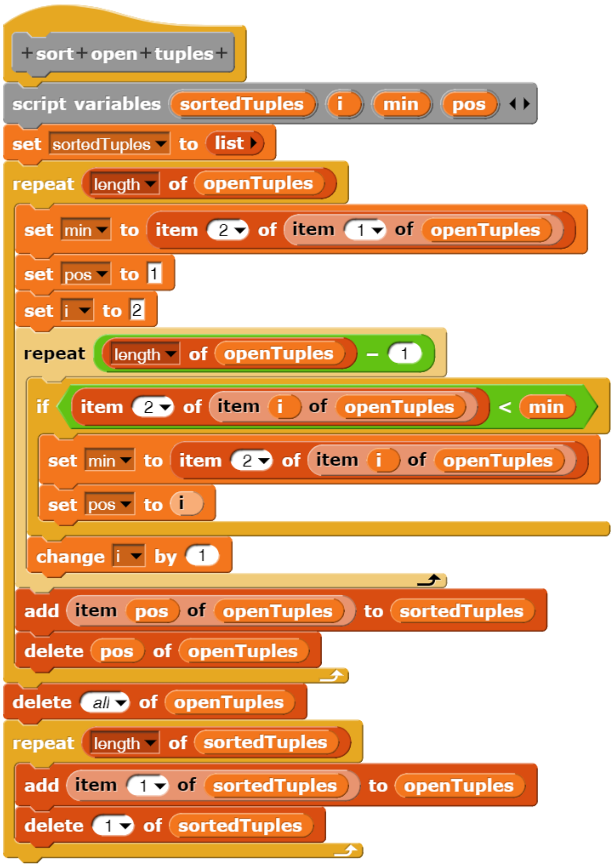
For the tuple currentTuple *with the smallest distance, the new distances are calculated for the neighboring nodes.*

Then the node is marked as edited and all unedited neighbors with new total distance and predecessor nodes are entered into openTupels.

This list is sorted by distance and tuples with larger distances are deleted.

Except for the three auxiliary methods, the routing is now complete:

We have seen above how to sort. Here it is done by selecting the smallest.



the sortedTuples *list takes in the* sorted *t*uples

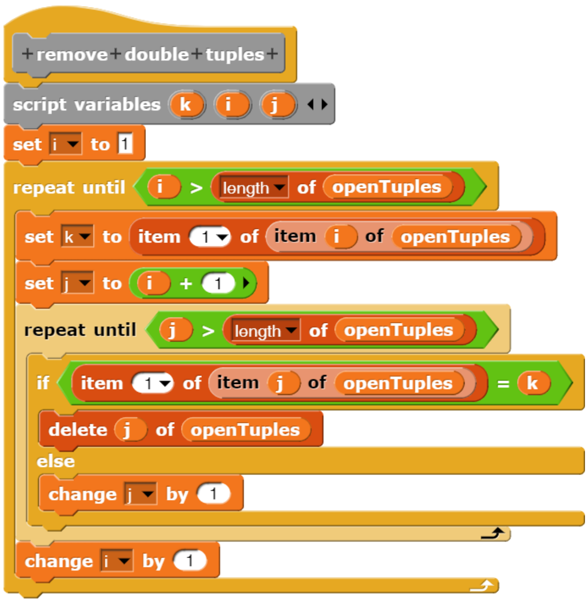
Assumption that the smallest distance is at the very front.

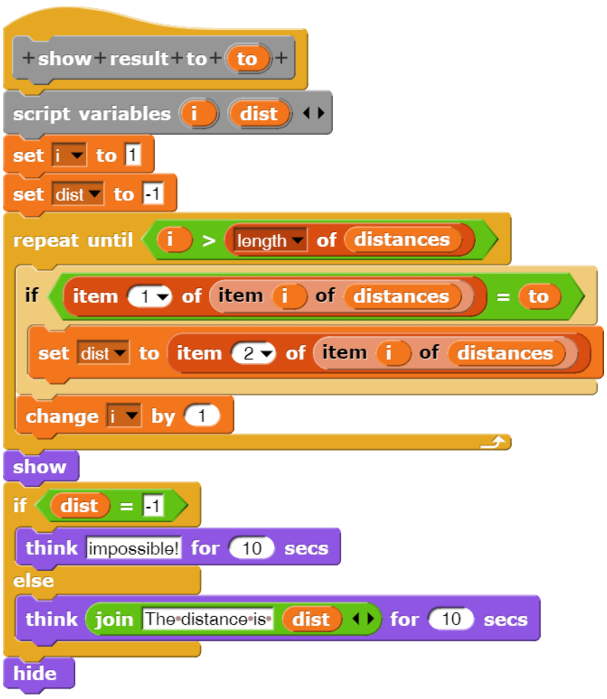
find even smaller distances if necessary

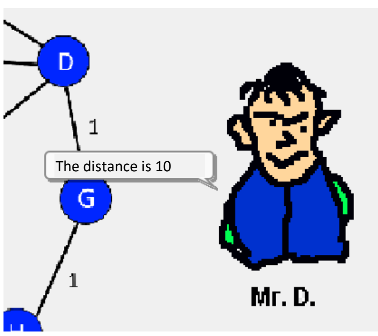
add the tuple with the smallest distance to sortedTuples *and delete it in* openTuples

lastly copy back the sorted list

Now, for each node, the tuple with the smallest distance is at the front of the list. If there are other tuples for this node, they are deleted. Then we only have to find the distance to the searched node from the distances list and let *Mr. D.* display it.







7.4 Matrices and own Loops

Level: *high school* Materials: *Matrices*

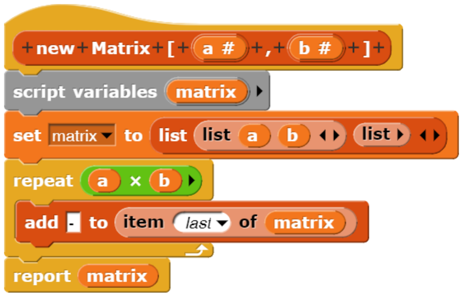
If we have lists with direct access to each element, then we don't really need special arrays, stacks, queues, etc. All higher data structures can be built from lists. Nevertheless, we will build the data structure *matrix*, because it is traditionally used e.g. for adjacency matrices. (*Attention: for the sake of brevity, we will omit all security checks!*)

We package a matrix in a list, of course. For this we declare (arbitrarily) the following list structure:

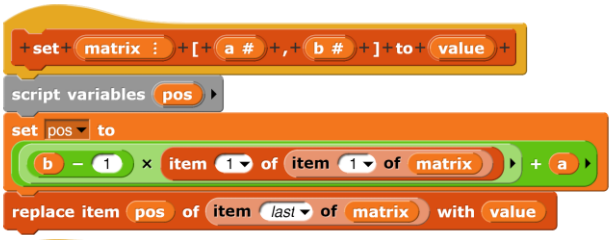
[[list with sizes of the index ranges] [list with Data………]]

The dimension of the matrix then results directly from the entries of the first partial list. A two-dimensional array with two values per row would have the following structure: *[[2,3] [1,2,3,4,5,6]]*

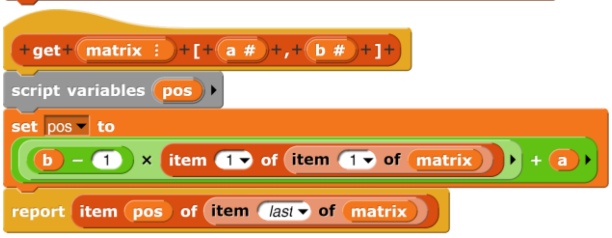
We create a two-dimensional matrix of size *a x b* by generating the two desired lists. The first one contains the two passed parameters, the second one should be marked as empty, e.g. by a minus sign for each element. We return the result. We use global methods, which we assign to the *Variables* palette. The syntax can be chosen completely freely, for example also with brackets, if you like that.



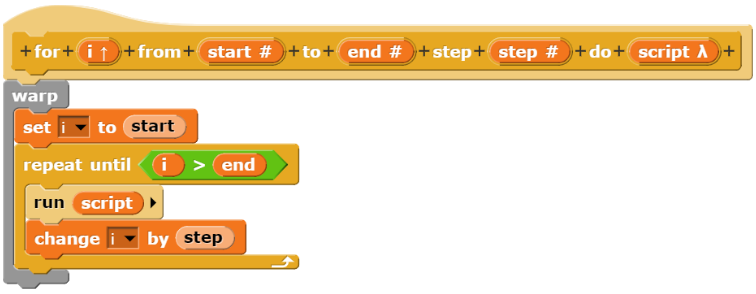
Now we write values into the matrix with *set*, nice and clear. We calculate the position of the place to be changed using the dimensions. Then we overwrite the corresponding entry.



To read matrix entries, use the method *get*.



In many programming languages the *for-loop* is the common tool to step through matrices. In *Snap!* we can find something like this in the *Control* palette, but we can also write such a control structure ourselves, e.g. to provide it with a step size. For this we create a new block *for <variable> from <start> to <end> step <step> do <script>* and take a closer look at the nature of the parameters.

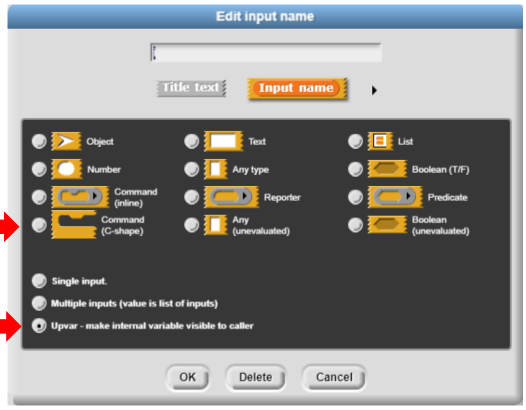


We mark the index variable *i* as *upvar*. This allows its name to be changed "externally", although its internal name remains the same - i.e. *i*.

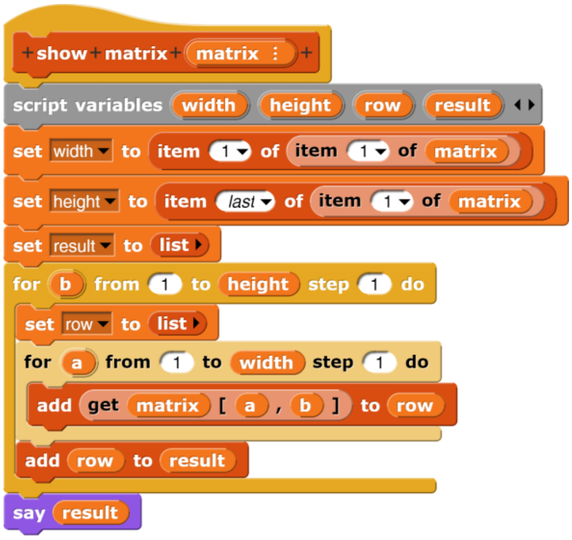
start, *end* and *step* are normal numerical parameters.

We mark the script as a *C-shaped command*. Thus, it is considered as a command sequence which is passed to the block unchanged, i.e. not evaluated.

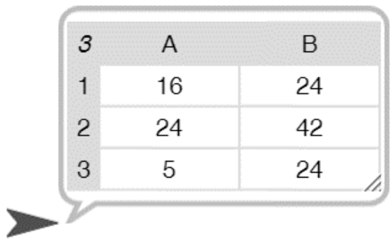
C-shaped ensures that the block has the usual appearance of *Snap!* commands, where the sequence of commands to be executed is inserted into the "mouth" of the *C*.



Using this type of loop, we can quickly fill a matrix with random numbers.

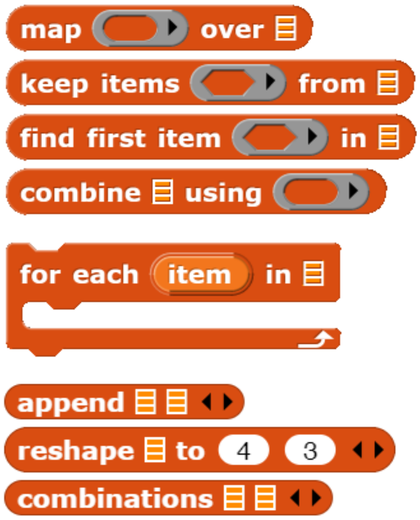


Finally, we want to display the matrix "properly" on the screen, i.e. in the usual two-dimensional table form. To do this, we create a list that is filled with sub-lists, the rows of the matrix, that contain the table data. This list is then displayed and can then be moved anywhere with *open in dialog...* from its context menu.



3	A	B
1	16	24
2	24	42
3	5	24

7.5 Higher Level List Operations



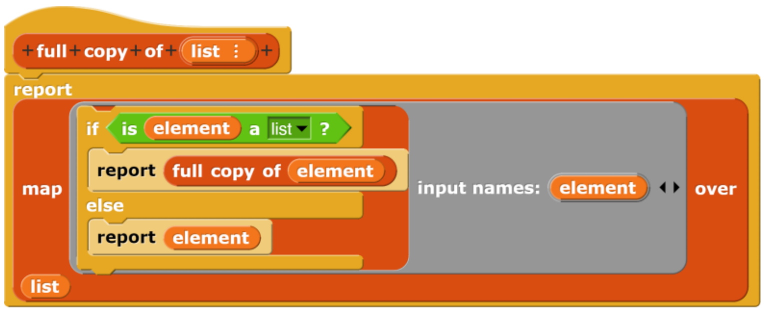
In the *Variables* palette we find some fast blocks that allow more complex operations even on large lists. The most important of them is the *map ... over...* block. It applies a script located in the gray ring to all elements of a list in order and returns the results as a new list. In the default case, the current list element is inserted into a placeholder left empty in the script. If you want to make it more readable, then you assign a name for the element and use that in the script. If you need the index of the list element, you get that in the field after the element name. After that you will find a reference to the complete list.

As an example, let's create a *copy of a list*. The first list should simply consist of the first 100,000 natural numbers.

From this list we can now create copies in different ways. In the simplest case, by using the *map...over...* function directly. But we can also name the old list element and return it under the name, and we can also do this explicitly using the *report* block.

Of course, we can also use one of these versions within a new block for copying simple lists.

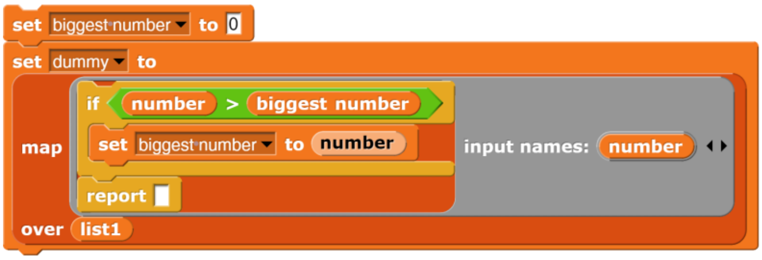
And if you remember that lists can also contain further sub-lists, then of course these must also be copied separately in the case of a copy - nicely recursive.



In some cases, you may want to take advantage of the speed of the *map...over...* block to search a list. As an example, we want to find the largest element of a list. Since the *map...over...* block must return a new list element for each element, we cannot simply get the largest element we are looking for as a result. Instead, we run through the list and determine the largest element separately as a *side effect*. We ignore the results of the actual function call, e.g. by assigning them to a *dummy* variable and not using them further.

First, we need unsorted numbers, which we quickly create using the *map...over...* block.

After that, we pick the largest number out of these 100,000 values by going through the list and finding the largest number in each case. We simply return "nothing" as elements of the result list.

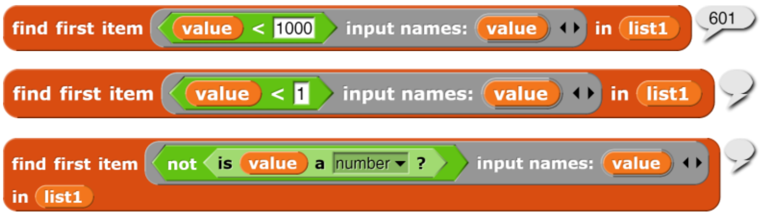


A special control structure for traversing a list is the block *for each...in....* In the libraries you can also find a version that allows access to the index. Also, with this block we can determine the largest number of a list. But this is only fast for long lists if we use the *warp* block - but then very fast.

Another very useful block is the *keep items...from...* block. It contains a *predicate* in the gray ring, i.e. a function that returns either *true* or *false*, which is applied to all list items. The result is a list containing only the list items evaluated as *true*.

We take again the just generated list with 100,000 random numbers and want to find out from this only those, which are even, i.e. divisible by 2. Here, too, we get access to the list element, its index and the total list via the small black arrows on the right of the gray ring.

The block *find first item ...in...* works very similar. It finds the first item that matches the given predicate. If it does not find an item, then it returns "nothing". The block is useful, for example, to make sure that there are no "illegal" elements in a list. For example, if we want to perform arithmetic operations on the list elements, then there should be only numbers in the list.



If you want to perform an arithmetic, logical or list operation on all list elements, you use the *combine ...using ...* block. It successively applies the specified binary operator to the entire list. As an example, we want to calculate the sum of the list elements of *list1*.

If the list contains strings, we can form a long word from it.

And of course, we can combine the higher list operations, e.g. by computing the sum of all multiples of 231 of a list.

Using the *append* block you can append lists to each other.

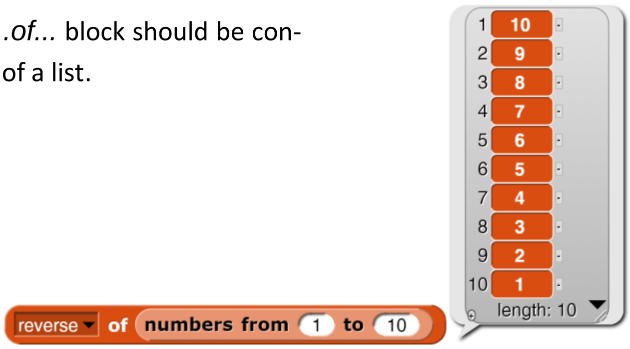
And the *reshape* block "reformats" a list to other dimensions.

3	A	B	C	D	E
1	1	2	3	4	5
2	6	7	8	9	10
3	11	12	13	14	15

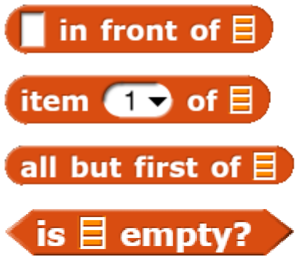
The *combinations* block is the Cartesian product of several sets. It combines (in the case of two sets) all elements of one set with all elements of the other set.

15	A	B
1	1	11
2	1	12
3	1	13
4	1	14
5	1	15
6	2	11
7	2	12
8	2	13
9	2	14
10	2	15
11	3	11
12	3	12
13	3	13
14	3	14
15	3	15

And as a last note: the possibilities of the *...of...* block should be considered if needed, e.g. to reverse the order of a list.



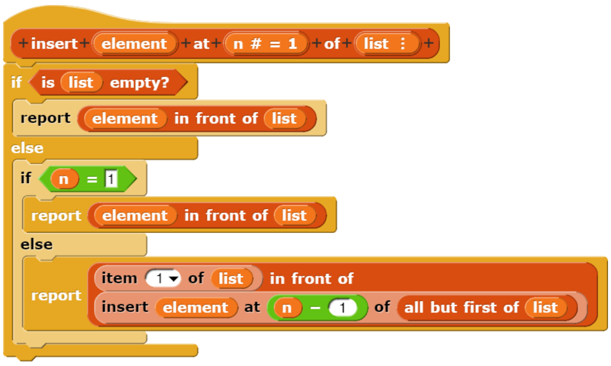
7.6 Recursive List Operations



After the very powerful blocks, let's briefly look at the more elementary blocks for recursive programming. All advanced operations can be built from these. Not too efficient, but elegant. The first block allows to insert an element at the front of a list, the second returns the first element[44]. The third block returns the remaining list after the first element, and the last one checks if a list is empty.

We can think of lists as pairs consisting of a first element and the rest, and these pair elements can also be empty. Traditionally, they are called *car* (pronounced "*carr*") and *cdr* ("*cudder*"). As an example for the application, we want to determine the length of a list.

In a very similar way, we can insert a new list element at a specified place in a list. If the list is empty, then we simply return a list containing only the new element. Otherwise, we check whether the element should be added at the front, and if so, we do so. If this is not the case either, we supply a list that contains the first element at the front and a list in which the new element is placed before the previous one, but in the rest of the list.

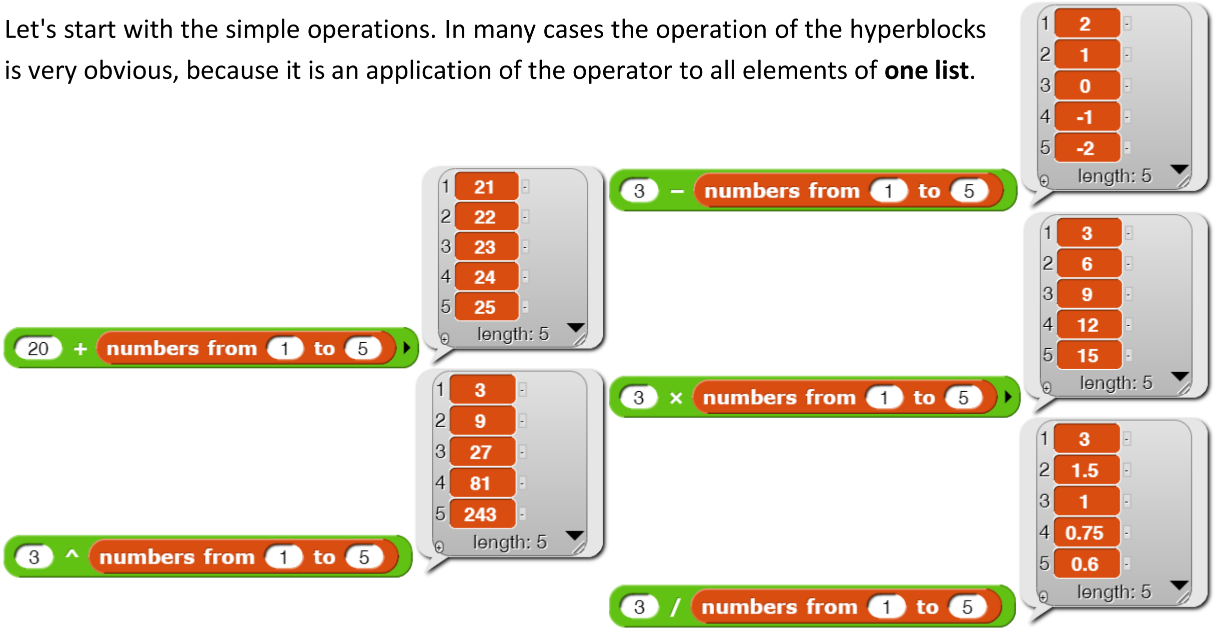


[44] Or any other, but let's forget about that. 😉

7.7 Hyperblocks

Some of the e.g. arithmetic operators have been extended in *Snap!* so that they can also be applied to lists. The result are extraordinarily fast list operations that allow, for example, the manipulation of animated images in real time. Hyperblocks can thus be applied to large amounts of data and are therefore suited to handling media. Some of the operations are immediately obvious, but some take quite a bit of getting used to. One should test the procedures in each case with small test lists, before one "lets them loose" on large data sets. Although many of the operations are based on mathematical procedures, they often do not provide mathematically correct results, e.g. because they are not mathematically permissible due to different dimensions. However, if you want to implement e.g. a matrix multiplication (see below), then it makes sense to check the dimensions in advance and then let the hyperblocks work. A detailed description of the hyperblocks can be found in the *Snap! manual*.

Let's start with the simple operations. In many cases the operation of the hyperblocks is very obvious, because it is an application of the operator to all elements of **one list**.

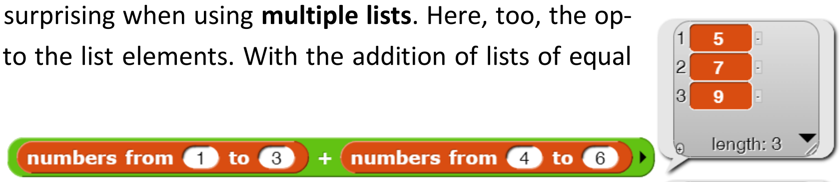


Applying an operation to lists also works if the list consists of partial lists, as is the case with the pixel list of an image. We select any image …

… and then switch to one modified by "coarsening" the value range.

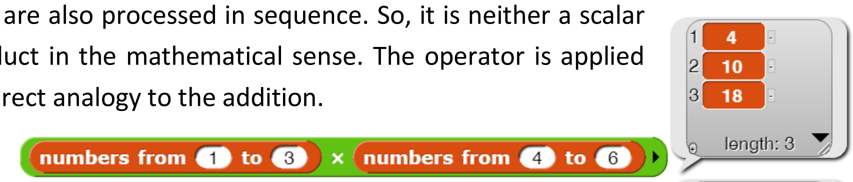
The operation is fast enough to transform, for example, video images in real time.

The results are somewhat more surprising when using **multiple lists**. Here, too, the operators are applied successively to the list elements. With the addition of lists of equal length this is still clear ...



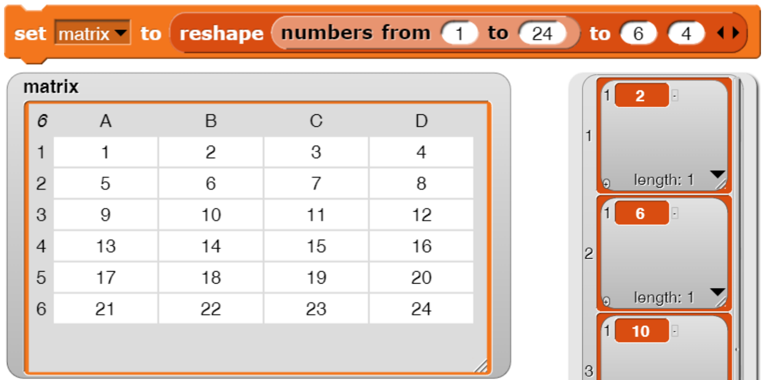
... but with lists of different length you have to know that the result will be truncated.

In multiplication, the elements are also processed in sequence. So, it is neither a scalar nor a vector nor a matrix product in the mathematical sense. The operator is applied regarding the "reductions" in direct analogy to the addition.



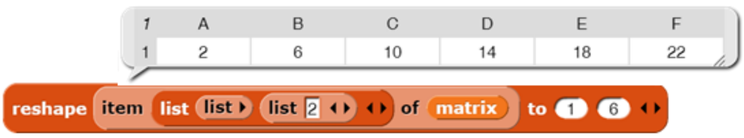
If you process more complex list structures, you should read the manual beforehand to understand how they are handled. As an example, for the use of empty lists as a "direction flag", the following example shows how columns of a matrix can be filled out.

First, the list elements are created and put into matrix form. Note that the first parameter determines the number of rows, the second the number of columns. So, you get a 4 X 6 matrix.

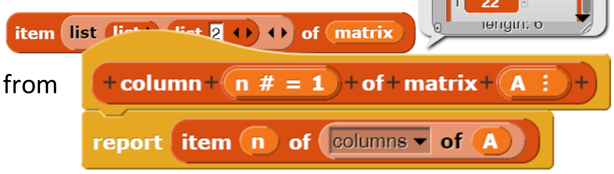


From this you can extract the nth column by specifying the column number ...

... and reformat if necessary.

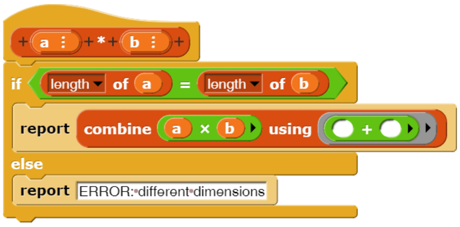


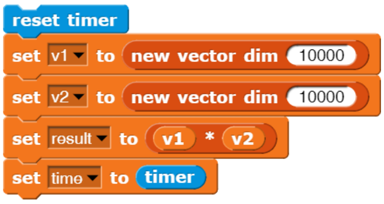
Much more understandable for the column determination is the calculation of the transposed matrix, from which then the nth row is taken:



With this knowledge you can already do something. First of all, we build a block for the *scalar product* of two vectors. Of course, we need vectors of different lengths for testing, here with random numbers as contents. For this we use the fast *map...over...* block.

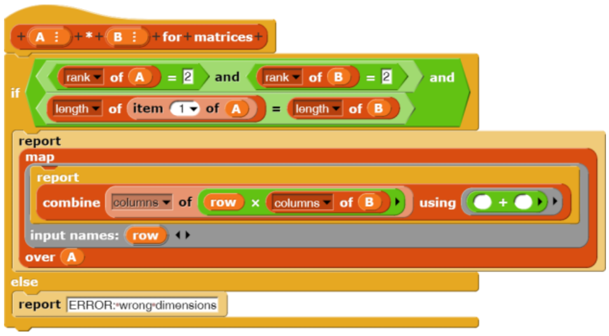
To calculate the scalar product, we use the multiplication block as a hyperblock and add the results using the *combine...using...* reporter. And because we want to stay mathematically correct, we check the dimensions of the vectors beforehand. The result is a very fast block that gives the measured multiplication time of 0 seconds for two 10,000-digit vectors, for example.

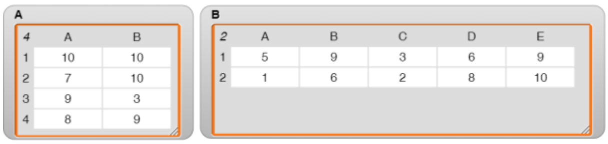




If this works so well for vectors, then of course we'll try our hand at *matrix multiplication* in the usual way. First of all, we have to create matrices. We do that similar to the vectors.

The block for the matrix multiplication checks first of course also whether the dimensions are correct. Then it uses hyperblocks and higher functions. Because *columns of <list>* calculates the transposed matrix, we can multiply the rows of the first matrix *A* respectively with all rows of the transposed matrix *B*, i.e. with the columns of *B*, scalarly.



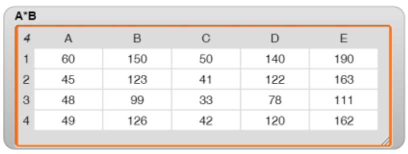


A

	A	B
1	10	10
2	7	10
3	9	3
4	8	9

B

	A	B	C	D	E
1	5	9	3	6	9
2	1	6	2	8	10



A*B

	A	B	C	D	E
1	60	150	50	140	190
2	45	123	41	122	163
3	48	99	33	78	111
4	49	126	42	120	162

Again, we can measure the time for larger amounts of data. If we multiply two 100X100 matrices, then it takes 0.1 seconds.

7.8 Fast Image Manipulation with Precompiled Blocks

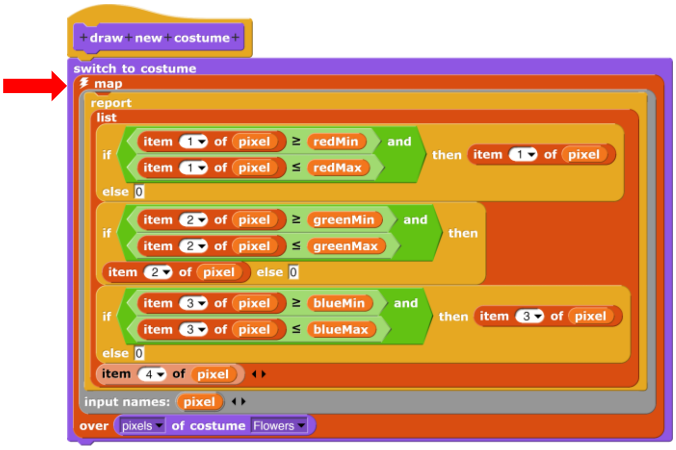
Level: *high school* Materials: *Expose flowers*

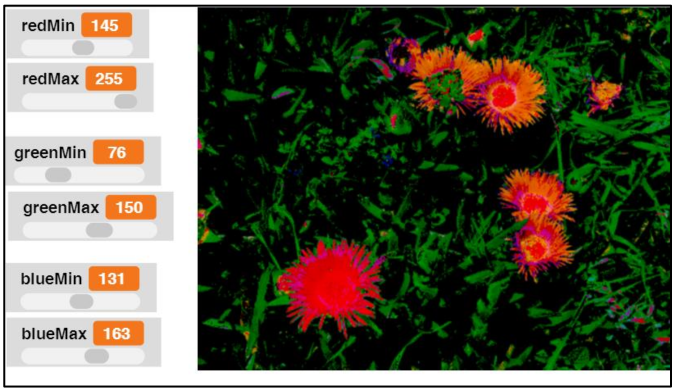
As a last application we want to show how to change and display an image in real time using the *pre-compiled map...over...* block. As an example, we choose a color image in which we want to display only adjustable color ranges. This can be used, for example, to identify faces in an image or, as here, to extract flowers.

In order to be able to react directly to changes, we use two variables each for the limits of the color ranges: the current value and the last value. If the current value changes, then the image is recalculated, and the last value of the color value is adjusted. For the change of the variable *values* we use the slider representation of the variables.

At the beginning the "old values" simply get the current values. After that, the scripts reacted to changes as described, e.g. for the red area.

The image is recalculated by checking for each of the three color channels whether the color value lies within the selected range. If this is the case, the color value is accepted, otherwise it is set to zero. Finally, the current transparency value is appended to the pixel. Note the small lightning bolt at the top of the map block. It means that the script is precompiled inside the block and therefore can be applied very quickly to all list elements. You can get this option by selecting it in the context menu of the block.





7.9 Tasks

1. Find out about the different **sorting methods** on the web. Implement some of them like shakersort, gnomsort, insertion sort, …

2. Complete the specified methods in such a way that **erroneous entries** are intercepted.

3. Implement **matrices** differently by structuring the lists used differently.

4. a: Learn about the data structure **dictionary**.
   b: Implement the structure with appropriate **access operations**.

5. a: Implement the data structure **stack**.
   b: Implement the data structure **queue**.

6. Implement a simple **binary tree** with the operations
   a: new tree
   b: add <element> to <tree>
   c: count elements of <tree>
   d: exists <element> in <tree>?
   e: remove <element> from <tree>
   f: determine the maximum depth of <tree>
   g: balance <tree>

7. Implement other **control structures**:
   a: do <script> until <predicate>
   b: while < predicate > do <script>
   c: case <variable> of < [[value1,script1], [value2,script2], [value3,script3], …] >

8. Implement **recursively** using only the elementary recursive operations
   a: the data structures stack and queue.
   b: an operation that deletes the nth element of a list.
   c: an operation that replaces the nth element of a list with a new element.
   d: an operation that returns the nth element of a list as result.
   e: an operation that adds a new element to the end of a list.

9. Implement common matrix operations using hyperblocks.

10. Implement a method to increase the contrast of an image using hyperblocks.

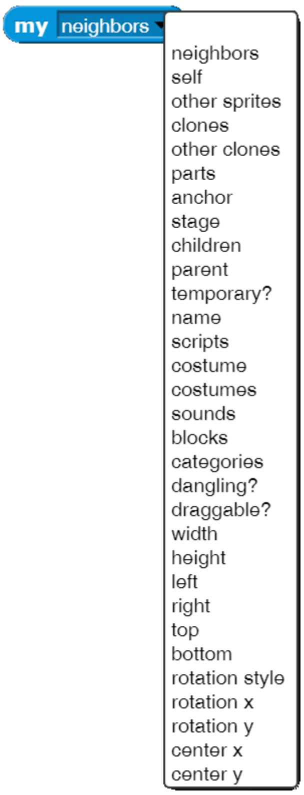
11. a: Implement a method to convert a color image to a black and white image using a precompiled block.
    b: Implement a method to obtain three color separations in the primary colors using precompiled blocks.

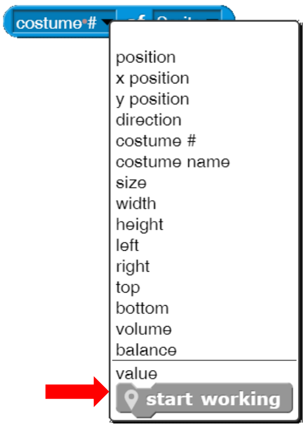
8 Object-Oriented Programming

OOP methods have also been used so far - because there is hardly any other way. At this point we want to present the OOP features of *Snap!* in more detail. We explicitly refer to the *Reference-Manual* of *Snap!*, where the methods are explained compactly. You can find it by clicking on the *Snap!* symbol in the upper left corner.

The blocks that are significant for OOP can be found in the *Control* and *Sensing* palettes, but the context menu in the Sprite area should also be noted. The lower blocks of the *Control* palette are for "dynamic" management of sprites, the menu for "static". This difference is significant because it is assumed that only the static clones should be permanent, the others are e.g. deleted when saving and not even displayed in the sprite area.

Snap! of course works all the time with objects, which are called *sprites* here. They have their own attributes (*position, direction, costume, ...*) which can be accessed using different blocks. The *my <attribute>* - block provides the whole palette, the *<attribute> of <sprite>* - block knows the most important ones and additionally shows the local variables and methods of a sprite.

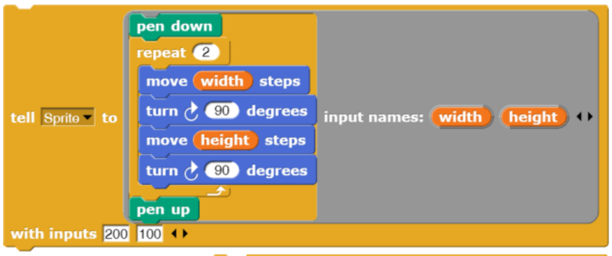


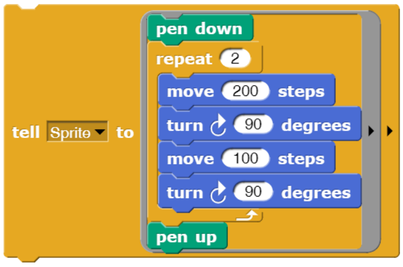


You can get the value of a local variable (here: the position) of another sprite e.g. with

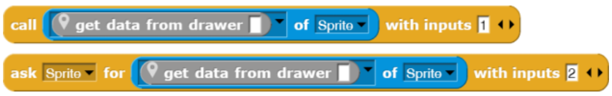
To select a local method, we put the prototype of the considered object into the *<attribute> of <sprite>* block on the right and then select the desired method. The block returns the code of the method, which can be seen by the gray ring around the method name. We execute this code in the context of a sprite that can do something with the code: usually the prototype, a clone, or a copy of it. This can be done using different blocks. If you call a local method of a sprite "from outside", then in my opinion the *run* block is the most intuitive to understand, if you ask a sprite to call global methods, then it is better done by the *tell* block. The *launch* block starts a script as an independent process.

Since a script is inserted into a gray ring, it can of course consist of several commands, and parameters can also be used, which are inserted into the empty slots of the blocks, and which can be named if required. This can be useful, for example, if you are using multiple parameters and want to make sure they are inserted in the right places. Since the parameters are determined outside the called sprite, they must also (usually) be listed outside the script under *with inputs*. If the parameters are not named, then they are inserted sequentially into empty slots in the script blocks. In many cases you can insert parameters directly in global blocks.

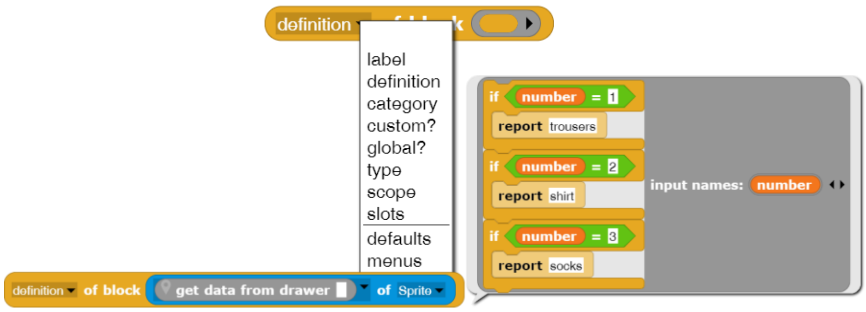




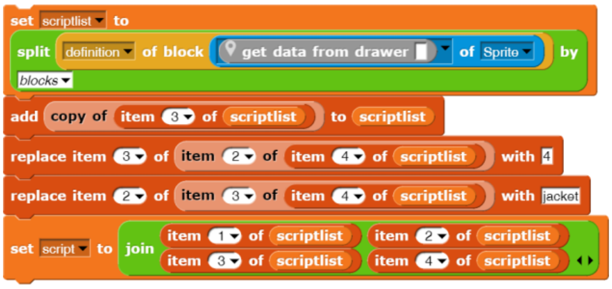
Local *reporter* blocks are handled quite similarly, but by the corresponding reporter blocks of the *Control* palette. Again, the *call* block is more suitable for local reporters, while the *ask* block is more for calling global methods in the context of another object.



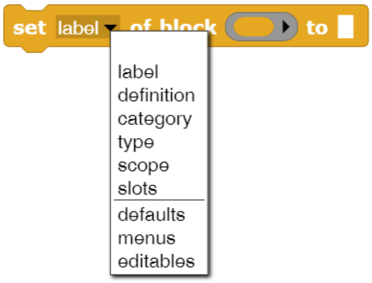
A newer feature of *Snap!* is *metaprogramming*, the ability to manipulate a script directly by other scripts. For example, if we are interested in the contents of the *get data from drawer <n>* block, then *<...> of block <a block>* block will get the corresponding script. We can convert that into a list of commands with the *split* block.

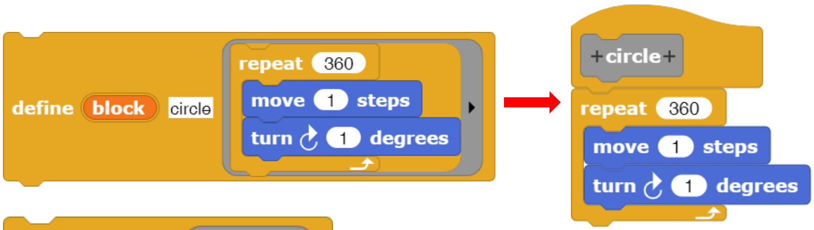


To this list we can simply add a copy of a command and change the parameters. The modified list is merged with *join* to form a new script that can be executed with the *call* block.



Setting code parameters can also be done during the program, and the block itself can be completely created or deleted.





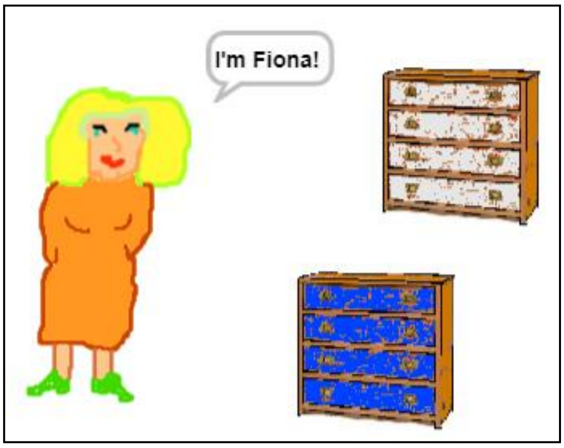
Using the *clone* command from the context menu of a sprite (see above) we can create additional static clones. These are randomly distributed in the output window. Dynamic cloning also creates new sprites, but they are all in the same place. If you save the project and reload it, the statically created clones will be created again, but the dynamically created ones will not.[45]

An essential aspect of OOP is *inheritance*. In *Snap!* this is based on Lieberman's delegation model[46], which works with prototypes (i.e. concrete objects, not abstract classes) and clones and modifies them if necessary. The model was described earlier. We will illustrate all procedures first with simple examples, then with more complex ones.

[45] This is a real benefit: with many clones, it is otherwise often difficult to get rid of them without destroying the project.

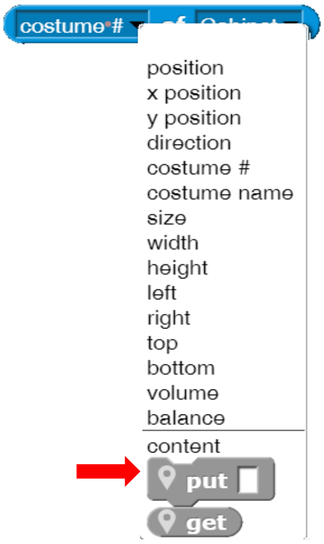
[46] Lieberman, Henry: Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, ACM SIGPLAN Notices, Volume 21 Issue 11, Nov. 1986

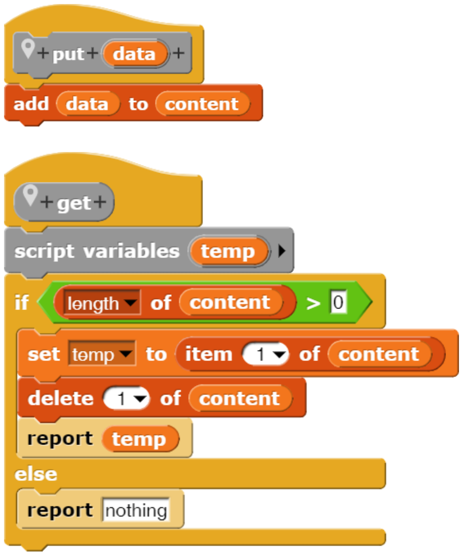
8.1 Fiona and the Filing Cabinets



Level: *high school*
Materials: *Fiona and the filing cabinets*

We draw the costume of an elegant commode and create a sprite named *Cabinet* for it. The commode contains a local list variable *content* as data storage, which we represent by this very commode. We equip it with local access methods to the data by implementing the methods *put <data>* and *get*. This results in a simple *queue*. We can use it to write arbitrary contents to and from the list. Both methods and the variable are indicated by the *<attribute> of <sprite>* block.



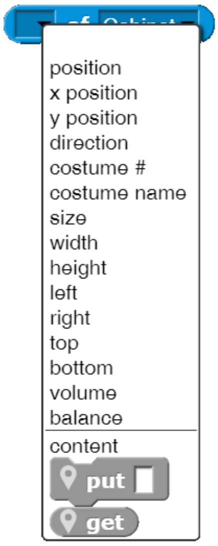


We want to use two of these data stores. For this purpose, we can either create *copies* or *clones* of the commode. With copies, later changes to the prototype are not applied, but with clones they are. An exception are list variables. Here, a *reference* to the list is copied in both cases, so that changes to the list, e.g. insert operations, affect clones and copies. To get independent lists, we need to break this link after cloning, e.g. by re-setting (*set <content> to <list>*) or copying (*set <content> to map <> over <list>*) the list. We opt for copies here and create two of them, the sprites *Papers* and *Souvenirs* with slightly different costumes. For these we need an access from outside.

We get help from the IT officer *Fiona*. Fiona can see the existing methods on other sprites, but how can she access the data stores? There are several ways to do this in *Snap!* for *commands* and *reporters* respectively.

Find method of another sprite: 

- Select sprite (prototype if necessary) in the right input field:
- Select method in left input field:



The call returns the code of the method:

execute local method of another sprite:

Parameters are passed in order in the fields after "with inputs". They are only inserted into the blanks of the method header on the side of the called object when it is clear which method will be executed at all.

Commands

with *tell*:	Fiona transmits to the addressed object (here: *Papers*) the method header to be executed with the associated parameter values (here: *personnel file*). The object being addressed follows *tell*. 
with *run*:	Fiona asks the *Papers* object to execute the submitted method with the associated parameter values (here: *personnel file*). The called object is named in the input window of the *of* block.  Important: The method is selected first by specifying a suitable prototype or clone as object. After that the actually meant object is inserted, which can also be stored e.g. in a variable!
with *launch*:	like *run*, except that the script is executed as a separate process, i.e. without waiting.

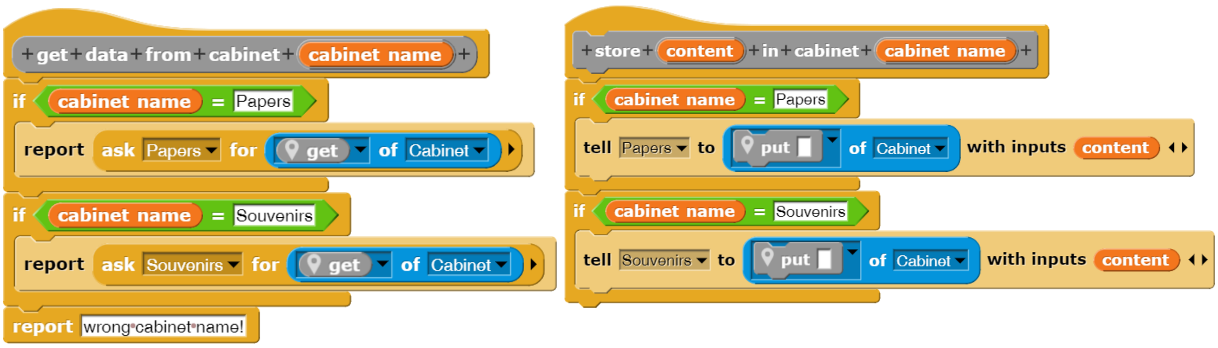
Reporter	
with *ask*:	Since this is a call to a reporter method (a function), a value is returned. Any parameters are passed as described above. The called object follows *ask*. 
with *call*:	Comparable to *run*. Here, too, the called object is named as the second input. 

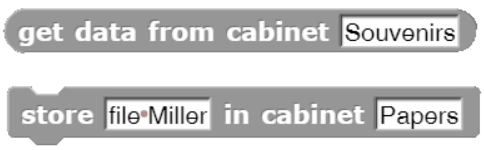
If attributes of another sprite are to be changed from the outside, then this can be achieved as usual via a *set* method. But it can also be done directly: we execute the *set <variable> to <value>* block in the right context. To do this, it must be wrapped in a gray ring to prevent it from being evaluated as a parameter even before run is executed. That would be in the wrong context. The ring is used to pass a block as code (see above), and not its result after execution.

This block is to be understood as: *"Execute the code that assigns a value to a variable in the context of the Papers object with the parameter values* content *and list (1,2,3)"*.

And of course, we can also call the standard blocks.

Fiona as a well-trained IT officer can of course issue such commands, but a normal user probably cannot. Fiona therefore provides new global blocks, which additionally receive the file cabinet to be used as a parameter. This simplifies the usage in the whole system very much. Fiona is pleased about the positive feedback.





Tasks

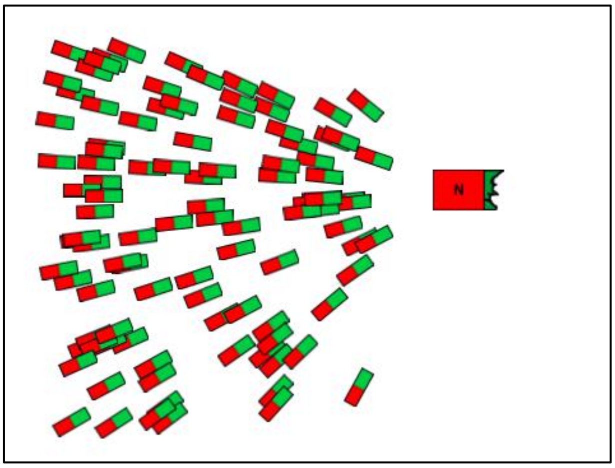
1. Implement **access control** at the filing cabinets themselves or with the IT representative
   a: by a password request.
   b: with lists of users and assigned passwords.

2. Process the data by
   a: introducing **plausibility checks**.
   b: introducing **encryption**.
   c: implementing **organizational forms** in lists, rows, stacks, queues, trees, etc.

3. Save the data in a suitable way in **text files**.

4. Organize a "**data center**" that keeps, secures and organizes the data of a company (a school, a family, ...). Define access rights and methods and implement the procedures.

8.2 Magnets

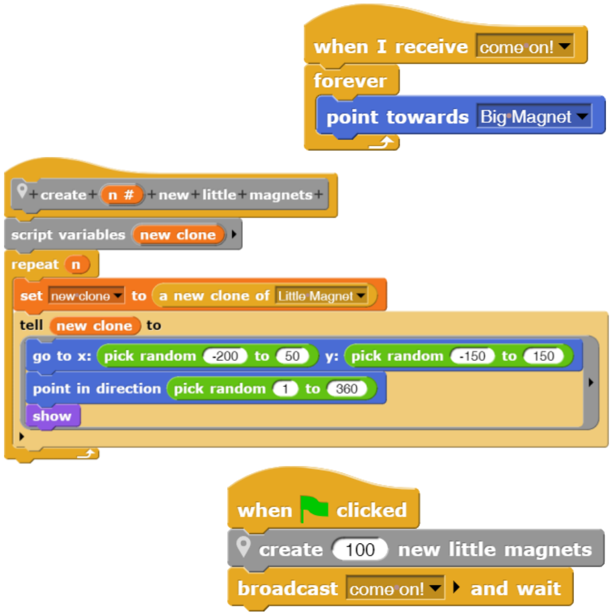


Level: *from middle school* Materials: *Magnets*

As a very simple example of how to deal with objects, we choose a magnetic field whose orientation near a "north pole" is indicated by "elementary magnets". The little ones are simply supposed to point to the north pole.

So, we draw the big magnet without any functionality (you can only push it around) and a single small one. We equip this one with the required properties and clone it as often as necessary.

Pointing to the big one is simple. If an elementary magnet receives the message "*come on!*", it continuously points to the north pole.



Cloning is a bit more complicated, because we want to distribute the clones naturally in the image area, like this: We write the method as a block of the large magnet. In it we create a clone of the small magnet and assign it to a local variable. We then send the clone to the position specified by the parameter values, rotate it in any direction and let it appear. Ready.

Dealing with many dynamically created clones is extremely easy: if we click the red stop button at the top-right of the window, all of them are gone again. And since dynamically created clones are not displayed in the sprite area, their scripts are really fast. If you move the big magnet, then all elementary magnets realign themselves - immediately.

Task:

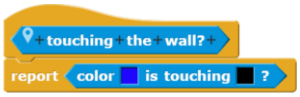
Add a "south pole" to the "north pole" and determine the direction of the force on the elementary magnets at their locations. Align elementary magnets in this field.

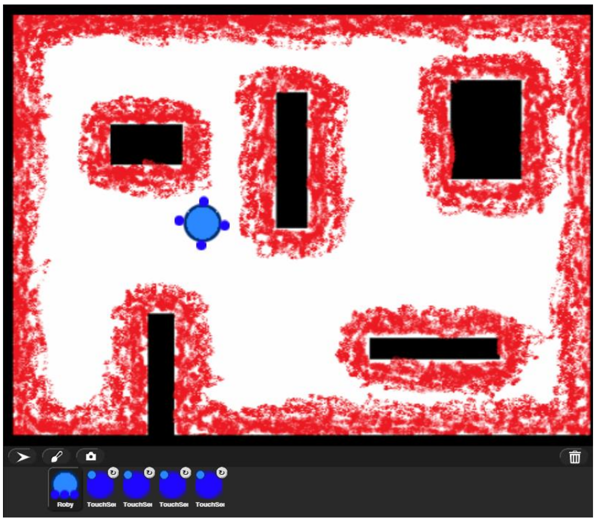
8.3 A Learning Robot[47]

Level: *high school* Materials: *Learning robot*

As another example of inheritance by delegation, let us consider a robot that has four touch sensors. If one of them comes into contact with an obstacle, then the robot changes its direction, but also has a new bump.

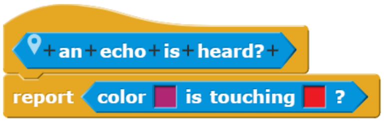
Using a drawing program, we draw a picture of a world bordered by black walls and in which there are some black obstacles. For reasons we will learn in a moment, we spray a diffuse red mist around the objects and along the walls with the spray can. Into this world we place *Roby* - as a small circular sprite. Further we draw an even smaller blue sprite, which we endow with a predicate *touching the wall?*, i.e. a touch sensor. We clone this sprite three times and then attach the four sensors to the robot.[48] We name them *TouchSensorN*, *TouchSensorE*, ... etc. according to the compass directions. The result is an *aggregation*. We equip the robot with two local variables *vx* and *vy*, which describe its velocity components in these directions. If now a touch sensor reports a wall, then the corresponding velocity component is changed. We get the following configuration, in which *Roby* moves safely between the obstacles - as said, with many bumps:





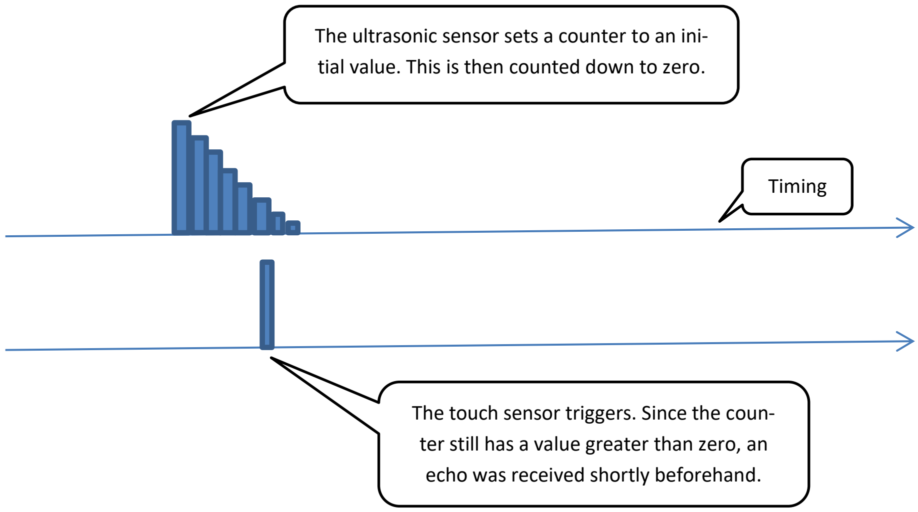
[47] The example has the walking robot of Prof. Florentin Wörgötter, Bernstein Center for Computational Neuroscience Göttingen, as a template, described e.g. in http://www.chip.de/news/Schnellster-Roboter-lernt-bergauf-zu-gehen_27892038.html.

[48] The next chapter describes how to create aggregations of sprites, i.e. how to pin sprites to others.

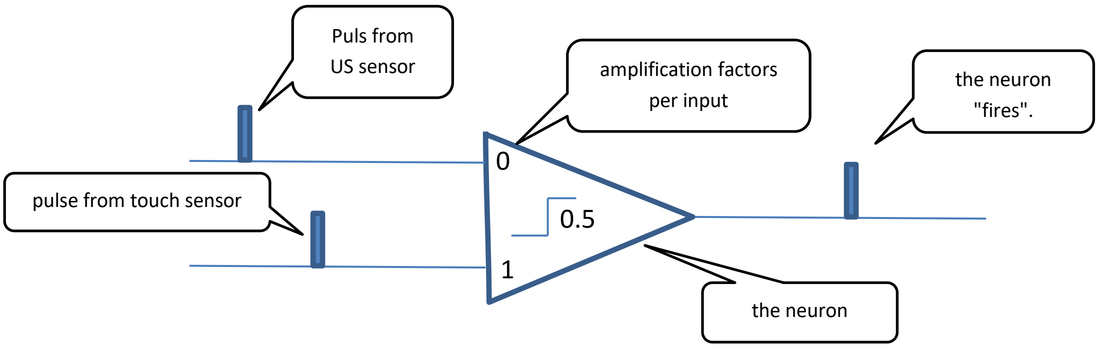


Now the red spray paint around the obstacles and walls comes into play. This is to mark areas where an ultrasonic sensor receives echoes from the objects. So, we equip the robot with four ultrasonic sensors that respond to this red paint. We call them *USsenorN*, ...

The robot should learn that an ultrasonic echo often precedes a collision, and that it is therefore better to turn back already at this echo. So, we need a mechanism that detects that an echo came before a collision. One way to achieve this is to have a counter in the ultrasonic sensor that is set to an initial value (here: 100) when it detects red color (i.e., an echo). This counter is continuously counted down to zero - and, if necessary, increased again beforehand. If this counter has a value greater than zero during a collision, then the echo has been received shortly beforehand.

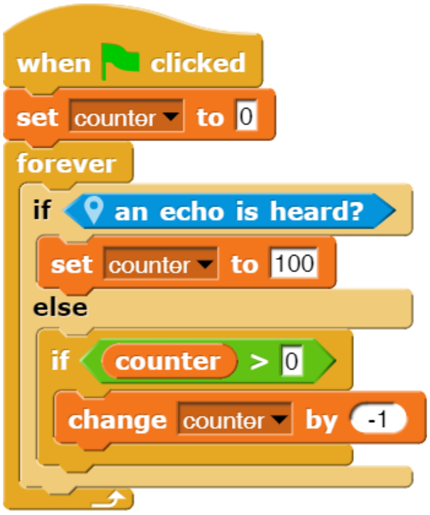


This constellation initiates a learning step that takes place in a *neuron*. This neuron has two inputs, which come from the associated touch sensor or ultrasonic sensor and are each assigned a *weight*, as well as a *threshold value*. The line from the touch sensor has the weight 1. If a signal e.g. of the strength 1 comes from there, then this is multiplied by the weight 1. The result is greater than the threshold value (here: 0.5) and the neuron "fires". The weight of the US sensor initially has a value of 0. It is increased whenever the touch sensor detects that the counter of the associated ultrasonic sensor has a value greater than zero during a collision. If a sufficient number of such learning steps take place, the product of weight and signal also exceeds the threshold value of the neuron at the US sensor, and this also fires in this case.

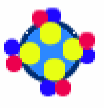


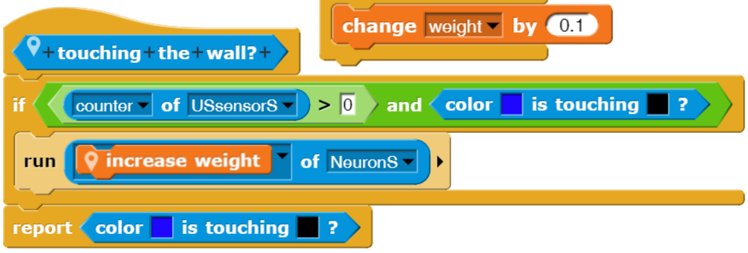
We now realize this form of *Pavlovian learning*.

The ultrasonic sensor works exactly as described above. The local attribute *counter* can be accessed directly with the *<attribute> of <object>* block. So, the actual changes take place in the touch sensors and the four associated neurons. Since these are clones of the only prototype in each case, it is almost sufficient to make the additions only in this one. The clones adopt them because they inherit the methods of the prototype. However, we must still indicate afterwards, to which element of the four groups the sprite is to react.



When touching a wall, we still have to determine whether the associated ultrasonic sensor triggered "just before". In the clones, we then overwrite the inherited "pale" method by adjusting the associated sensor. Thus, the paleness also disappears. Before we have cloned the ultrasonic sensor and the neuron three times and added the four new purple ultrasonic sensors and the yellow neurons to *Roby*. He looks like this now:

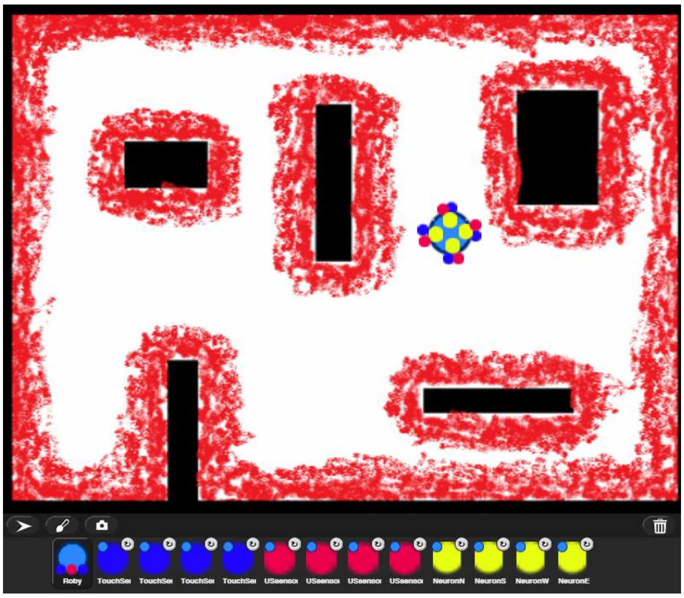




The neuron still needs a predicate *is firing?* that works as described above.

Finally, we change the behavior of *Roby*: he changes his direction when the corresponding neuron fires.

Roby is now looking for his way, initially between the obstacles, then along the "echo area". He has become really smart! 😉

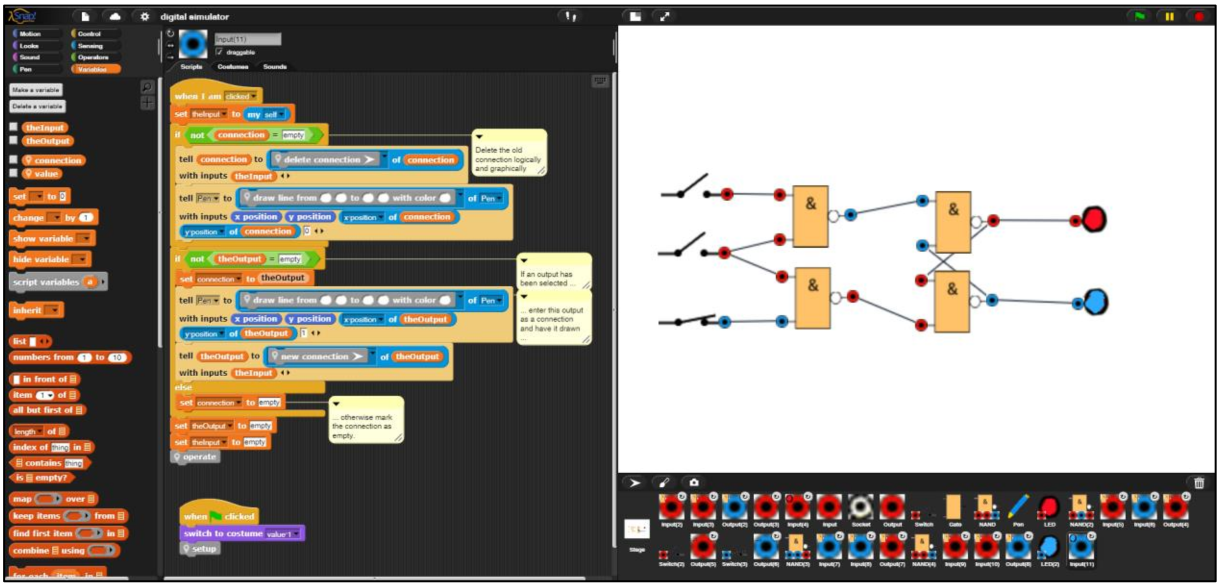


Tasks

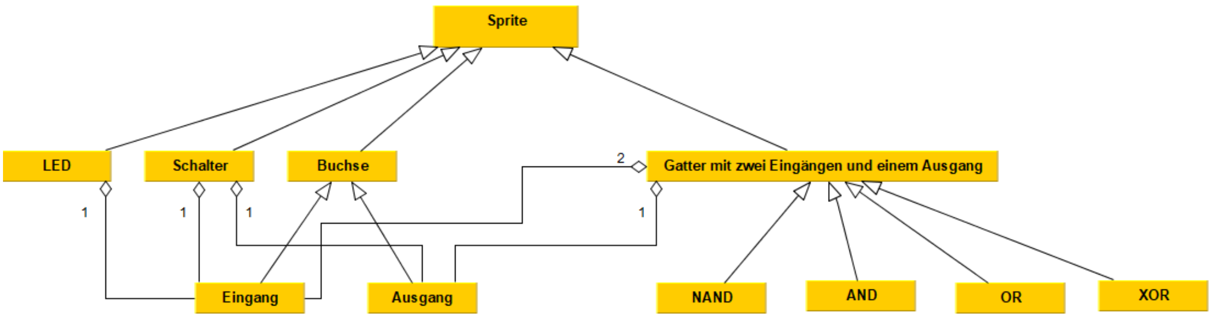
1. Give the program an **interface** that makes it easy to change the essential factors: the speed, the weights, the thresholds.
2. Introduce additional **sensor types** as well as additional events in addition to collisions.
   a: Let Roby find correlations between sensor values and events in different "worlds". Roby adapts to its environment this way.
   b: Discuss other ways that Roby is adapting to a changing environment.
3. Discuss the need for "**forgetting**" as well as ways to make this process happen.
4. Replace Roby with a mouse with a cheese sensor. Put them in a **labyrinth**. There she should search for the cheese.

8.4 A Digital Simulator

Level: *high school* Materials: *Digital simulator*

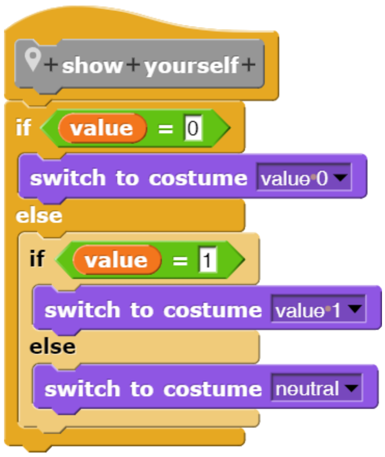


A *digital simulator* is a program that can be used to simulate digital circuits. It consists of switches, LEDs and gates, in this case only NANDs (Not AND), from which all other circuits can be built. On the components sit different kinds of sockets, with whose help signals are passed on. We can show the connections clearly in a (simplified) UML diagram. The inheritance is done in this case via delegation.



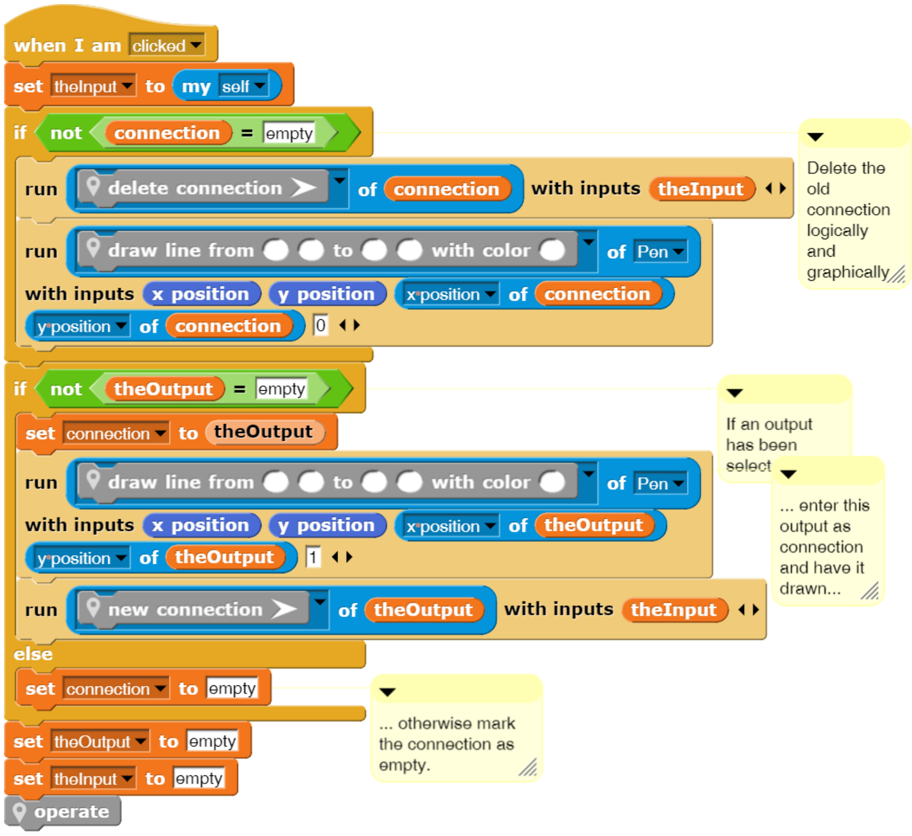
Sockets and Connections

As the "mother of all sockets" we draw a neutral *socket*, which serves as a prototype for *input* and *output* sockets. All sockets have a *value*, which can be *0* or *1*. Inputs get their value from the connected cable, or they get the value 1 for technical reasons if they are not connected. Outputs get their value from the component on which they are located. So, they represent the result of a logical circuit. All sockets inherit from the neutral socket the method *show yourself*, which represents their value in color, as well as the local variable *value*.

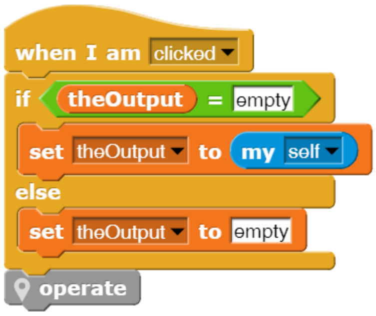


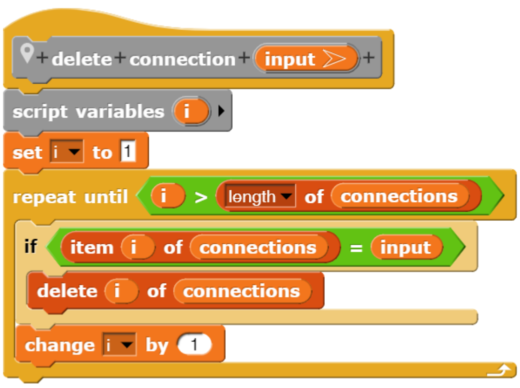
Using the context menu (*clone*), we create two clones of the neutral socket, which serve as prototypes for inputs and outputs in the following.

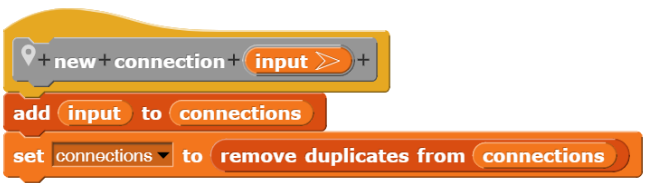
Sockets are to be connected by first clicking an output and then an input. If only the input is clicked, then its connection to an output is deleted - if it exists. Connections are displayed only rudimentarily as lines on the stage. If you move the switching elements afterwards, the lines remain "free in space"[49]. Inputs can at most be connected with an output. For this they get an additional variable *connection*. Outputs can distribute their values to several inputs; therefore, they receive a list variable *connections*, in which the connected inputs are entered or removed. If an output is clicked, then the global variable *theOutput* receives this output as value. If an input is clicked, then it provides for the update of the connections.



Outputs have it somewhat easier: they provide the capabilities to add and remove connections - and wait what comes.



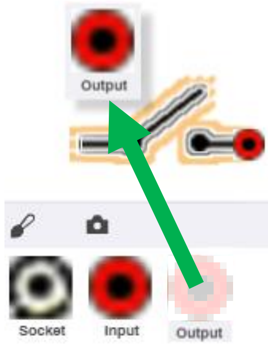




[49] The representation and especially the distribution of lines is an independent problem.

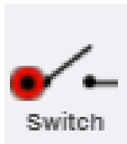
Switches

Switches are used to change input values. We create two costumes representing the open and closed state respectively. The left end of the switch we leave open, it symbolizes the connection to the ground and thus has the fixed value *0*. To the right end we add an output socket, which gets either the value *1* (state "open") or *0* (state "closed"). We get the new socket by cloning the output socket. Then we move the obtained sprite to the right place at the switch.



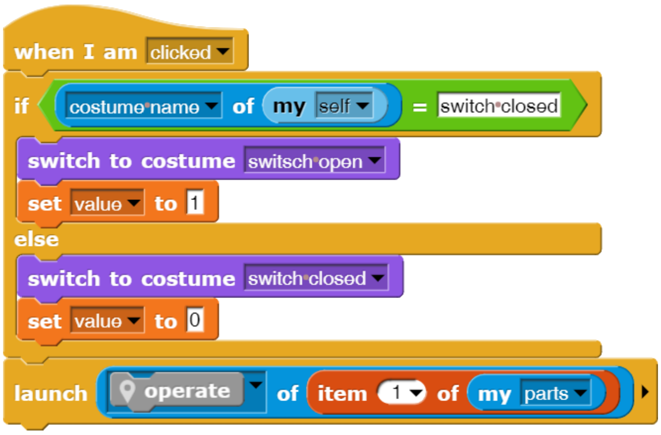
There it must now be anchored. To do this, we move the sprite symbol of the output from the sprite area over the switch in the output window. Its outline lights up when it realizes that it is meant. This attaches the socket to the switch: it is the *anchor* of the resulting *aggregation*.

So, an aggregation of sprites is created by <u>first</u> dragging the elements on the stage to the right places <u>and then</u> dragging the sprite symbols from the sprite coral onto the anchor element. The attached sprites (here: the socket) become elements of the *parts* list of the anchor (here: the switch) and are displayed at the sprite symbol of the anchor. With *detach from ...* from the context menu of the attached sprites they can be detached from the anchor again.

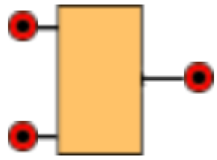


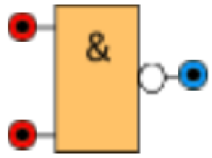
Since we want to operate the components of our digital simulator by mouse, it makes sense that the switch reacts to mouse clicks. This is easy to achieve: with every click he changes his costume. To do this, he must know what he looks like at the moment: with *<costume-name> of <myself>* he gets the current costume.

We still need a mechanism to control the reactions of the *parts*, in this case the output socket. Since it should be transferable, the method must be generally usable. So, we equip each part with an *operate* method and a variable *value*. If the state of the switch changes, then the switch changes its value. Finally, it calls the *operate* method of the output - which is the only element of its parts list here. We use the *launch* block to not let the program execution wait.



Gates

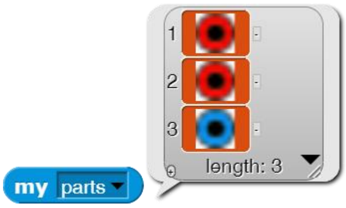


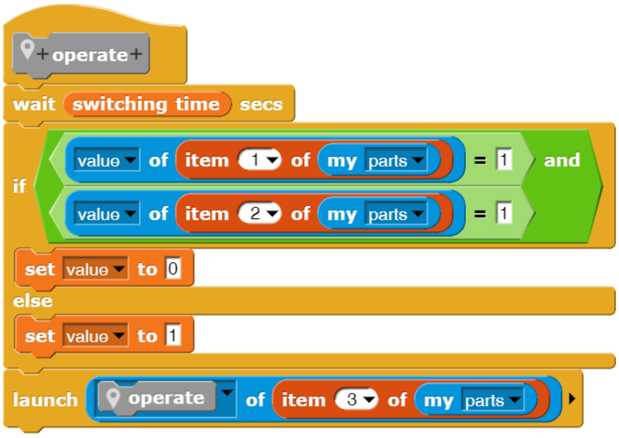


To create gates, we first introduce a prototype *Gate* that has two inputs and one output. Furthermore, it contains a variable *switching time*. We add the necessary sockets as we learned with the switches. From this gate we can derive other gates like AND, OR, XOR or NAND. For the NAND we create a clone of the gate called *NAND* and give it a customized costume.

The prototypes derived from the gate inherit the gate's *operate* method and the instance variable *value*. Both are actually useless, of course, since the gate has no real function at all. We therefore leave the method empty and overwrite it in the derived prototypes. (If we forgot something, we can also create variables and methods in the prototype afterwards. These are inherited to the clones immediately). Inherited variables appear slightly lighter in clones than own ones. If they are overwritten, then the changed elements get the normal color.

The *operate* method of the NAND is simple to write. The *my <parts>* block shows us the inputs and outputs of the NAND. We can read their values or set them as we did with the switch. We use the *launch*- instead of the *run*-block again.



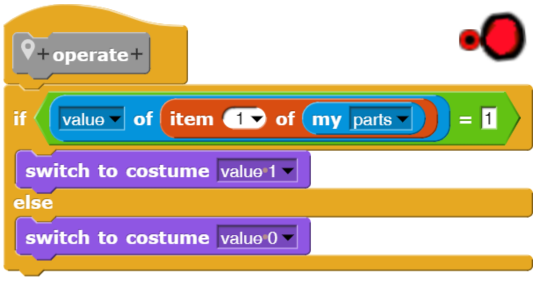


The Pen

The pen provides only a simple method of drawing straight lines in different colors on the stage. It has no other tasks.

LEDs

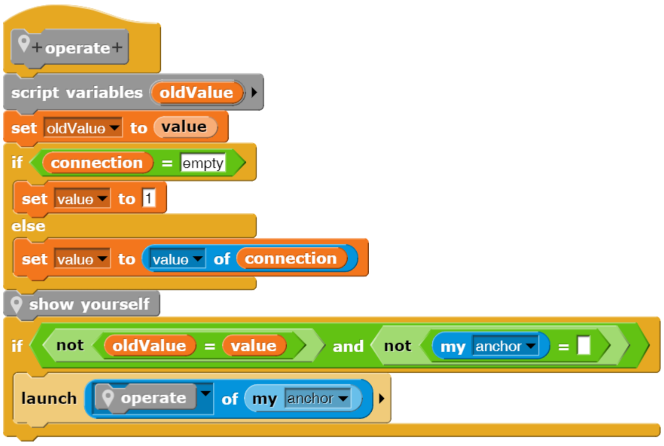
As a very simple example of adding new components to the system, we introduce the prototype of an *LED* (light emitting diode). This is given two costumes for the values *0* and *1* as well as an input. Since this one knows the system well, the LED can fully rely on it and limit itself to what LEDs do - glow. There is nothing more to do.

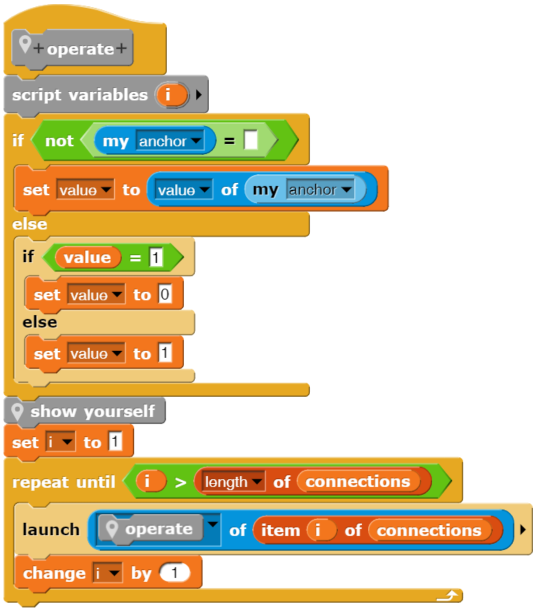


The Interaction of Components

The activity is supposed to wavelike pass our switching network in a *feed-forward* manner: Each component notifies the connected parts and calls their *operate* method when something has changed. If, for example, an output socket sits on a switch, then the switch calls the *operate* method of the output if it was clicked and therefore changed its value. This in turn activates all connected inputs. Each of these inputs calls the *operate* method of the gate it sits on - but only if its value has changed. If not, the wave is stopped here. So far, the gate can only be a NAND. This waits for its switching time, reads the values of its inputs, and activates the output - and so on.

The *operate* methods of input and output serve as examples.





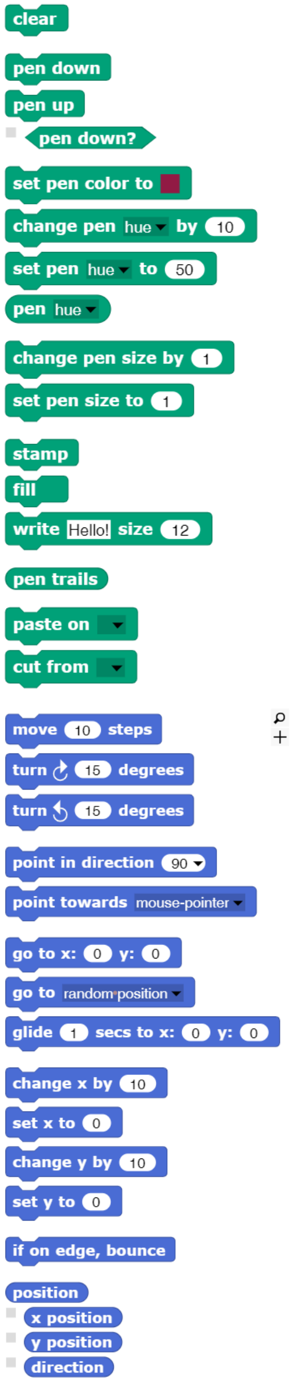
Tasks

1. Create prototypes for the following **gates** based on the NAND model.:
   a: an AND
   b: an OR
   c: an XOR
   d: a NOT-OR (NOR)
2. Create a prototype for **NOT** gates, which have only one input and one output.
3. Create a prototype for a **clock generator**. The clock frequency is to be adjustable.
4. Create a prototype for **RS-FlipFlops** (RS-FF). Inform yourself about their mode of operation beforehand.
5. Create a prototype for **JK-master-slave-flip-flops** (JK-FF). Find out about how they work beforehand.
6. Our gates react after a switching time, which can be different. Why actually?
7. Develop an **interface** with buttons, selection boxes, ... for the digital simulator, with the help of which components can be created and deleted, elements can be selected, the simulation can be started and stopped again, ...

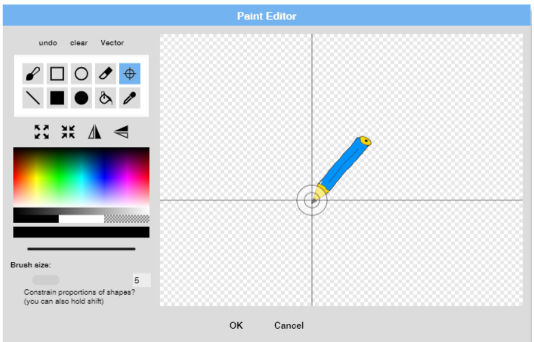
9 Graphics

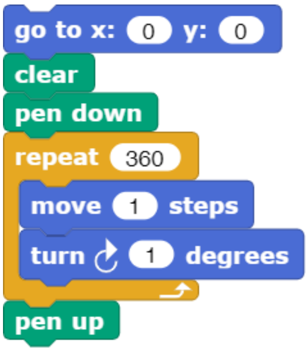
9.1 Line Graphics with Koch- and Hilbert Curve

Level: *from middle school* Materials: *Snowflake, Hilbert curve*

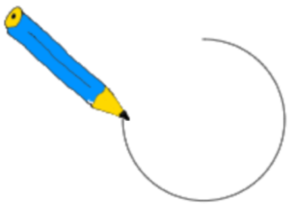


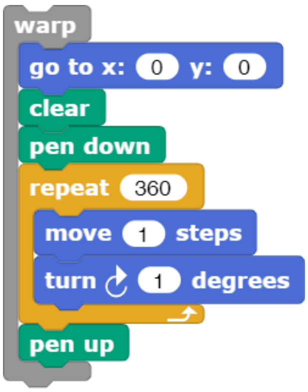
In *Snap!* each sprite has a (virtual) pencil to draw on the stage. The functionality corresponds to the well-known *Turtle graphics*[50]. The blocks for this can be found in the two palettes *Pen* and *Motion*. In the first one the pen is controlled, i.e. raised or lowered, pen color and width are set, ... In the second one the commands for moving the sprite are found. During this movement, the pen leaves "traces" depending on its state, which then form the generated line graphics - and which can also be further processed as *pen trails*. Note that the pen is located in the *rotation center* of the current costume of the sprite. You can move this in the costume editor using the crosshair tool.





If we choose the already known *pen* as costume, then the adjacent script creates a simple circle.

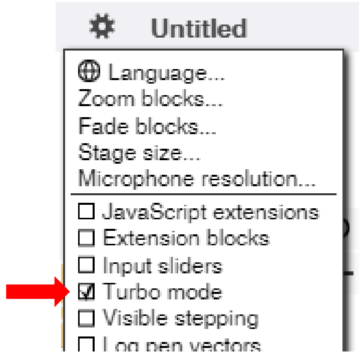




The example is a good way to demonstrate the effect of the warp block. While without it the pen draws the circle quite comfortably, the finished circle with warp block appears practically immediately. The reason is that in the first case the state of the system is shown anew after each block execution, while in the second case this happens only in larger intervals. The difference is "dramatic".

[50] https://de.wikipedia.org/wiki/Turtle-Grafik

A similar acceleration can be achieved using the *Turbo mode* option in the settings menu. However, this applies to the entire program execution and not only to a selected area.

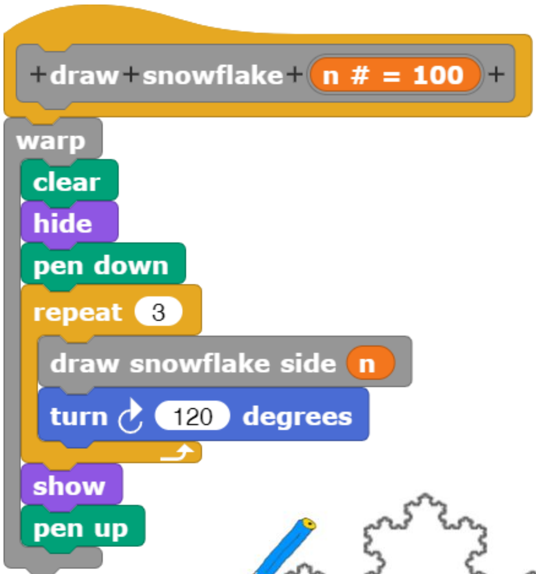


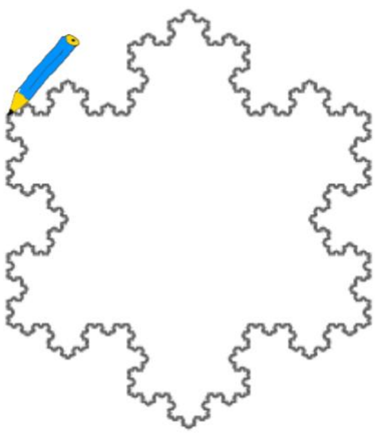
With the help of the Turtle graphics, some of the well-known recursive curves can be drawn very elegantly. We start with the *snowflake-* (or *Koch-*) *curve*. It is created by repeatedly "bulging out" triangles in the center of the sides of a triangle until the sides become too short for this process. In this case, the sides are drawn only as straight lines. A snowflake is created by assembling an equilateral "triangle" from three such sides.

Draw a snowflake side of length n

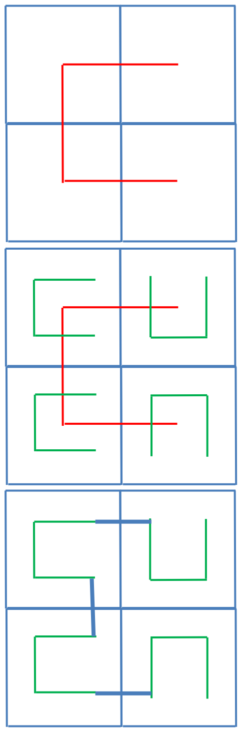
n < 2: true	n < 2: false
Draw a line of length n	Draw a snowflake side of the length n/3
	Turn by -60°
	Draw a snowflake side of the length n/3
	Turn by 120°
	Draw a snowflake side of the length n/3
	Turn by -60°

The procedure can be translated directly to *Snap!*

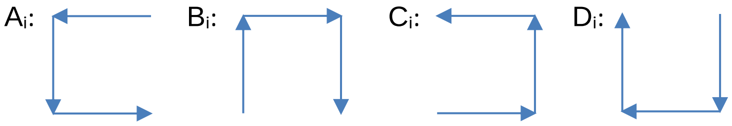




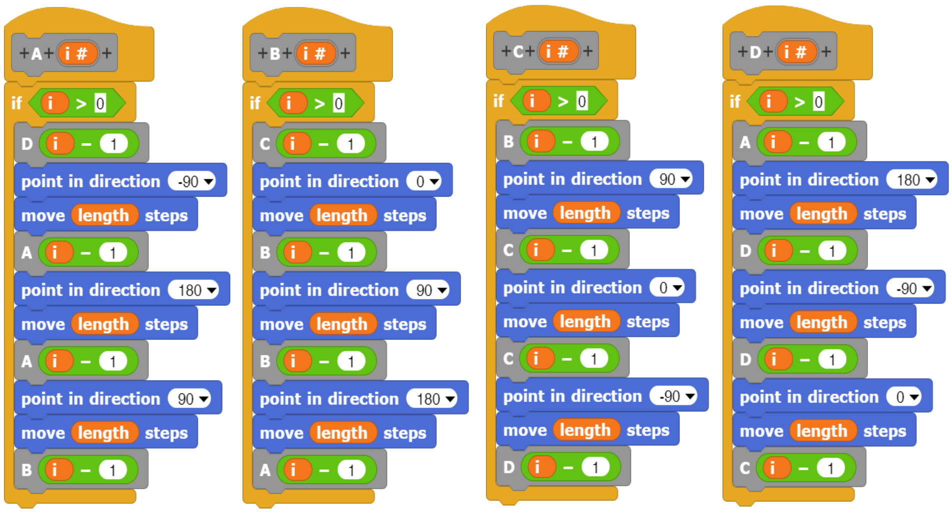
For the construction of the *Hilbert curve*, we use a version after László Böszörményi[51]. It is one of the area-filling curves, which has a kind of box as generator. The corners of the box lie in the centers of the four quadrants of a square. In the next stage, this box is reduced by half, and four versions of it are rearranged in mirrored or rotated versions in the quadrants. Finally, the smaller boxes are connected to each other as shown.

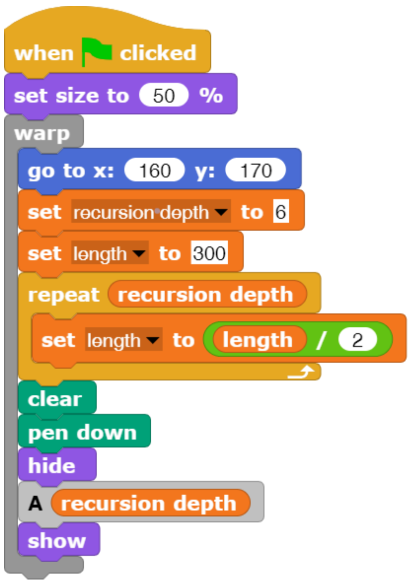


In Böszörményi's version, the boxes are marked A to D according to orientation and direction of rotation.

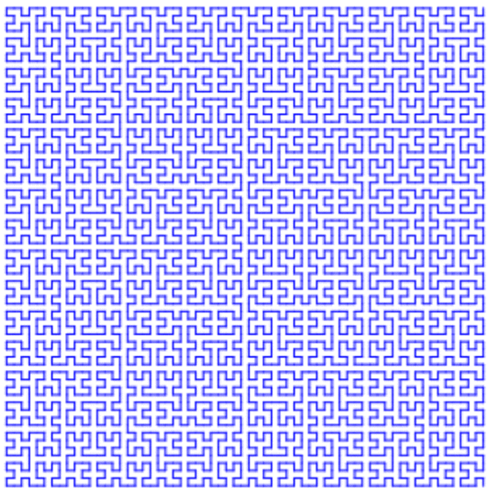


The Hilbert curve is composed of these elements by starting with A and calling the other elements "twisted". The parameter *i* indicates the recursion depth and thus the size of the elements. It is "counted down" to zero.





The call is made as described, after the sprite has been sent to the starting point right-up. The final length of the sections to be drawn is determined from the recursion depth - and then drawing takes place. Again, the effect of the *warp* block is drastic.

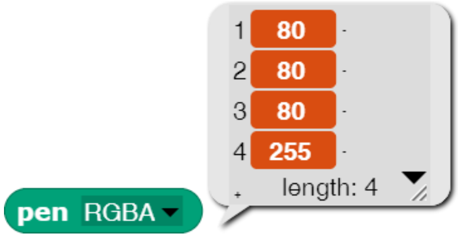


[51] http://bscwpub-itec.uni-klu.ac.at/pub/bscw.cgi/d11952/10.%20Rekursive%20Algorithmen.pdf

9.2 The RGB Color Cube

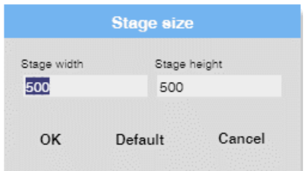
Level: *from middle school* Materials: *Color cube on stage*

Encouraged by this success, we will next try to create the familiar color cube of the RGB color space[52] ourselves. To do this, of course, we need to be able to set the RGB colors for the pen. We find ways to do this in the Pen palette. First, let's see how it represents RGB(A) colors. We already know that!

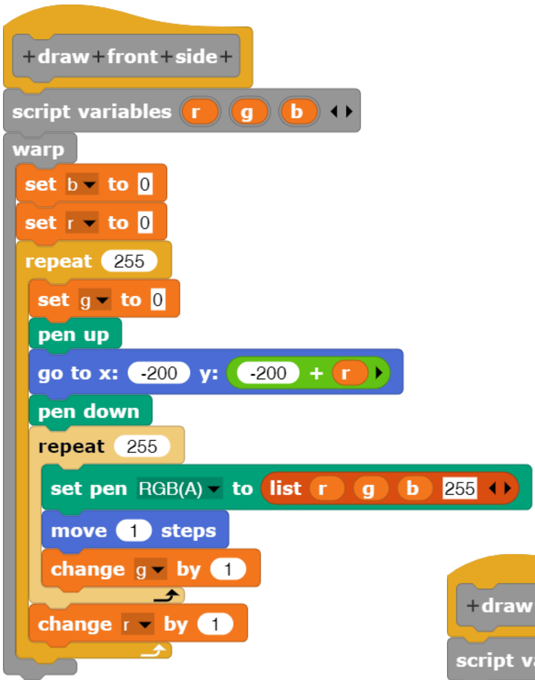


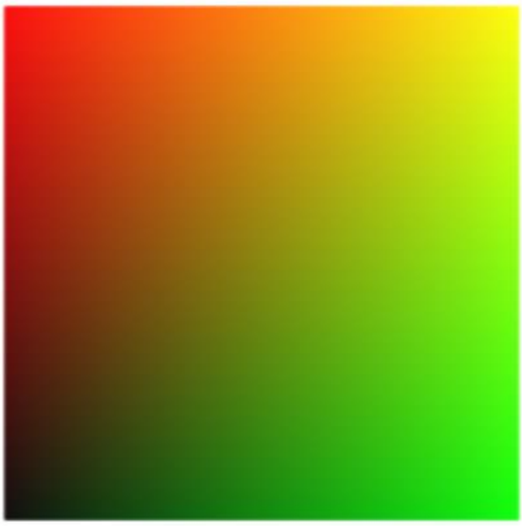
In the same way we can set the pen color as well.

Alright: We enlarge the stage to the dimensions 500 x 500 pixels by changing the corresponding entries in the Settings menu. Then we draw.

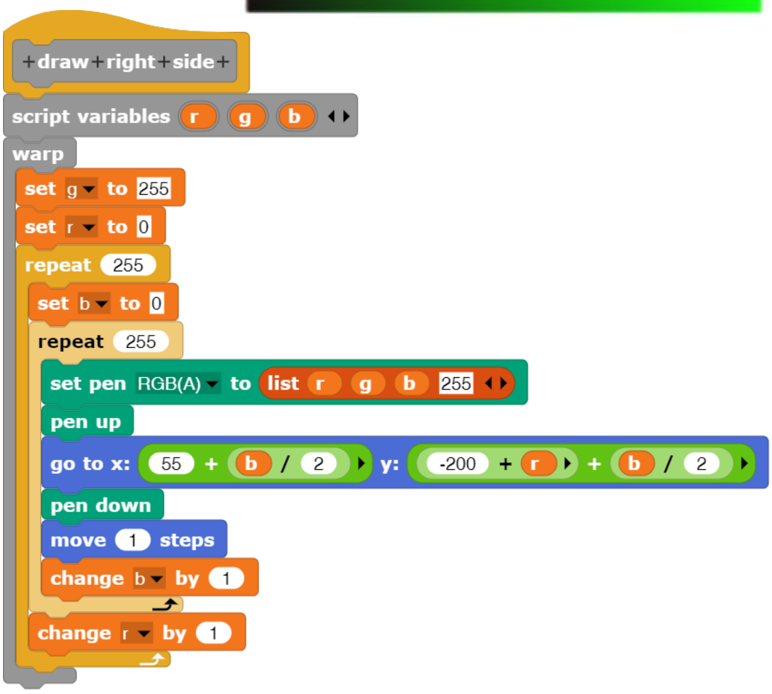


First of all, the front side of the color cube.



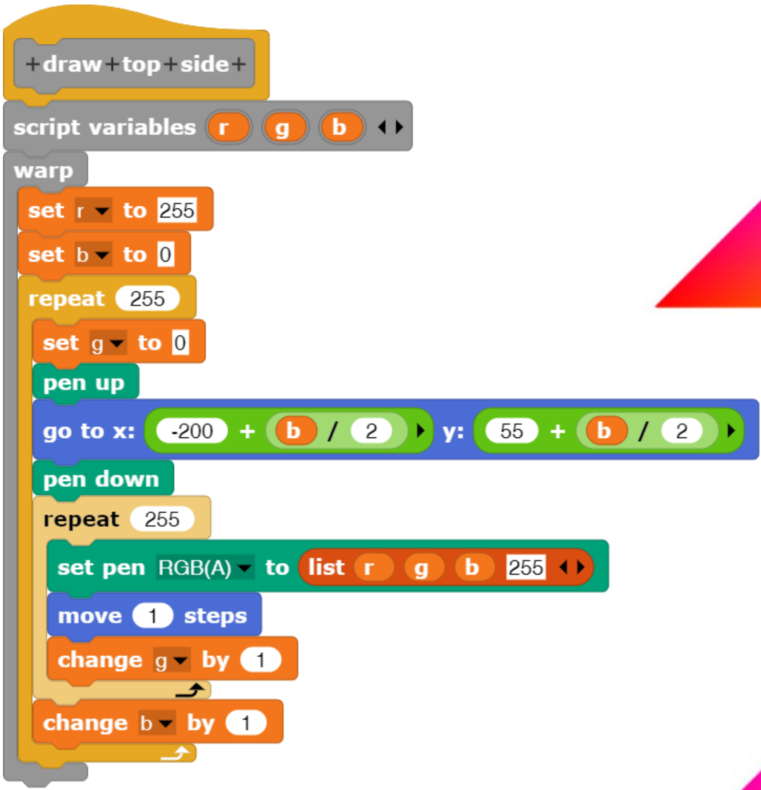


Then the right side.

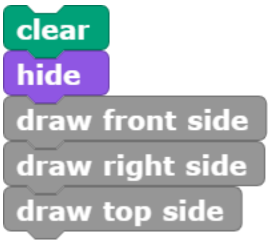


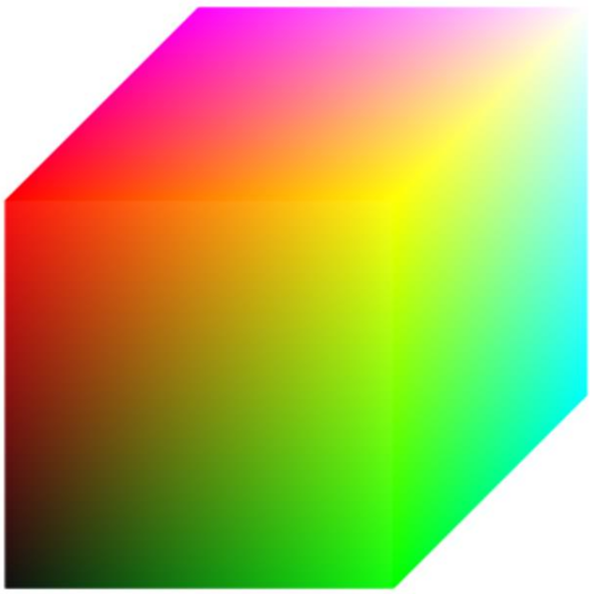
[52] https://de.wikipedia.org/wiki/RGB-Farbraum

And finally, the top on it.

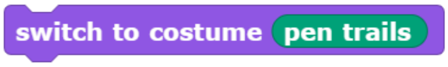


This results in the total RGB color space.



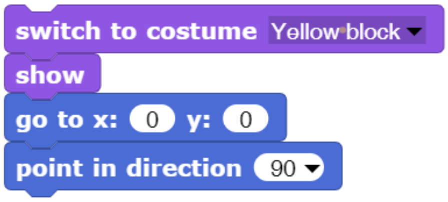


9.3 Printing and Cutting Costumes



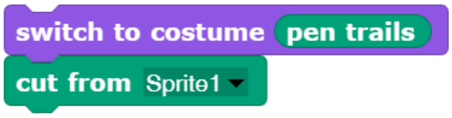
The pens of the sprites draw on the stage, but pixel graphics are also possible on costumes of sprites. With the help of the *pen trails* block, the current state of the stage can be transferred into a costume, which can also be "printed" back onto the stage if necessary. The *paste on* block prints the current costume of a sprite either onto the stage or onto a selected other sprite, as far as it overlaps with it. The block *cut from* cuts the area of the own costume out of the costume of the specified sprite.

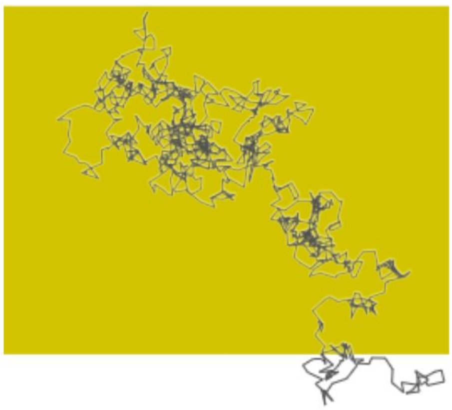
As an example, we first create a "scribble" on the stage.



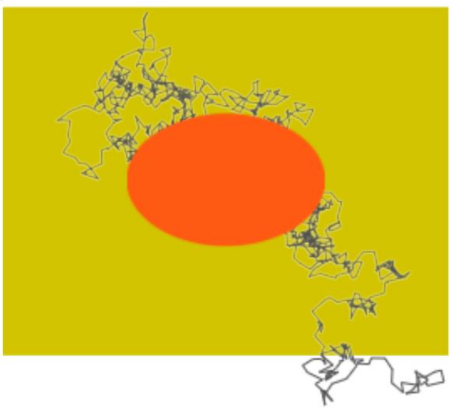
Next, we give the sprite a yellow rectangle as a costume and send it back to the center.

We create a second sprite and give it the costume of the *pen trails*. Then we cut this costume out of the yellow block.





Finally, we want to draw an ellipsoid on the yellow block. We give the second sprite an appropriate costume and "paste" it onto the yellow block.

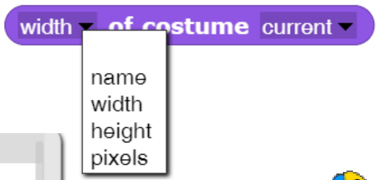


9.4 Drawing on Costumes - with an own JavaScript Library

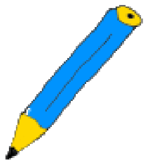
Level: *high school* Materials: *Color cube on costume*

Snap! is originally based on the *HSV color model*[53], similar to *Scratch*.[54] But I prefer the *RGB model*[55], because it corresponds directly to the color sensors in the human eye and many technical applications. But maybe also out of habit. 😉 Meanwhile *Snap!* supports both *HSV* and *RGB* representations of colors in the blocks.

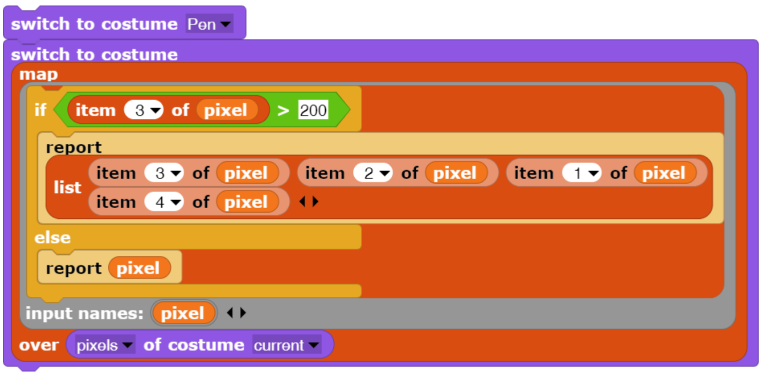
Using the *<property> of costume <costume>* block, we get access to the properties of a costume, such as its dimensions and its pixels. As an example, let's take our pen. Let's take a closer look at its pixels.



7200	A	B	C	D
141	0	0	0	0
142	0	0	0	0
143	0	0	0	0
144	0	0	0	0
145	73	72	70	255
146	110	99	40	255
147	175	152	23	255
148	240	207	22	255



We get a list that contains as elements 4-element lists with the three RGB values of the pixels as well as their transparency (alpha) values. All values come from the range 0...255, so each can be represented by a byte. For transparency[56], the value 0 means that the pixel is "invisible", and 255 means that it should be drawn with full colors. With the help of this pixel list, we now want to recolor the pen. We therefore swap the blue values with the red values, but only if the pixel is "quite blue".

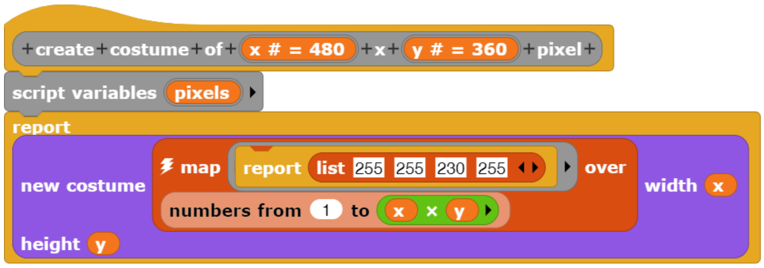


There we go!

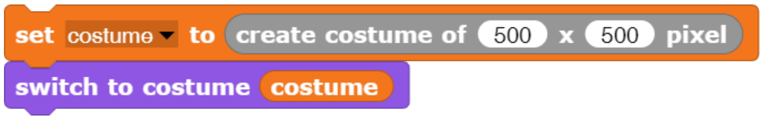
[53] https://de.wikipedia.org/wiki/HSV-Farbraum
[54] In the libraries of Snap! you can find more color models.
[55] https://de.wikipedia.org/wiki/RGB-Farbraum
[56] actually better: visibility

Drawing on costumes has, among other things, the advantage that *JavaScript* commands related to this area can be used without knowledge of and consideration for the rest of the *Snap!*. Thus, if necessary, one has a small playground where parts within the graphical language *Snap!* can be written in the text-based language *JavaScript*.[57] As an example we create the color cube again, but this time on a sprite costume.

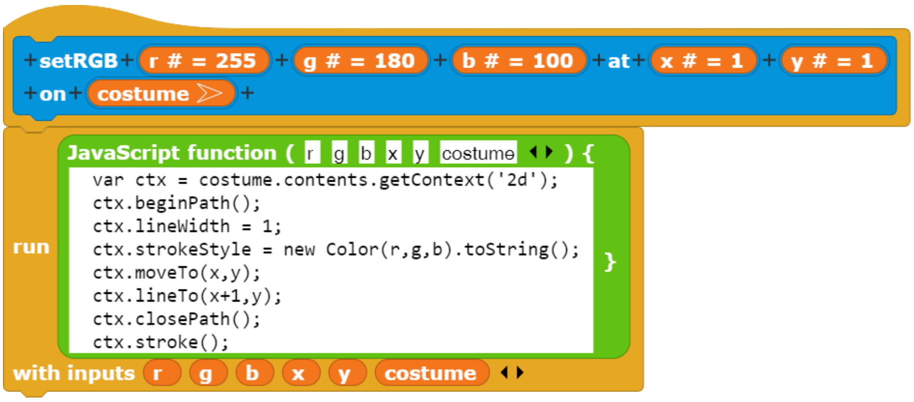
First of all, we need a costume: faint yellow and sufficiently large. We set the stage to 600x600 pixels and write a fast block for it.

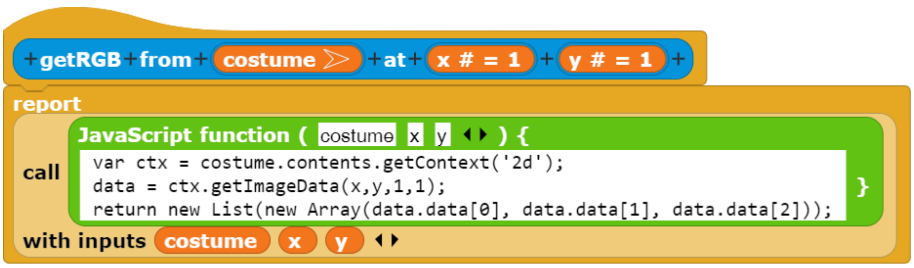


After creating the appropriate variables, we found the beginning of our script like this.



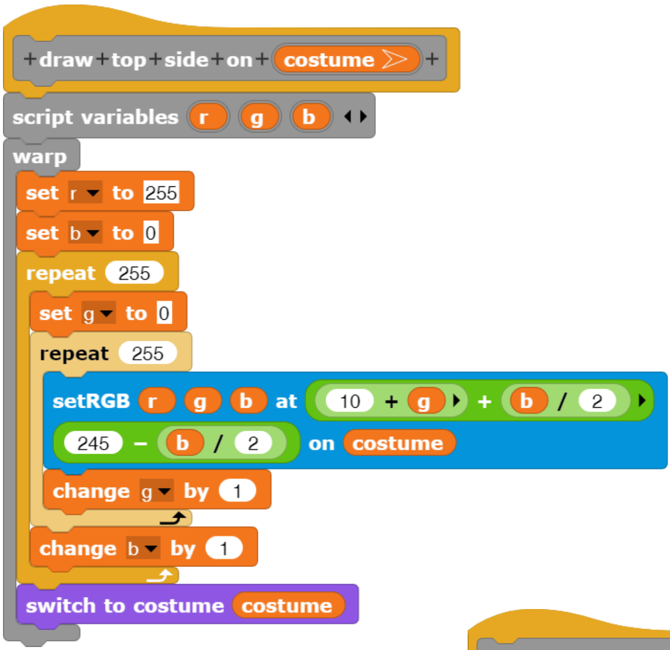
We can now manipulate the pixels of this costume. For this we write two small *JavaScript* methods to read and set the color of a pixel respectively.

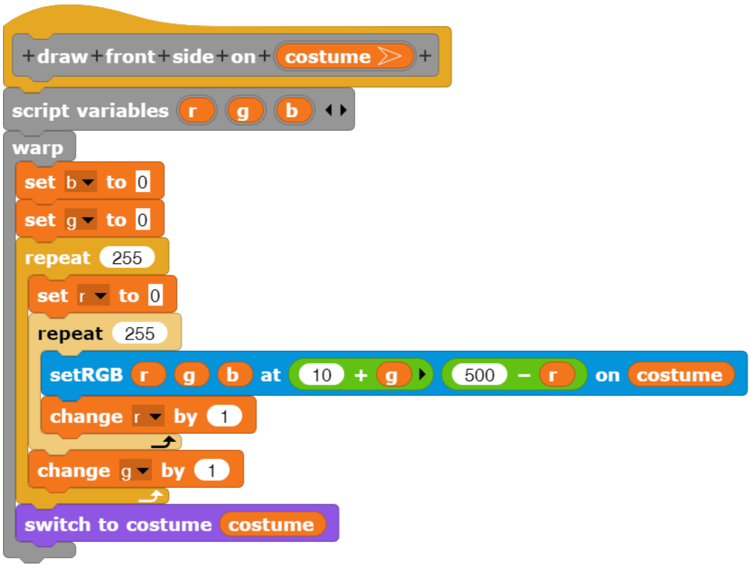




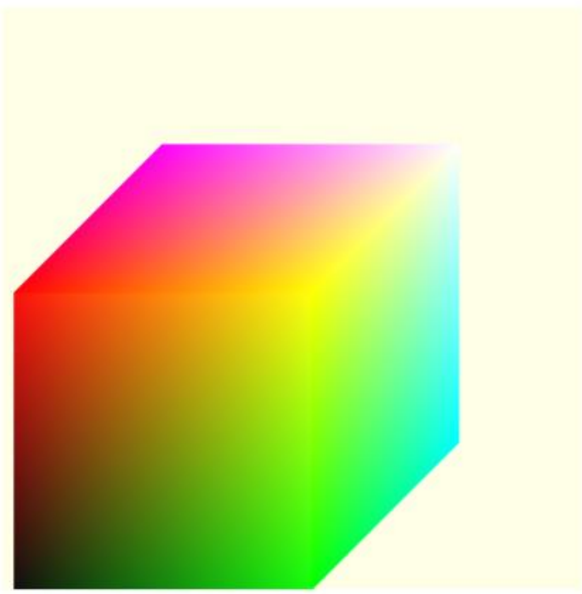
[57] If you want to. Snap! is now fast enough that such extensions can usually be dispensed with.

With these two methods we can now create the RGB color cube again, after we have allowed the use of *JavaScript* in the *Settings* menu.

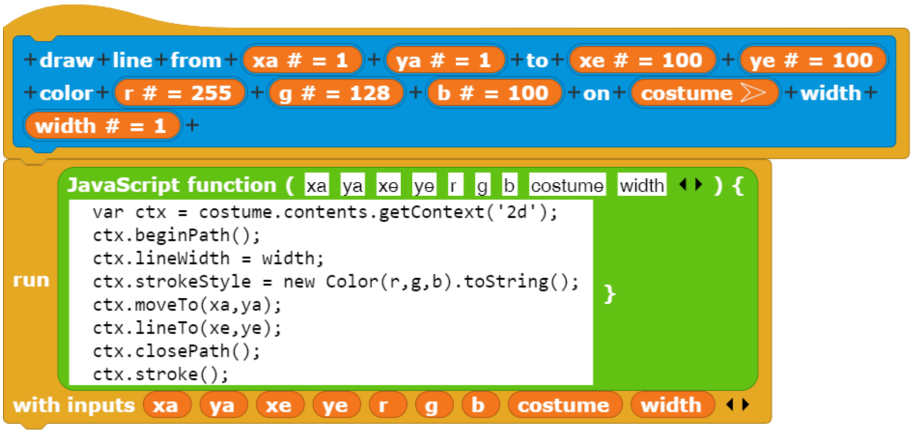


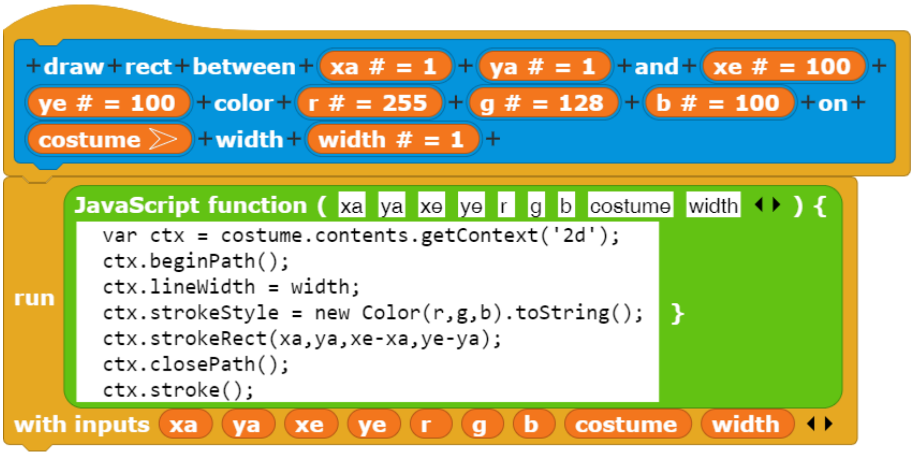


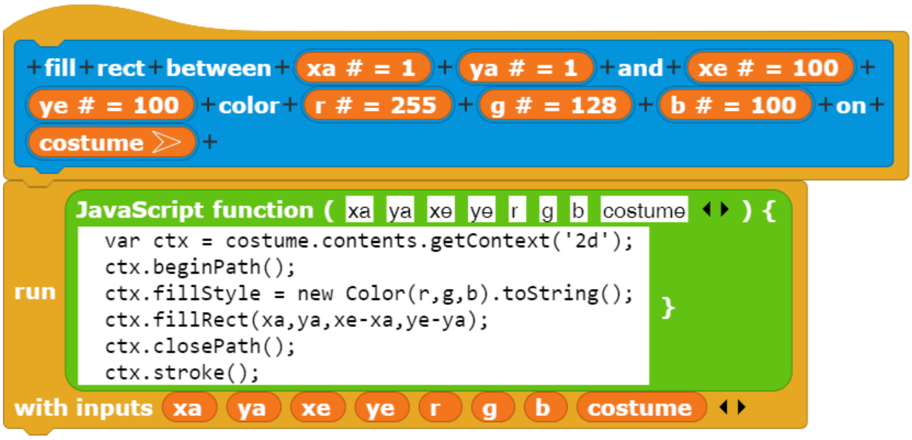
With this we can have the color cube drawn again.

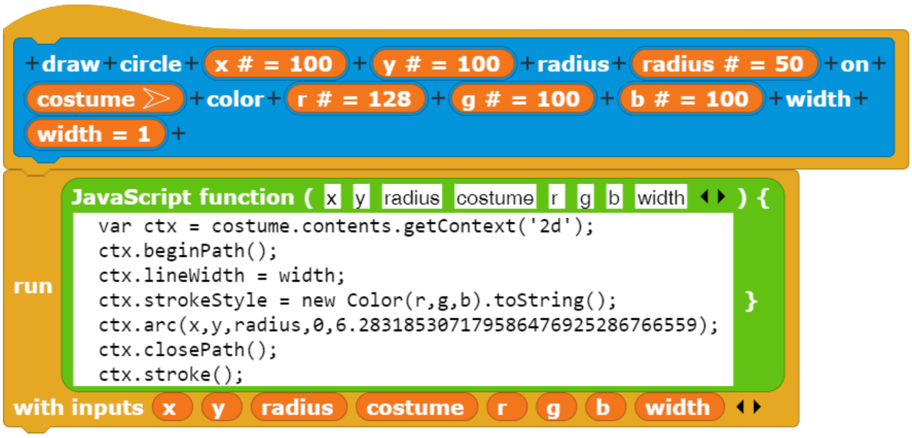


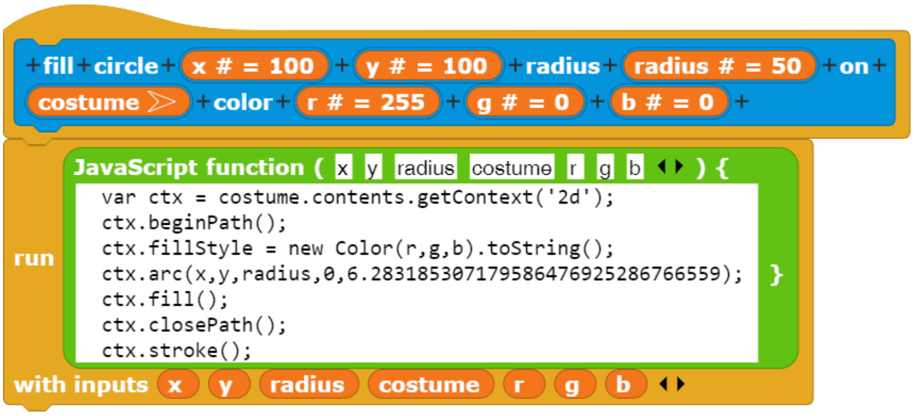
While we're at it, we might as well implement some more of the common graphics operations in *JavaScript*.



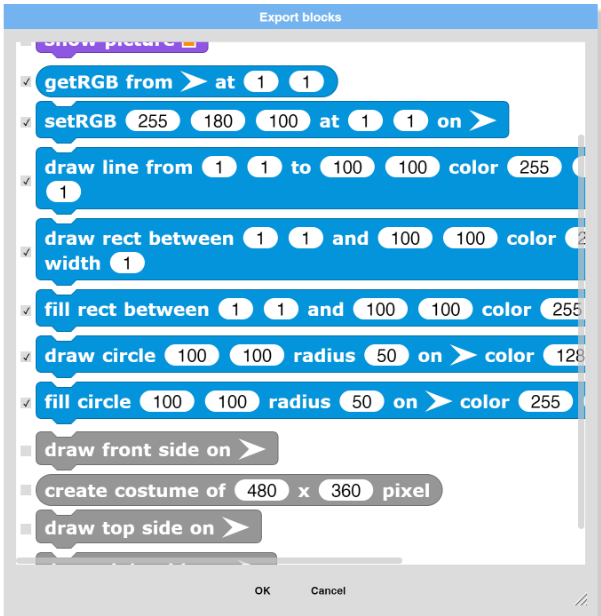








We save these blocks in a separate library (*File → Export blocks...*), selecting beforehand which blocks should be included in it. We rename the file saved in the download directory, e.g. to *MyOwnDrawingLibrary.xml* and move it to a suitable location. From there we can load the blocks into other projects via *File → Import...* and use them - just like any other library.



9.5 Drip Painting

Level: *high school* Materials: *Drip painting*

One of the methods of bringing randomness into artistic design in modern painting is to splash blobs of paint on the canvas with a brush. The impinging drops of paint are further divided on impact, resulting in a random pattern. We want to simulate this process, *drip painting* - and that's not so easy.

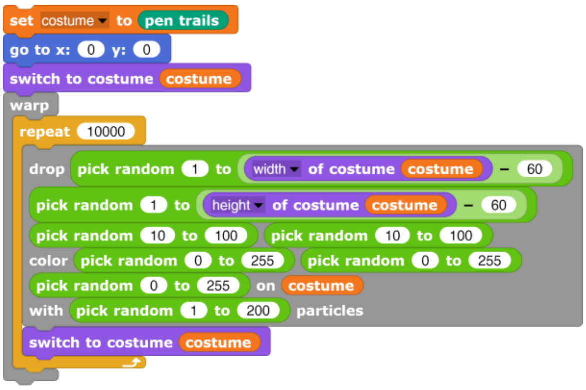
We try a simple but computationally very expensive approach: Within a rectangle, *n* random circular spots with slightly different colors are created, which become more transparent towards the edges of the rectangle. Eventually, the color thickness decreases there. Since *n* is in the order of *100* and we want to distribute a few thousand drops per image, we transfer the drop drawing to a JavaScript function that can do something like this very fast.

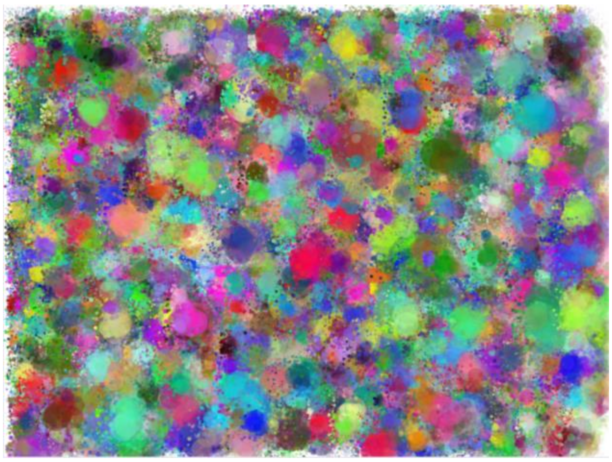
```
+drop+ xa # + ya # + br # + ho # +nl+color+ r # + g # + b # +on
+ costume +nl+with+ n # +particles+
run
JavaScript function ( xa ya br ho r g b costume n ) {
  var ctx = costume.contents.getContext('2d');
  var radius = Math.min(br,ho)/4;

  var xm = xa + br/2;
  var ym = ya + ho/2;
  for(i=0; i < n; i++)
  {
    ctx.beginPath();
    x = xa+Math.random()*br;
    y = ya+Math.random()*ho;
    dist = Math.sqrt((x-xm)*(x-xm)+(y-ym)*(y-ym));
    if(dist<radius) crad = Math.random()*radius;
    else crad = Math.random()*5*radius/dist;
    ctx.fillStyle = new Color(r+50-100*Math.random(),g+50-100*Math.random(),b+50-100*Math.random()).toString();
    ctx.strokeStyle = ctx.fillStyle;
    alpha = 1 - Math.sqrt((x-xa)/br);
    if(alpha < 0.01) alpha = 0.01;
    ctx.globalAlpha = alpha;
    ctx.arc(x,y,Math.abs(crad),0,2*Math.PI);
    ctx.fill();
    ctx.closePath()
  }
  ctx.stroke();
}
with inputs xa ya br ho r g b costume n
```

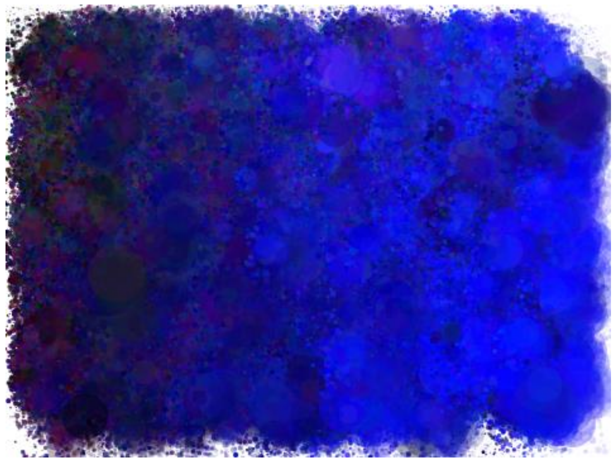
As parameters we pass the coordinates of the upper-left corner in the also passed costume, the width and height of the rectangle circumscribing the drop, the three RGB color values and the number of "partial drops". In the function (as known by now) the 2D graphic context is determined and a radius for the core area of the drop is calculated. Then the coordinates of the image center are determined, and *n* partial drops are drawn, whose positions, radii, colors and transparency are chosen randomly. A strongly enlarged "drop" looks then e.g. like this:

Of these drops we now distribute a few thousand on the canvas - and get an optimistic abstract *Spring Picture*.

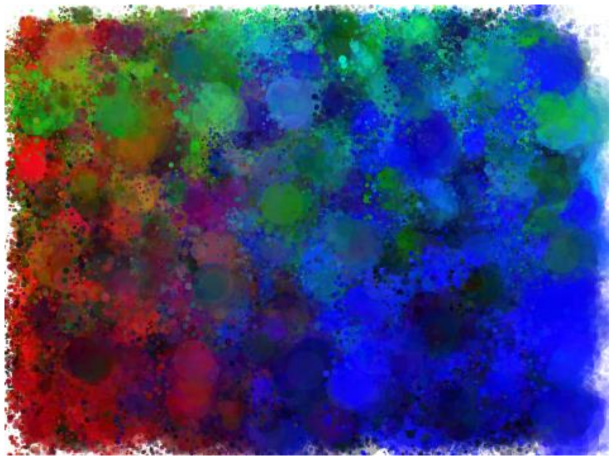




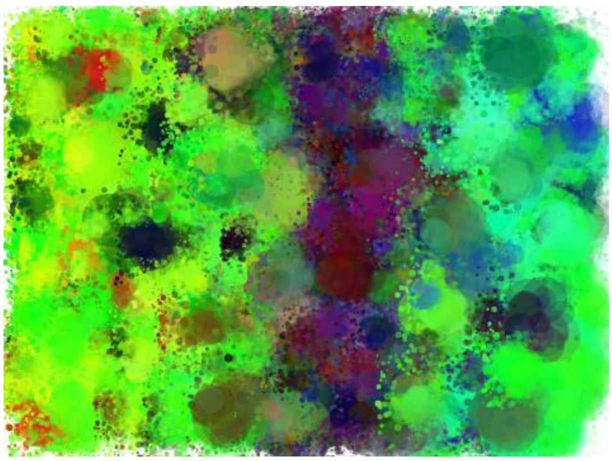
But of course, we can also make the color distribution dependent on the position - and get *Some Red and a lot of Blue*.



With some green to it: *Untitled 37*.



And of course, you can become more courageous: *Balancing Act* 😉

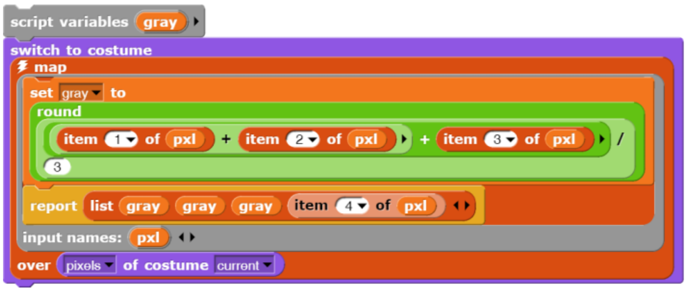


9.6 Edge Detection

Level: *high school* Materials: *Edge detection*

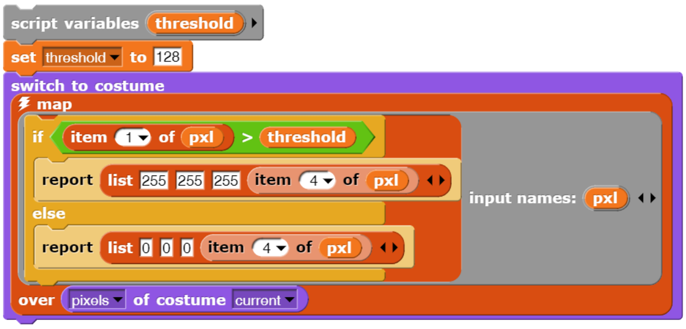
In order to recognize objects in an image, it is often helpful to highlight the boundaries of these objects - the *edges*. One way to do this is to 1) convert to a *grayscale image*, 2) use a threshold to convert to a *black and white image*, and 3) *detect edges* in that black and white image. The first two steps can be done relatively fast in *Snap!* using the *map...over* function, the third requires a lot of computation, so there are plenty of opportunities for coffee breaks. Or, after developing the procedure in *Snap!*, we transfer this task to a *JavaScript* function. Edge detection is a precursor to object detection. The recognition of the license plate of a motor vehicle on a video image can serve as an example.

We get an image that has edges that are clearly visible and load it as our sprite's costume. Next, we switch to a copy of the costume to preserve the original. We determine the width, height, and the pixel list of the image with the block provided for this purpose from the *Looks* palette.

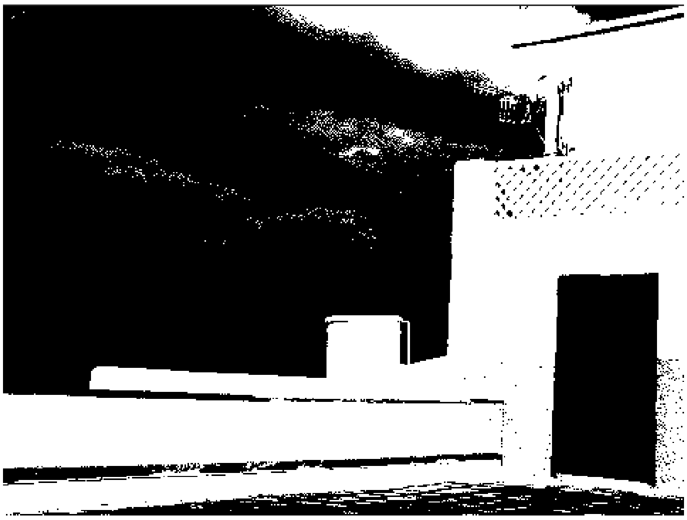


This image is to be converted into a grayscale image. We can achieve this step by step by processing the individual pixels - a typical task for the *map...over* function, here using the precompiled version. This needs a script that can apply it to the individual list items. It calculates the average *gray* value of the three RGB values and assigns it to the three color channels. It leaves the transparency value unchanged.

A black and white image is to be created from the grayscale image. To do this, we specify a *threshold value*. All gray values greater than the threshold are set to full white, the others to black. Also, for this we write a function that is run by *map...over*.

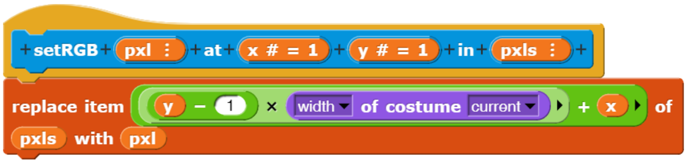


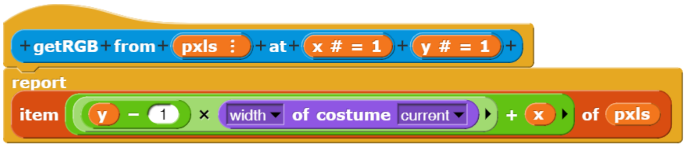
In the black and white image, some repair work should still be done: individual isolated points should be deleted, line gaps should be closed, and so on. (see Tasks). We will do without that here.

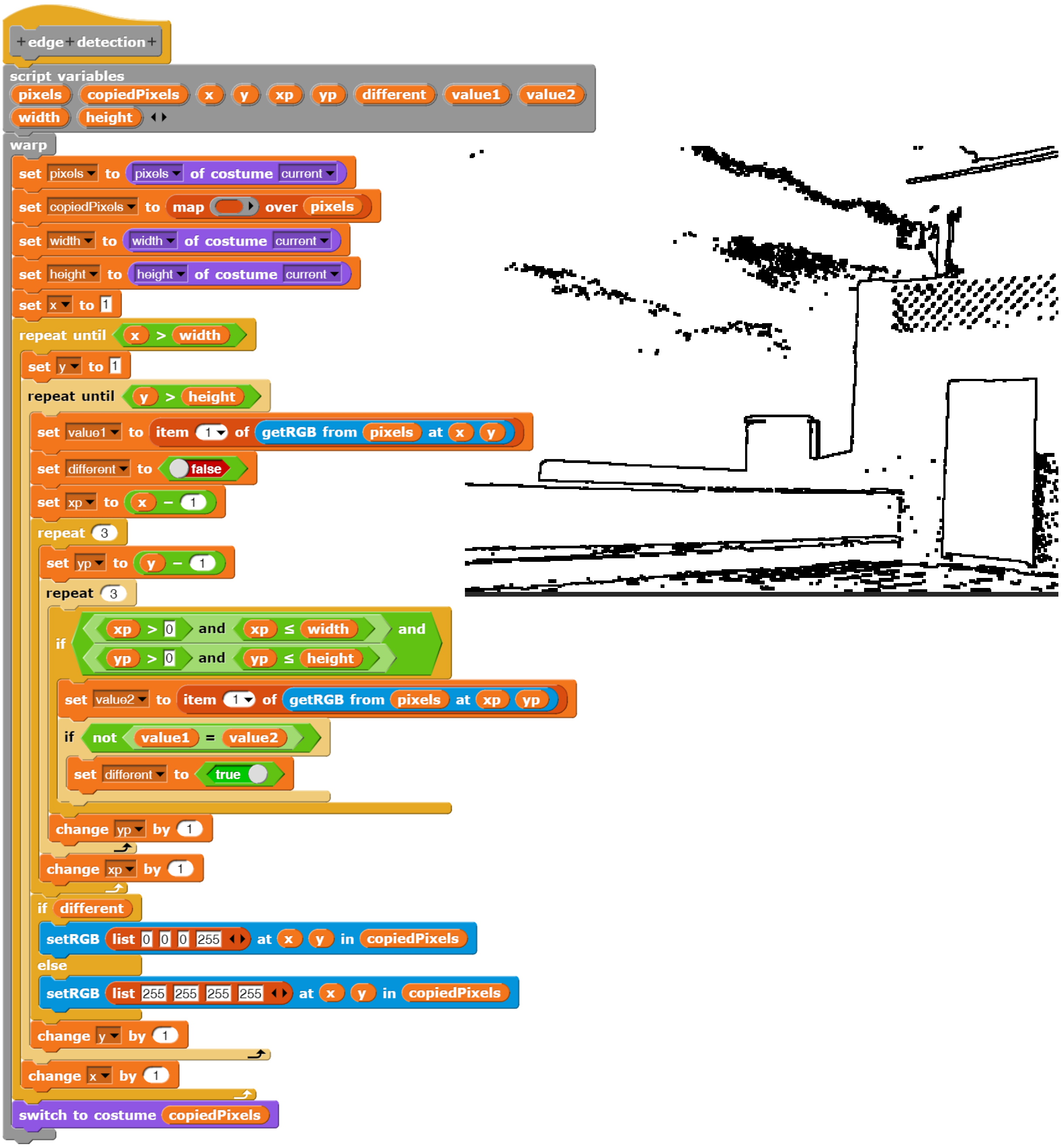


The last step is to find the edges in the black and white image. To do this, we examine the surroundings of each pixel. If all points have the same color as the pixel, then this pixel is located in an area and is drawn white. If at least one differently colored pixel is found, then we have found an edge pixel and color it black. Since changes in pixel values affect the neighborhood, the changes are made in a list *copiedPixels*.

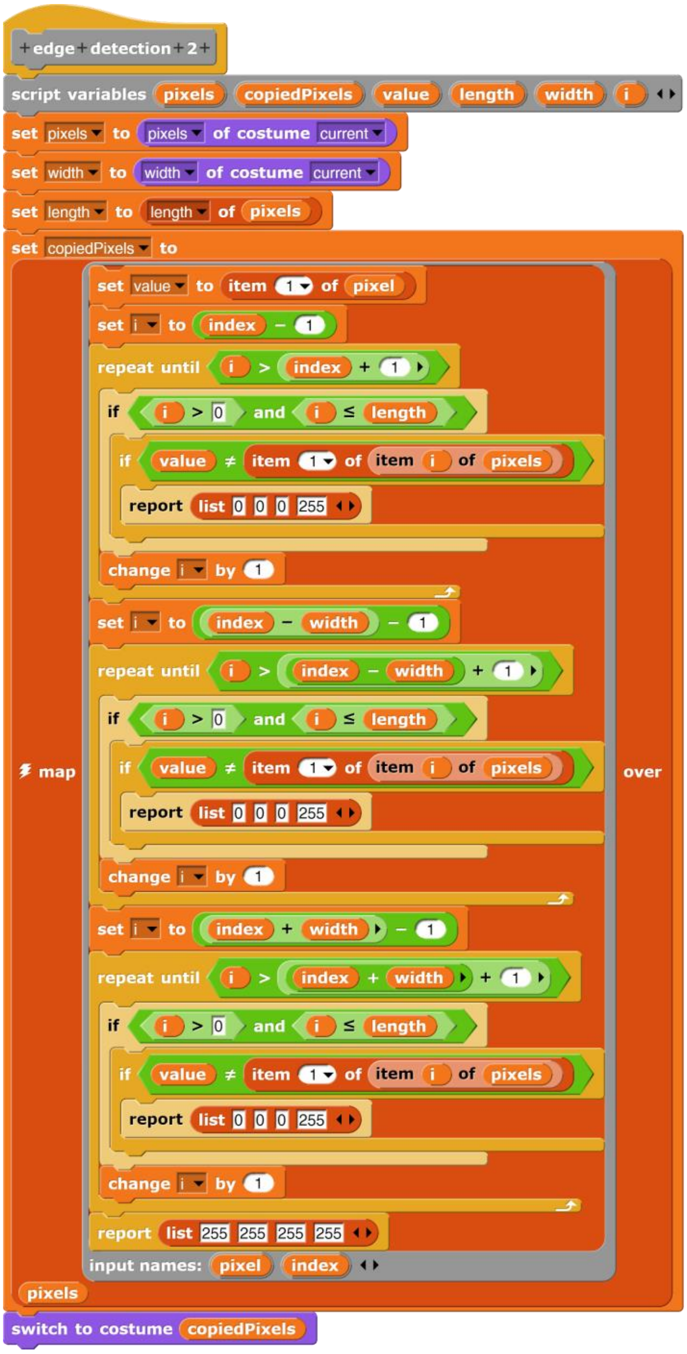
First of all, we need to have access to the individual pixels via their coordinates in the image. We could use the *JavaScript* graphics library developed earlier for this, but we want to create two new blocks for it here. We will use them in a block *edge detection*.







In edge detection we have very traditionally examined the environments of all points, which takes a correspondingly long time. However, we can just as well examine the pixel list sequentially, taking into account that we find the neighbors of a pixel partly next to the pixel itself, partly shifted "to the left" or "to the right" by one image width. The sequential run allows us to use the *map...over* block in the precompiled version. Let's see if this is worth the effort. For the sake of brevity, we won't check here if we have an edge pixel. So, we treat the image as a torus.



Examine the three pixels in the center.

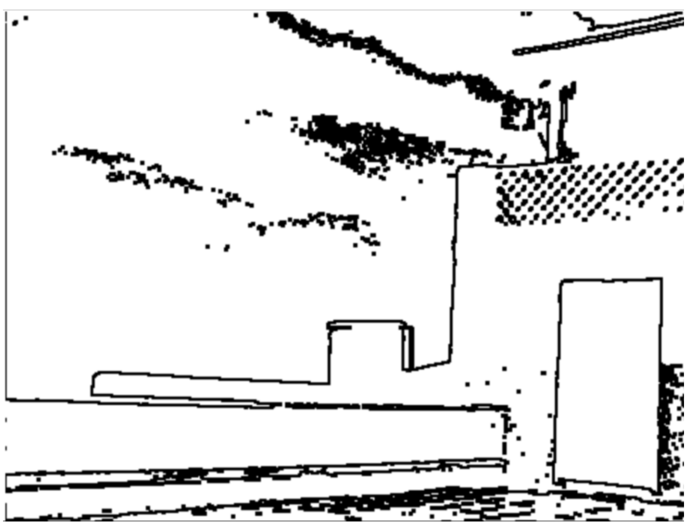
If a color difference was found, return a black pixel.

Now the one in the upper row above the pixel under consideration, …

… and then the three below.

If no differences were found, return a white pixel.

We get - with small deviations at the edges - the same image as before. However, this time the processing took only half as long.



9.7 Tasks

1. a: Find out about the **C-curve** on the Internet.
   b: Try some steps to construct the curve "by hand".
   c: Implement a script to draw the curve in *Snap*.
   d: Proceed accordingly for the **Dragon curve**, the **Peano curve** and the **Sierpinski curve**.

2. Display the RGB cube **from another point of view** so that the three previously hidden sides become visible.

3. If you want to try your hand at JavaScript: create color gradients and, if necessary, the RGB color cube in a **JavaScript function**.

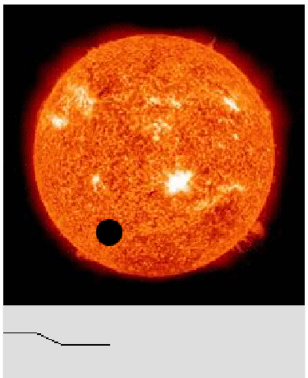
4. Change the color values iteratively, i.e. without the *map* function, by accessing the individual pixels. Measure the **execution times** for the different methods.

5. Some painters apply the colors with a spatula. Create "**spatula pictures**" that "spill out" in one direction and can contain multiple colors. Create random pictures from them.

6. a: Delete individual **isolated pixels** in black and white images.
   b: If you delete all **edge points** in black and white images ("melt off" the edges) and then add points to all edge points again - or vice versa, then you can delete single pixels, close gaps in lines, etc. by alternately and possibly repeatedly applying the procedures. Implement the procedures and test them.

7. If you want to program something in JavaScript:
   a: Implement the **conversion of grayscale images** to black and white images as a JavaScript function. The threshold value is to be given by a variable in slider representation.
   b: Implement **edge detection** as a JavaScript function.

8. Extrasolar planets are usually discovered when they darken their sun a bit as they pass between their star and Earth. Get an image of the Sun and make a black circle, the planet, pass in front of the Sun. Count the number of bright pixels visible in each case and plot the results of the **planetary transit** on a graph.



10 Image Recognition

The following three examples represent a sequence in which, with increasing difficulty, some possibilities of *Snap!* in image processing are shown. Problems were chosen that provide access to the current discussion of digital media and are thus relevant to the field of *computer science and society*.

10.1 A Barcode Scanner[58]

Level: *from middle school* Materials: *Barcode reader*

We want to analyze a barcode, as used on the labels of goods in a supermarket, with the help of a "laser" (a red dot) and convert it into a string of characters. First of all, let's have a look at the planned setup. We should not miss the very small red dot on the left side of the working area - this is the "laser"!

What is EAN code?

The European Article Number (EAN) code comes in different variants. Here we consider the EAN-8 code, which consists of 8 digits, the last of which is a check digit[59]. The digits are represented by four black and white stripes of different width. The space between two black bars is therefore also part of the code! On the left and on the right of the barcode there are two black and one white bar in between as delimiters. The center is marked by five such bars. All of them have the width "1". The code was chosen so that all the digits have a total width of "7". We will not go into further details here.

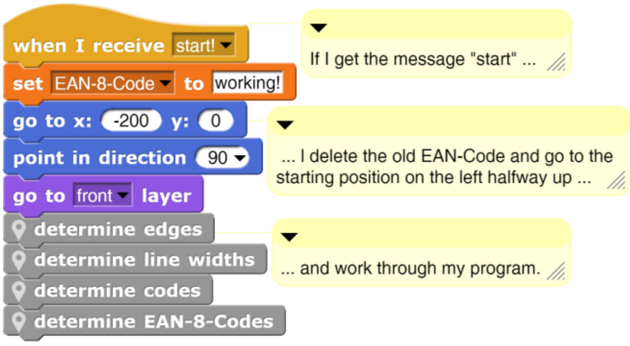
To determine the coded numbers, the laser dot is moved across the code from left to right. It "measures" the positions of the color changes and enters them in a list. The bar widths are calculated from this list. Since the first three bars have the width "1", we can determine this value quite well by averaging. The other line widths are multiples of this unit. Four bars each result in the code of a digit, which we determine using the table. The procedure can be summarized succinctly in the form of a structogram.

EAN-8-Codetabelle	
Ziffer	Code
0	3211
1	2221
2	2122
3	1411
4	1132
5	1231
6	1114
7	1312
8	1213
9	3112

Determine the x-positions of the edges of the black and white lines.
Determine the line widths from these, deleting the markings in the process.
Determine from these the eight four-digit codes of the numbers.
Determine the EAN code from these.

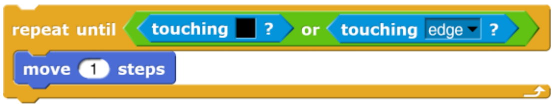
[58] Partly according to E. Modrow, The SQLsnap supermarket, Scratch2015 Amsterdam
[59] Siehe z. B. https://de.wikipedia.org/wiki/European_Article_Number

Converted into a *Snap!* script of the laser we get:

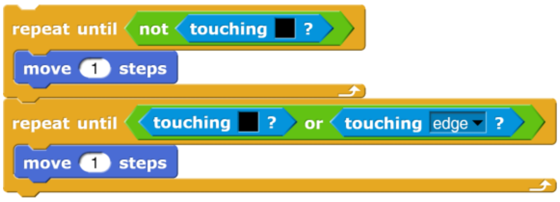


To do this, we pressed the "*Make a variable*" button in the *Variables* palette of *Snap!*, entered the variable name *EAN-8-Code* in the window that popped up, and marked this variable as local ("*for this sprite only*"). Since it is not needed for any other object, we limit its validity to the scripts of the laser. After that, the variable appears in the *Variables* palette. While we're at it, we'll also create three other variables with the names *edges*, *line widths* and *encoding*. The check mark in front of the *EAN-8-Code* variable means that the variable will be displayed in the output window. There we can still change its appearance in the context menu (right click on the variable). We drag the first block under the variable names *set <variable> to <value>* into the script area. Using the small black arrow, we can then select a variable identifier that is "visible" to the laser and specify a value for it. If we now click on the block, it will be executed and the variable will get the desired value, which can be seen immediately in the output area.

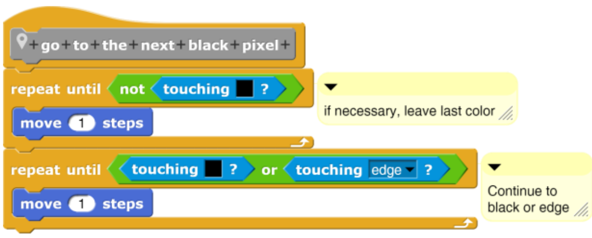
After these preparations, we must slowly start to solve the actual problem. One thing we have to teach the laser in any case: to find the next black line. To do this, we switch to the *Costumes* section and draw a small red dot there as a new costume - the laser dot. Alternatively, we can create the costume with a graphics program, save it as a *png* file and drag it to the *Costumes* area. Using the block *touching* from the Sensing palette, we can now check if our laser sprite touches the specified color. We can select this color after clicking on the color field anywhere from the *Snap!* window or from the color field that pops up. We use this block and a second one that determines whether the edge of the workspace has been reached as a termination condition of a loop from the *Control* palette, in which the laser sprite is moved one step to the right at a time.

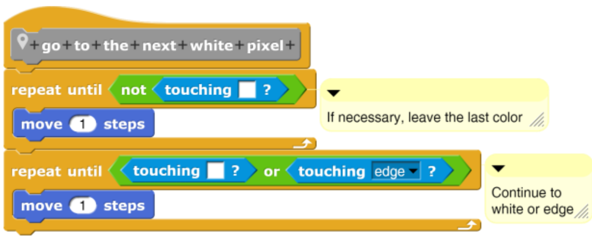


When testing this block, we notice that sometimes the laser does not move at all. When repeatedly crossing the bars, it will happen that the laser touches a white bar on the one hand, but still touches a black one on the other hand. After all, it has an expansion, albeit a small one. We therefore make sure that it first advances so far that it no longer touches any black areas. Then he runs off.



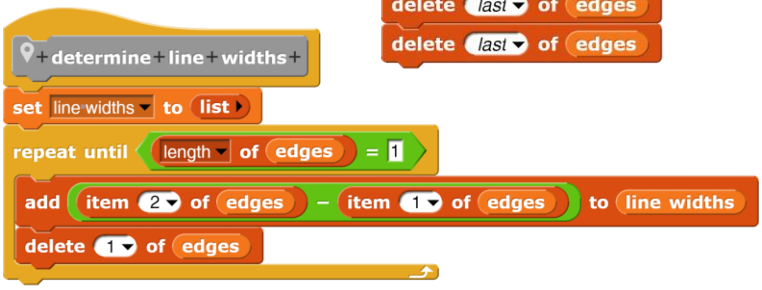
After testing this script extensively, we wrap it in its own method, a new block, called *go to the next black pixel*, which is marked local because no one else needs it. After that, we create a very similar method *go to the next white pixel*. The *comment blocks* can be found in the context menu after right-clicking on the script area.

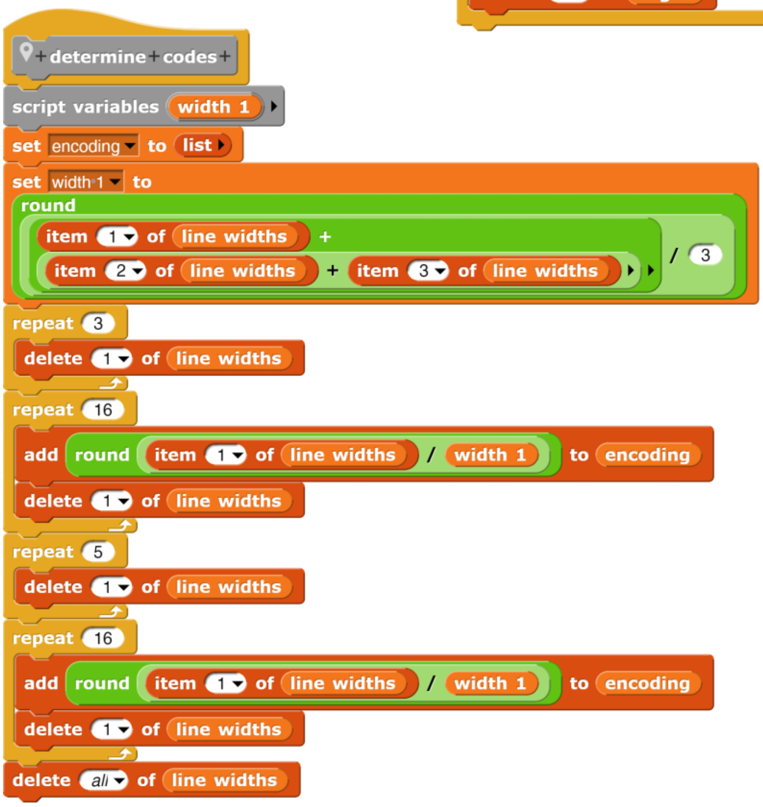




We test the interaction of these two methods in detail. After that we make sure that the variable *edges* gets an empty list as value (*set <edges> to* <list>) and that the x-position of the laser is added to this list each time a new stroke is reached (*add <x-position> to* <edges>). We delete the last two values of this list since they are generated when the right border is reached. We can observe the work of this script if we mark *edges* as visible with a small check mark. Since everything works, the script is wrapped in a new block *determine edges*.

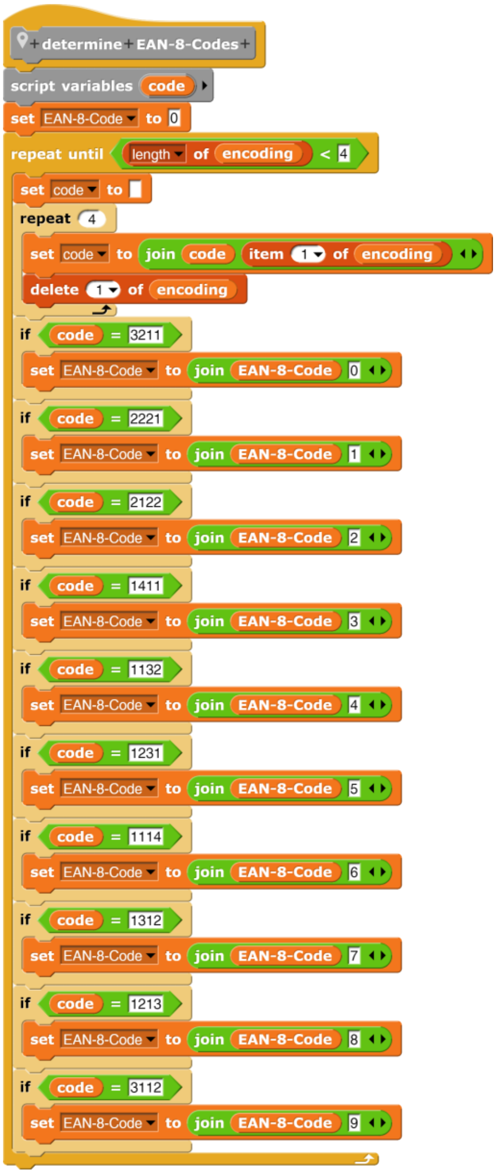
Now three very similar methods follow, in which in each case the last list just generated is traversed to determine the next values. We process the first values of each list and then delete them until we are "through".



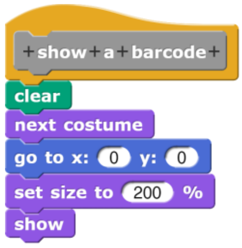


First, we calculate the widths of the sampled bars as differences of the values of the *edges* list and store them in the *line widths* list. Next, we determine the codes represented by this by averaging the width "*1*" from the first three bar widths and storing it in the script variable *width 1*, which is only known within the new block. We then delete the initial marker and calculate the first 16 stroke widths for the first four numbers. After that we delete the middle marker and proceed accordingly for the second four numbers. Finally, the rest of the list of line widths is deleted. The determined values are stored in the *encoding* list.

Now only the decoding of the numerical values in the *encoding* list is missing. We again declare a script variable *code* for the new block. We repeatedly compose this from four numerical values (using the *join* block from the *Operators* palette, which works with strings). Depending on the value of the result we get the next digit of the EAN code.



Our new blocks, which we can use like any other command block at the laser script level, can be found at the very bottom of the *Variables* palette. The small marker pin in front of the method names indicates that the methods are local to the sprite. They are not visible in other sprites.



We create the arrow with one of the generators for this on the Internet and save them as costumes of a new sprite, which we create with the arrow button above the sprite area at the bottom-right of the window. We name this sprite *Barcode*. To switch between costumes, we create a global block *show a barcode* (to also show this way of communication between objects). This enlarges the costume to twice its size and moves the sprite to the center. The block is visible for all sprites.

Our little project is to be controlled by scripts of the stage. If the green flag is clicked, then first the *Barcode* object is asked to show a new barcode - that is, to change the costume. This is done with *tell <barcode> to <show a barcode>*.

Since the block to be executed, outlined in gray, and thus marked as code, has been globally declared, we can simply drag it into the previously empty slot in the *tell* block. Then the stage sends the "*start!*" message only to the *Laser* object. Alternatively, it could have sent this message to all of them. If only the *Laser* sprite reacts, then this would have the same effect.

The last two scripts are used to initiate costume changes also by pressing the space bar and read operations by clicking on the stage.

10.2 Project: Transit Prohibited!

Level: *from middle school* Materials: *Transit prohibited*

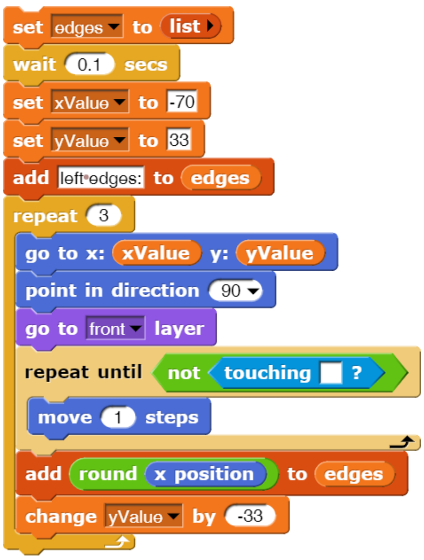
Modern cars have a camera that helps them "see" and recognize traffic signs. We want to try something like that. We'll find images of some common traffic signs and scale them all to 100 x 100 pixels using a graphics program. After that we drag them into the Costumes area of a *Snap!* sprite that we call *Traffic sign*.

As you can see, the signs are quite different. Therefore, one task will be to identify the shape of the sign. We find *round*, *rectangular* and different *triangular* signs. Fortunately, we already have a laser from the last project, which we will modify for the new task. To do this, we export the *Laser* sprite from the *Barcode* project to an XML file *Laser.xml* (right-click on the sprite, click "*export...*" from the context menu) and import this file into the new project either using the File menu or by dragging it onto the *Snap!* window. In the *Variables* palette of the laser, we delete all variables except *edges*, then we delete the local methods except *go to the next black pixel*. We open this in the block editor (right click on it), drag the blocks to the script level and then delete this method too.

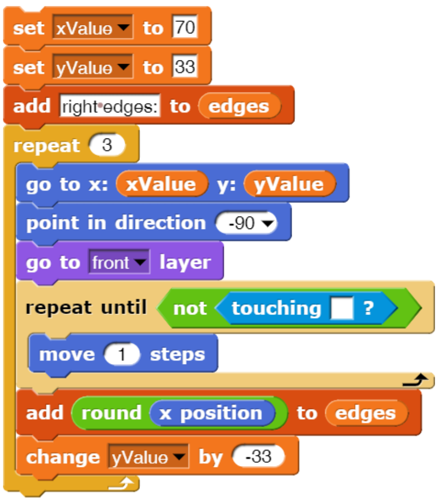
How do we now distinguish the forms of the signs?

You can come up with very different methods for this. We will try it this way: We determine the horizontal limits of the signs at three heights and then the vertical limits at three positions. Then we look at the results.

First the left edges ...

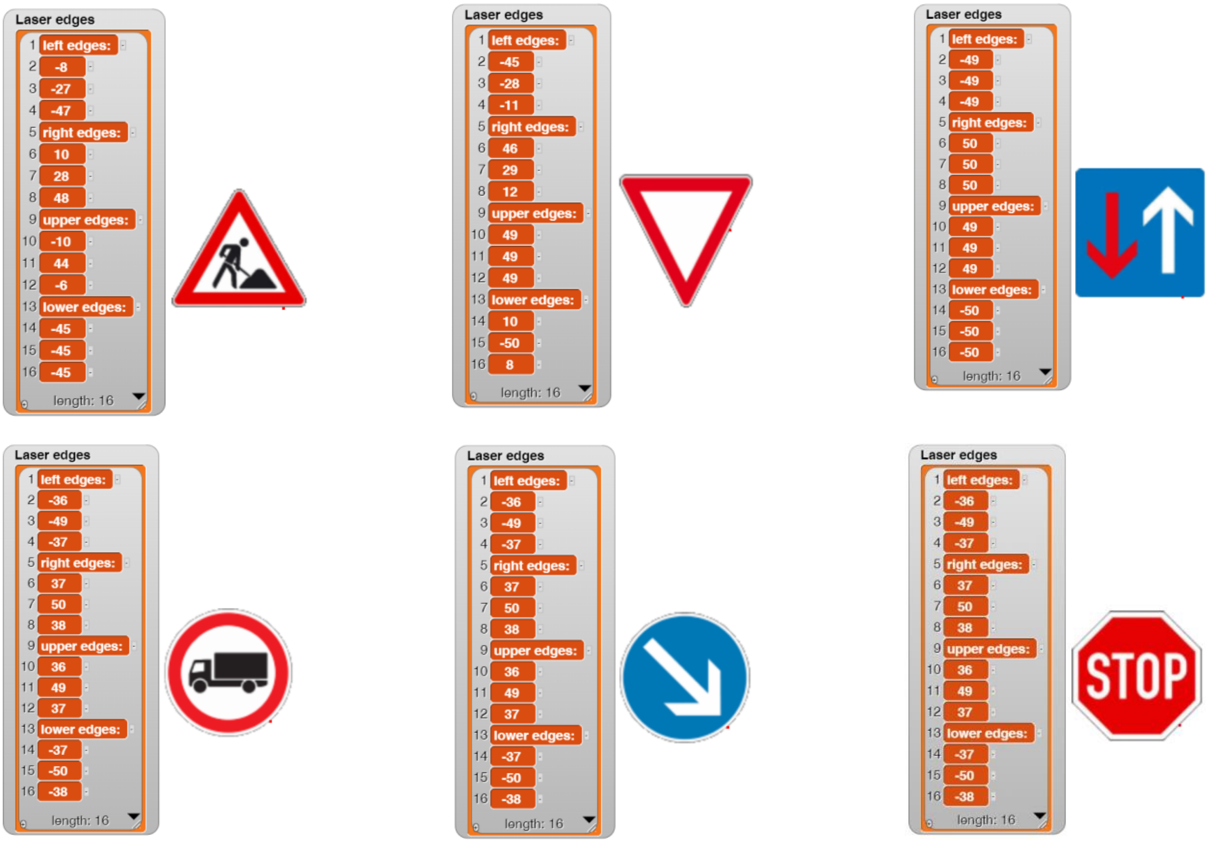


then the right ...

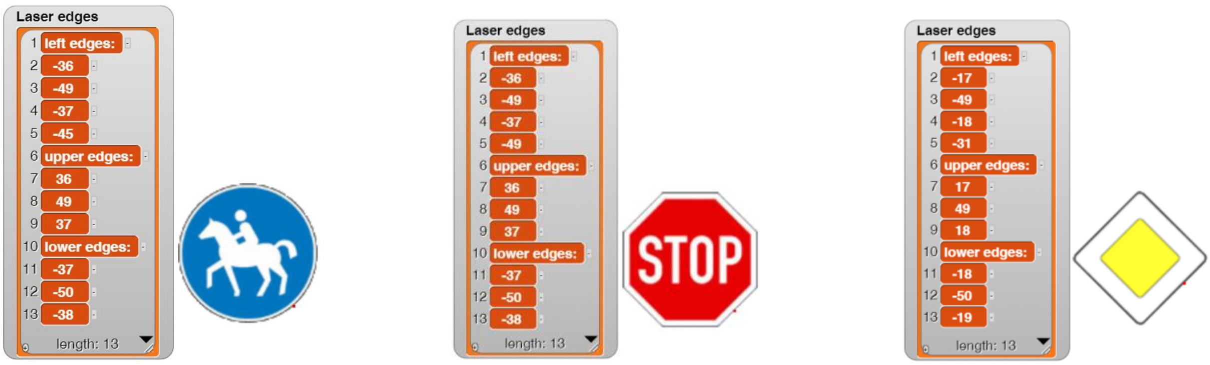


... and accordingly, the upper and lower.

We join the four scripts together and wrap them in a method *determine edges*. For example, we get the following results.



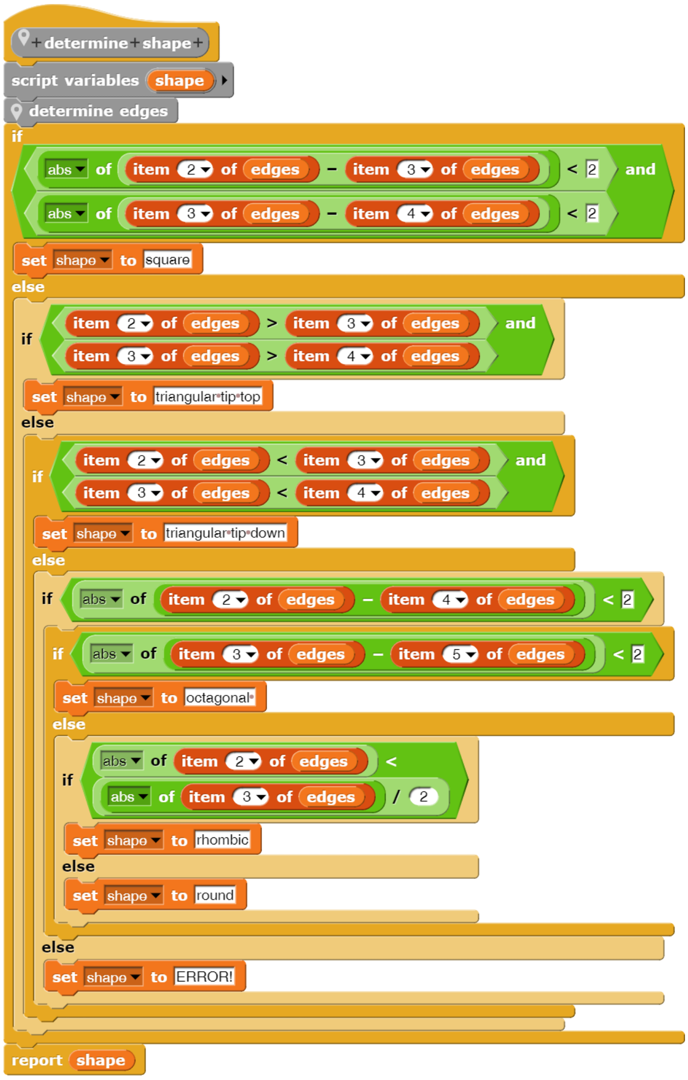
That looks good - except for the stop sign. Its edges look suspiciously like a round sign; we still have to come up with something for them. Maybe a 13th "cut" at a suitable (here: fourth, in the list: fifth) position? For this we can omit the right edges because the signs are obviously symmetrical. If we do that, then we get for the "round" candidates:



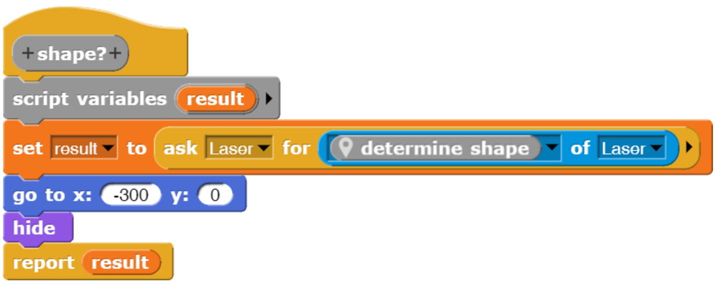
The 5th list entry contains the value for height 19 in each case - and thus a measurable difference.

To evaluate our results, we write a *determine shape* block. This is supposed to be a reporter block that determines and returns a value - the shape.

For rectangular signs the entries 2, 3 and the 4th entry should be about the same, for the triangular signs the values grow or fall. If we assume something round (the second and fourth entries should be about the same size), then it is the octagon of the stop sign, if the third and the fifth entries are about the same. And the rhombus of the right-of-way sign, if the second entry is rather small. Otherwise, really around a round sign. And errors can occur of course.

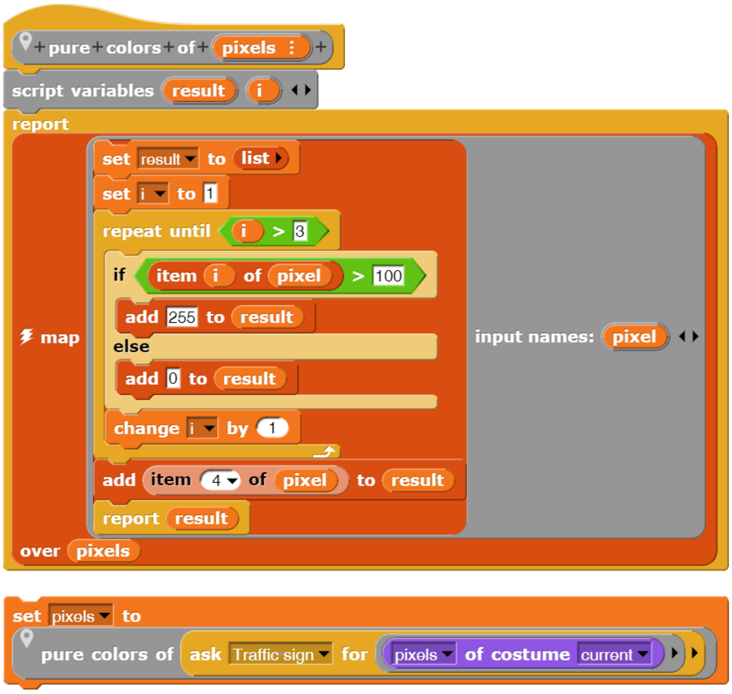


With this we have already quite limited the number of possibilities, and we see that - so far at least - we get by with the results for the left edge. We write a local method *shape?*, which determines the shape of the currently displayed sign. In addition, the laser is sent "into the heath" and hidden so that it does not interfere further. Its work is done.



For the further meanings of the signs, the colors on the edge and inside are important. For the final determination of the type of traffic sign, let's just count the number of different colored pixels in the sign. Maybe that will be enough. We leave this work to a new object called *Color counter*. This needs the pixel list of the current costume of the *Traffic sign* object. We politely ask this for the required data, which we store in a local variable *pixels*. There we have a list of the three color values and the transparency of the pixels of the current costume. Since this has the dimensions *100 x 100*, we get *10,000* values.

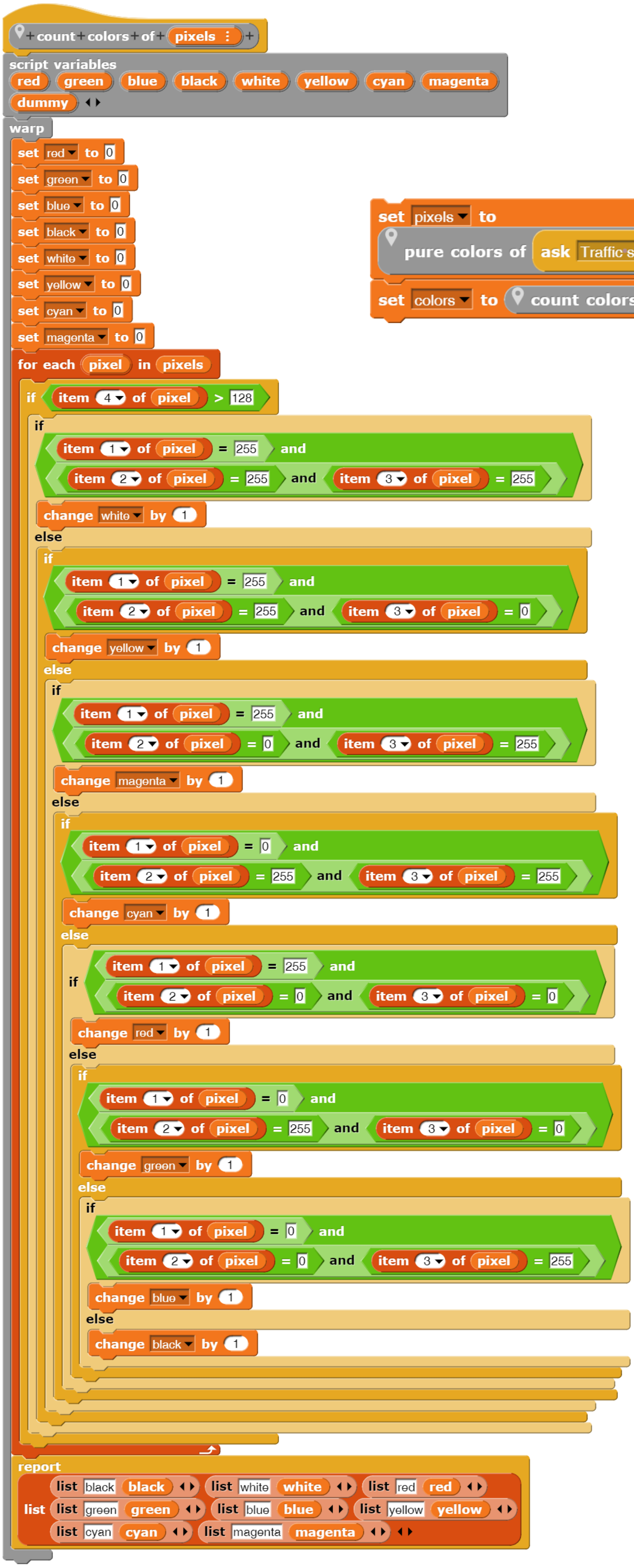
In this list, the pixels outside the actual sign have the transparency 0, those inside the value 255. The three RGB values in front of them do not represent "pure" colors, but mixed values that are "predominantly" red, for example. We change this by a method *pure colors of ...*, which sets color values above *100* to *255*, the others to *0*. This works very fast in the compiled version of the *map-over* block even with *10,000* values.



10000	A	B	C	D
841	237	28	36	255
842	237	28	36	255
843	237	28	36	255
844	237	28	36	255
845	237	28	36	255
846	237	28	36	255
847	237	28	36	255
848	237	28	36	255
849	237	28	36	255
850	237	28	36	255
851	237	28	36	255
852	237	28	36	255
853	237	28	36	255
854	237	28	36	255
855	237	28	36	255
856	237	28	36	255
857	237	28	36	255
858	237	28	36	255
859	237	28	36	255
860	237	28	36	255
861	237	28	36	255

10000	A	B	C	D
841	255	0	0	255
842	255	0	0	255
843	255	0	0	255
844	255	0	0	255
845	255	0	0	255
846	255	0	0	255
847	255	0	0	255
848	255	0	0	255
849	255	0	0	255
850	255	0	0	255
851	255	0	0	255
852	255	0	0	255
853	255	0	0	255
854	255	0	0	255
855	255	0	0	255
856	255	0	0	255
857	255	0	0	255
858	255	0	0	255
859	255	0	0	255
860	255	0	0	255
861	255	0	0	255

Similarly, let's count the "pure" colors in the image: We introduce a separate script variable for each, which we initially set to zero. Then we look at all pixels of the sign that have a sufficiently large transparency. For these we analyze the RGB values and increase the value of the correct variable. Finally, we return a list with the results, in which we insert the color labels so that we don't get confused.



Color counter colors

8	A	B
1	black	0
2	white	1544
3	red	6284
4	green	0
5	blue	0
6	yellow	0
7	cyan	0
8	magenta	12

Color counter colors

8	A	B
1	black	81
2	white	3052
3	red	2519
4	green	0
5	blue	0
6	yellow	0
7	cyan	0
8	magenta	15

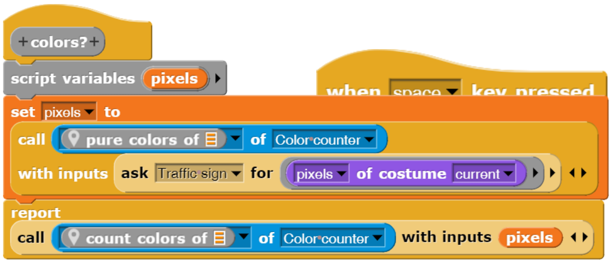
Color counter colors

8	A	B
1	black	121
2	white	2249
3	red	0
4	green	0
5	blue	0
6	yellow	0
7	cyan	5482
8	magenta	0

Color counter colors

8	A	B
1	black	0
2	white	1027
3	red	994
4	green	0
5	blue	119
6	yellow	0
7	cyan	7832
8	magenta	16

For easy use of the methods, we write again a global method *colors?* which initiates the corresponding operations.



We leave the control of the objects to the stage. When the space bar is pressed, the traffic sign should change and when the green flag is clicked, the analysis takes place. The stage object queries the results of the others and evaluates their data.

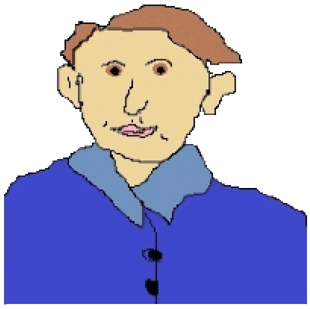
For evaluation, we use the determined shape on the one hand and the counted color values on the other. In simple form, this can be done as described opposite.

The results are as desired.

10.3 Project: Face Recognition

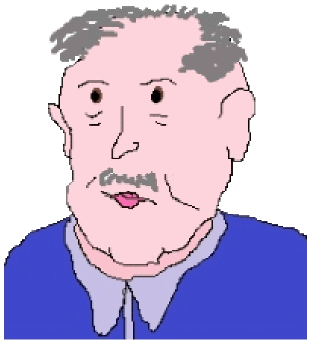
Level: *high school* Materials: *Face recognition*

In order to discuss the social consequences of informatics systems, face recognition is a good topic. Therefore, we want to use the already known features of *Snap!* for this purpose.



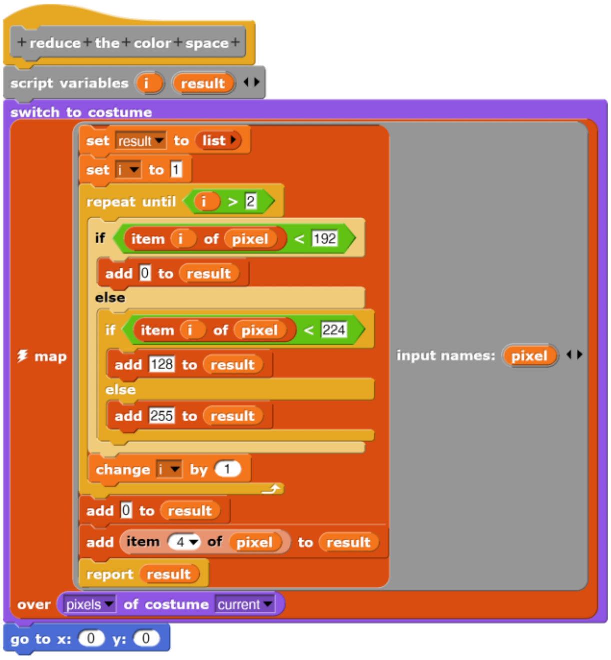
Peter

Passport photos are strongly standardized for good reasons: the facial posture is prescribed, ears must be visible, ... This makes face recognition much easier. We therefore draw four faces that roughly correspond to these regulations. On these "photos" we then apply the already known procedures.



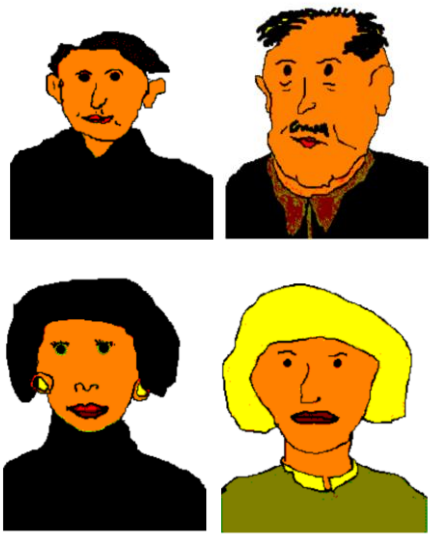
Paul

We are looking for the face, which in these four cases is roughly "pink". Since the face colors are nevertheless different, we first perform a color space reduction. We find suitable limits of the (here) three intervals by trial and error.



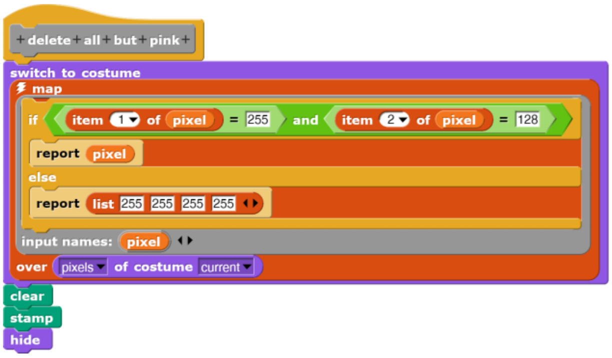
The process is from the traffic sign recognition in the previous section. The faces now stand out very nicely in orange - regardless of how they looked before.

Mary



Hannah

When we erase all the colors except orange, only the faces remain. We stamp the result on the stage and let the passport photo disappear. It has done its duty.



In these faces we now must identify the eyes, the mouth, the nose, etc. From the ratios of the sizes *eye distance to nose length*, *mouth width to face height*, ... we can then identify the person.

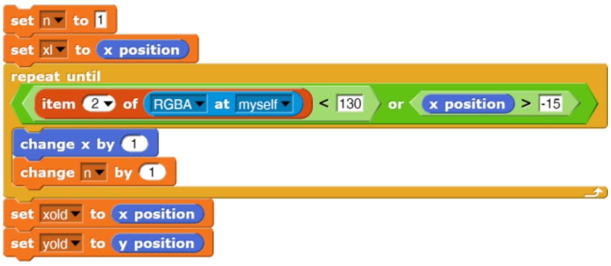
How to find eyes?

They represent "holes" in the face, which should not be too large and not too small. For example, the right eye (from the person's point of view) should be at the top-left of the passport image. To do this, we first need to be able to access individual pixels in the image. For this purpose, we use our *Laser* sprite, which directly provides us with the color value at its position.

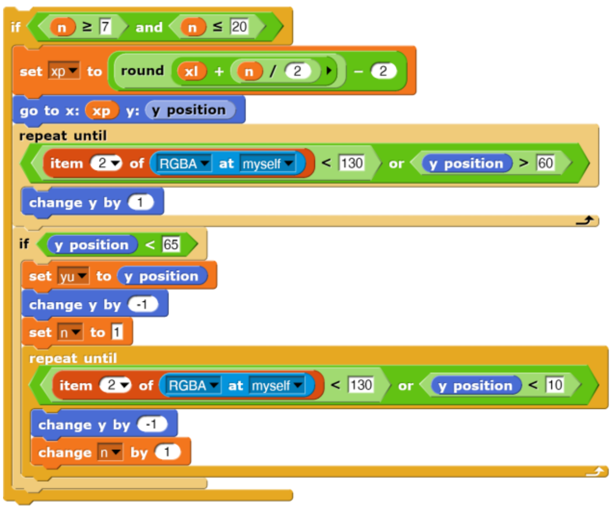
With this, we search the upper-left area of the image for a "hole". We examine the range *-50 < x < -15, 10 < y < 60*. We find the values by trial and error. For the comparisons, we take the green value, and we examine the range line by line.

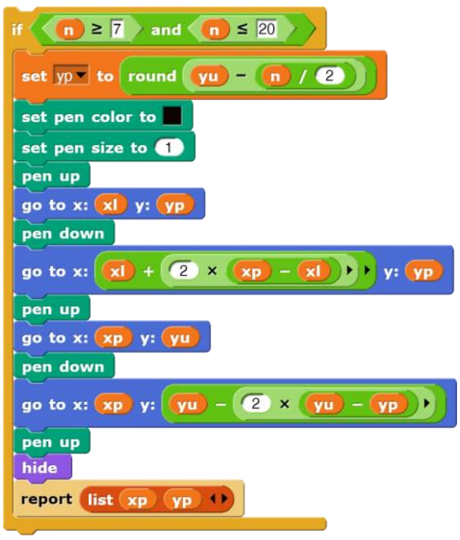
We pass over a possibly existing orange area and stop at the first white pixel.

Then we note the left x-value, count the following white pixels to the right and note the endpoint.

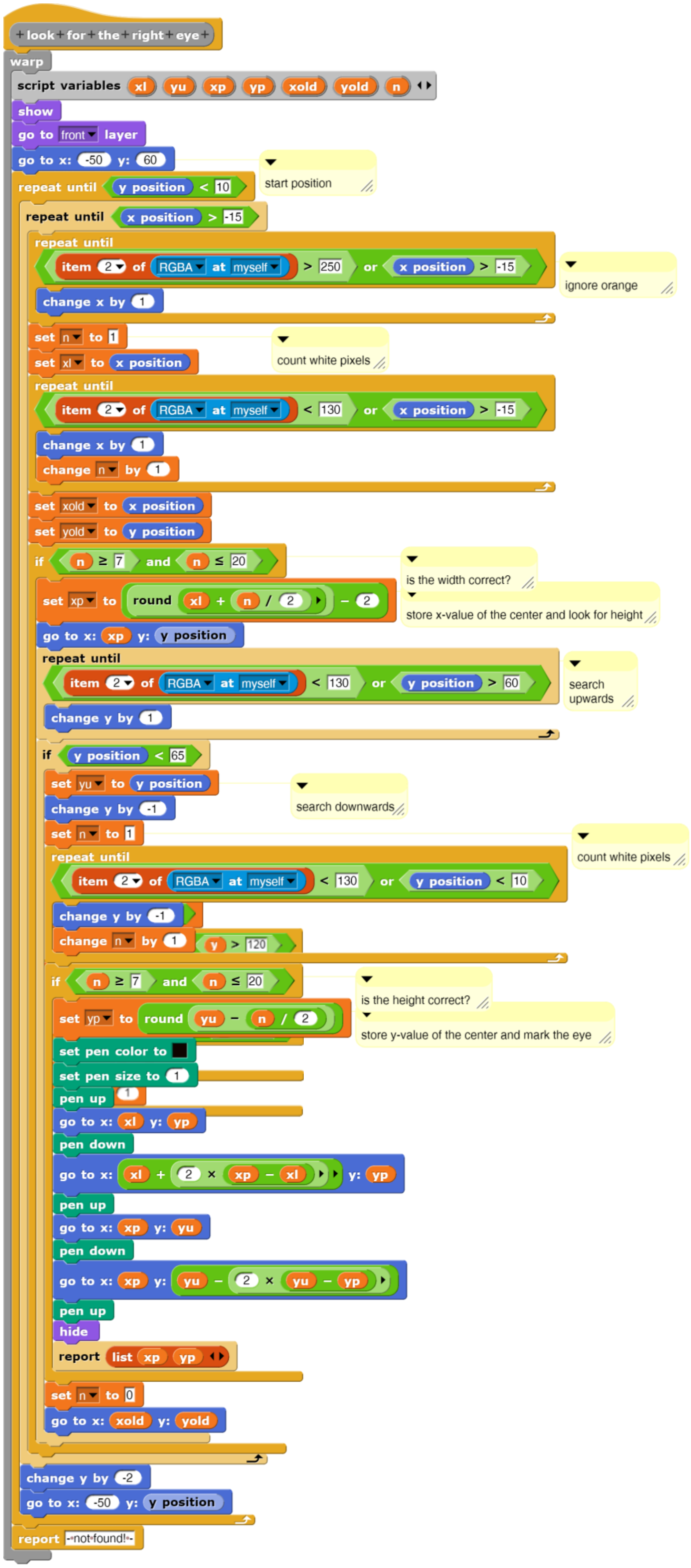


If the width corresponds to an eye, we determine the center and measure the number of white pixels in the vertical there.





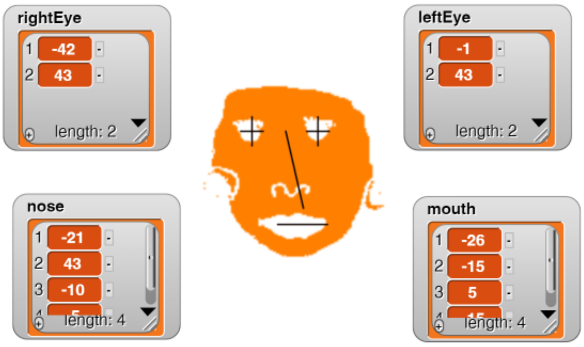
If the result is also "correct", we let draw a cross at the position and return the center point.

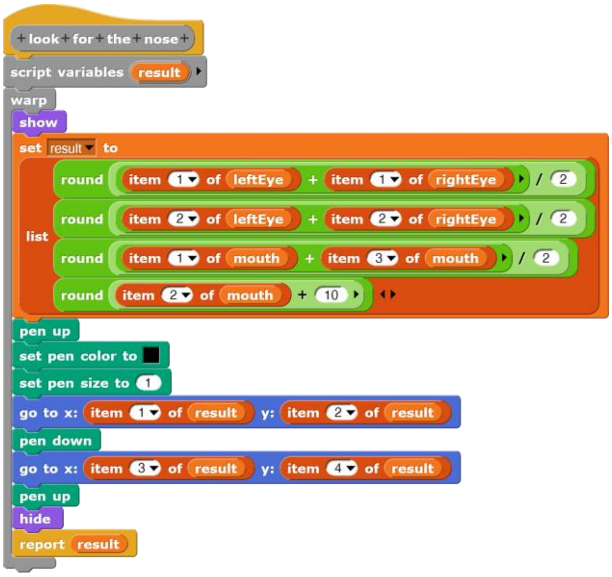


If we still haven't found an eye, then we first continue searching to the right. If there was nothing there either, then we repeat everything in the next lines.

The procedure in full is shown opposite.

We find the left eye using the same procedure, and for the mouth we determine the two corners of the mouth. The nose we simply draw between the eyes and mouth.





From the determined values we calculate some ratios and store them together with the names in a list *allAttributes*. By comparison with the currently determined values, the searched person can be easily identified.

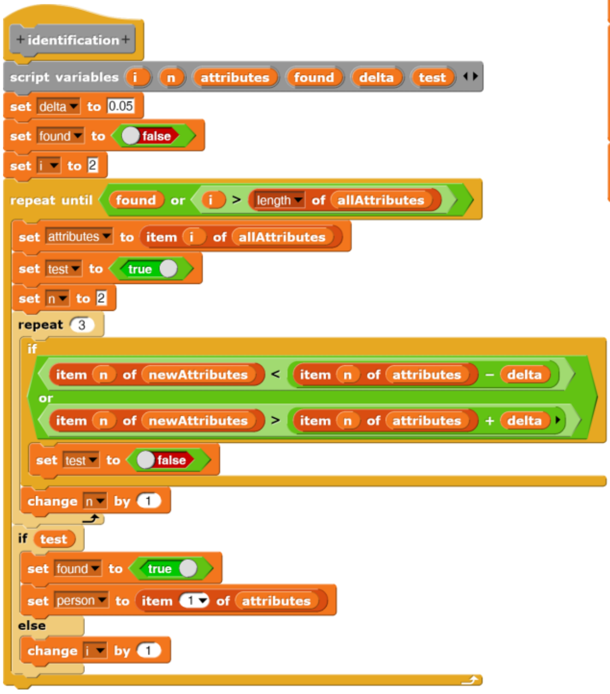
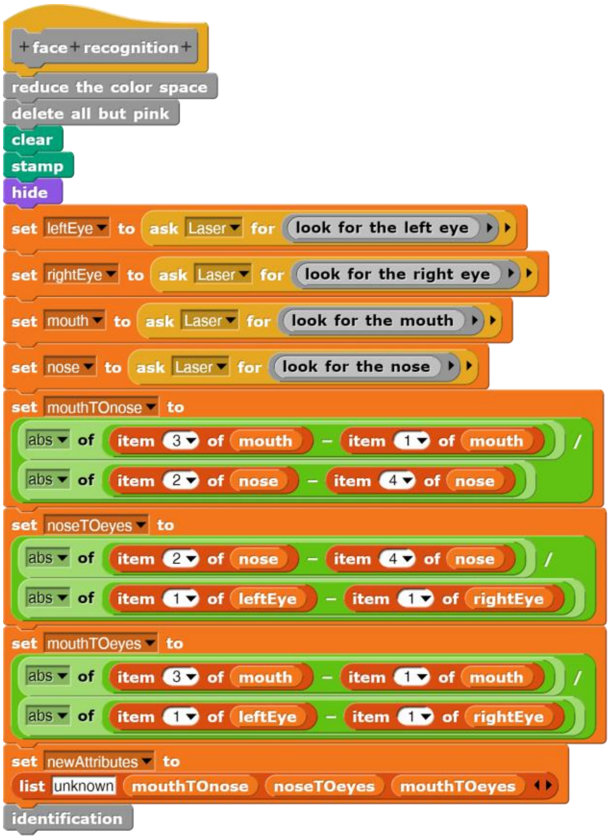


Table view

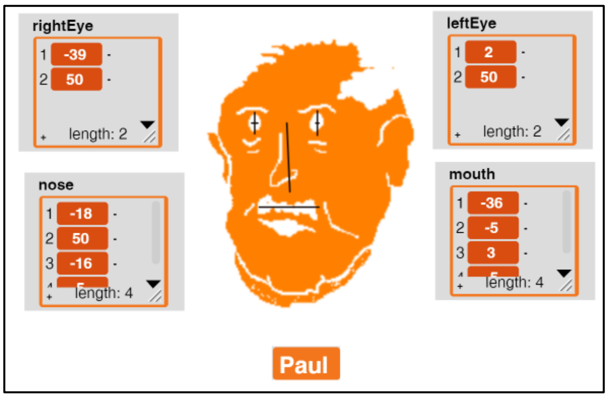
5	A	2	C	D
1	Name	Mouth : Nose	Nose : Eye	Mouth : Eye
2	Mary	0.490	1.114	0.545
3	Hannah	0.674	0.896	0.604
4	Peter	1.034	0.8132	0.844
5	Paul	0.465	1.049	0.488

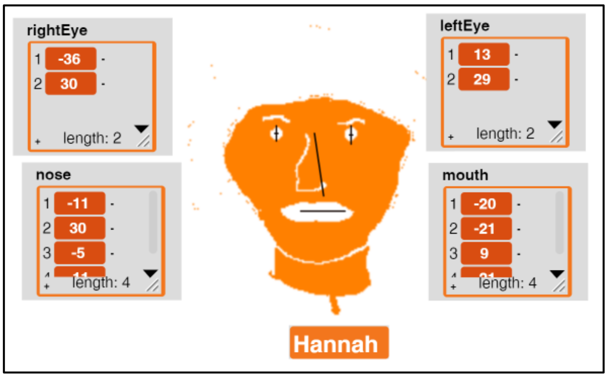
OK

The overall problem can be solved by composing the subproblems. We assume that the image of the person to be identified is on the screen. This is transformed, stamped on the stage and the changes are displayed.



The four people are safely recognized.





10.4 Tasks

1. a: Find out about the calculation of the **check digit** in the EAN-8 code. Use some examples to test whether you have understood the procedure.

   b: Have the barcode scanner check after each reading process whether the check digit has the correct value.

   c: Extend the barcode scanner with more capabilities: Codes can also be read "backwards" and there are also longer codes, e.g. EAN-13.

   d: Extract the manufacturer and product number from the barcodes read. Using appropriate data, indicate the results in plain text: *"Honey from the bee house", ...*

2. Develop a **barcode generator**. It is given a sequence of numbers. From this, it calculates the check digit and prints the barcode. This can be done, for example, with the help of appropriate costumes that are printed on the stage at the right places with the *stamp* block from the *Pen* palette.

3. Have **foreign traffic signs** identified. Use the signs to determine where a picture was taken.

4. A **speed warning system** in a car is designed to determine whether the speed limit has been exceeded based on changing traffic signs.

5. German **license plates** contain a character set that is very suitable for image recognition (uniform character width, ...). Develop a method that recognizes license plates. Discuss the consequences.

6. **Facial recognition** can be found today when logging into a computer system, in cameras and smartphones, in social networks, ... Learn about other applications and discuss their results.

7. In some states, a system of **social credits** is being introduced or its introduction is being discussed. Find out about the system and discuss the consequences in connection with extensive video surveillance.

11 Sounds

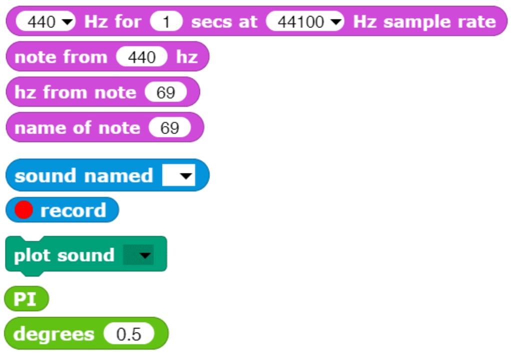
Similar to animated graphics, it is somewhat difficult to only describe the handling of sounds. Therefore, just the different possibilities are presented here - with the urgent recommendation to test and experiment with the "code snippets", too.

11.1 Find sounds

First, we need a sound in *WAV* format. For this we can either import it via the file menu (*File → Sounds...*) ...

... or drag it "from outside" into the *Snap!* window as usual ...

... or simply record it yourself. For short recordings, this can be done directly using the *Snap!* sound recorder on the *Sounds* page. For longer recordings you should use one of the common tools.



For further processing we load the library *Audio Comp* from the file menu. This provides us with the following blocks from the *Sound*, *Sensing*, *Pen*, and *Operators* palettes.

In the following we work with the file *sound check.wav*, which we created in one of the described ways.

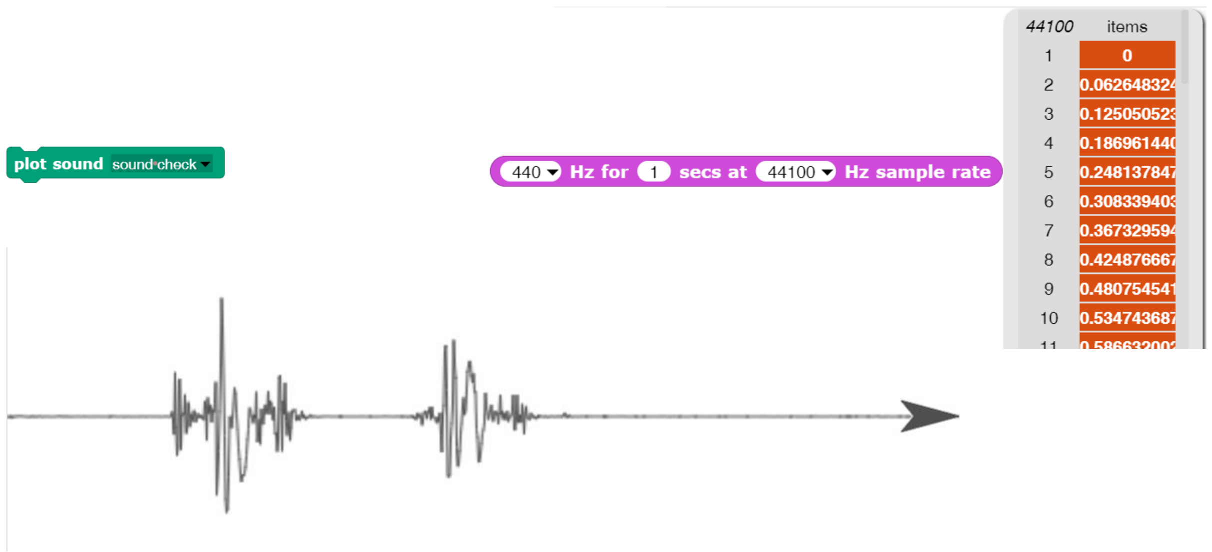
11.2 Process Sounds

If a sound is available on the *Sounds* page, it can be displayed in the corresponding blocks. The easiest way to try this is in the block for playing sounds.

For further processing we need a representative of our sound. This is what the *sound named <soundname>* block is for. If you edit it, you have found a small example for the use of the sound blocks.[60]

The *of* block for sounds provides access to further properties of sounds. In particular, its *samples*[61] can be determined as a list. These are needed if a sound is to be edited actively. For example, we can influence the playback speed of the sound by changing the sample rate. The *Hz for ...*-block creates samples with the properties to be specified, e.g. "pure sounds".

The visualization of the sounds is interesting. Using the *plot <sound>* block we get a graphic of the sample on the stage.

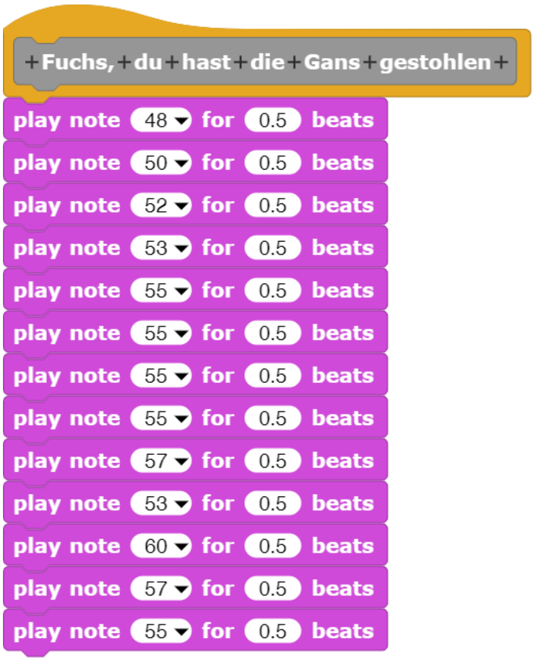


[60] The same applies to (almost) all other sound blocks. If you edit them, you will find examples e.g. for the use of JavaScript.
[61] https://de.wikipedia.org/wiki/Abtastrate

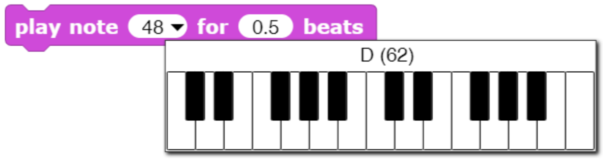
11.3 Make Music with Jens Mönig[62]

Level: *high school* Materials: *Music*

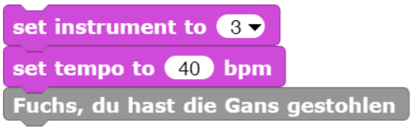
A sample consists of a list of numbers, stereo sounds of a two-element list of samples (see above). Consequently, sounds can be manipulated with the usual list operations, e.g. invert, change value, …



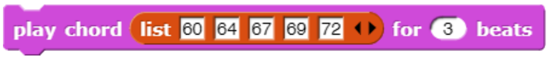
But songs can also be composed of notes, even quite comfortably. The selection of the note is done on a keyboard (piano keyboard), which you get when you click on the *play note...for...beats* block the little arrow down for the drop-down menu. From this you can quickly compose songs …



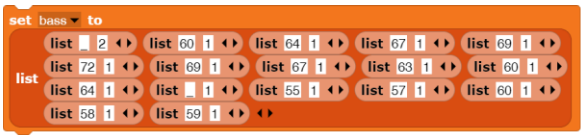
… and play them on different instruments and at different speeds.



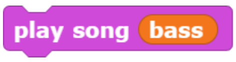
If you play several notes in parallel, chords are created …

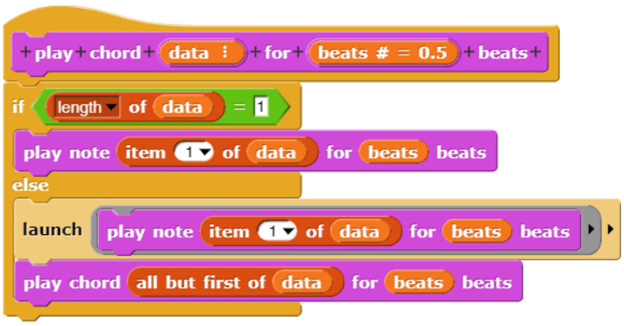


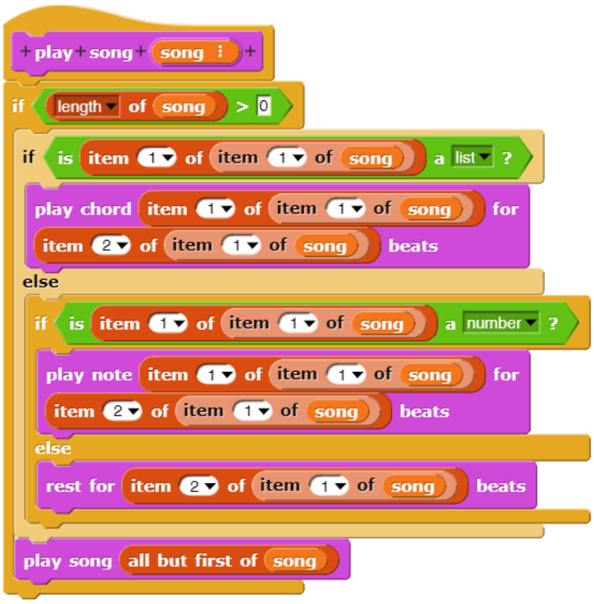
… and from these songs, using suitable list of pairs of (note, duration), …



… which can be played and varied.







62 Following the example of "music" by Jens Mönig.

specify two basic chords

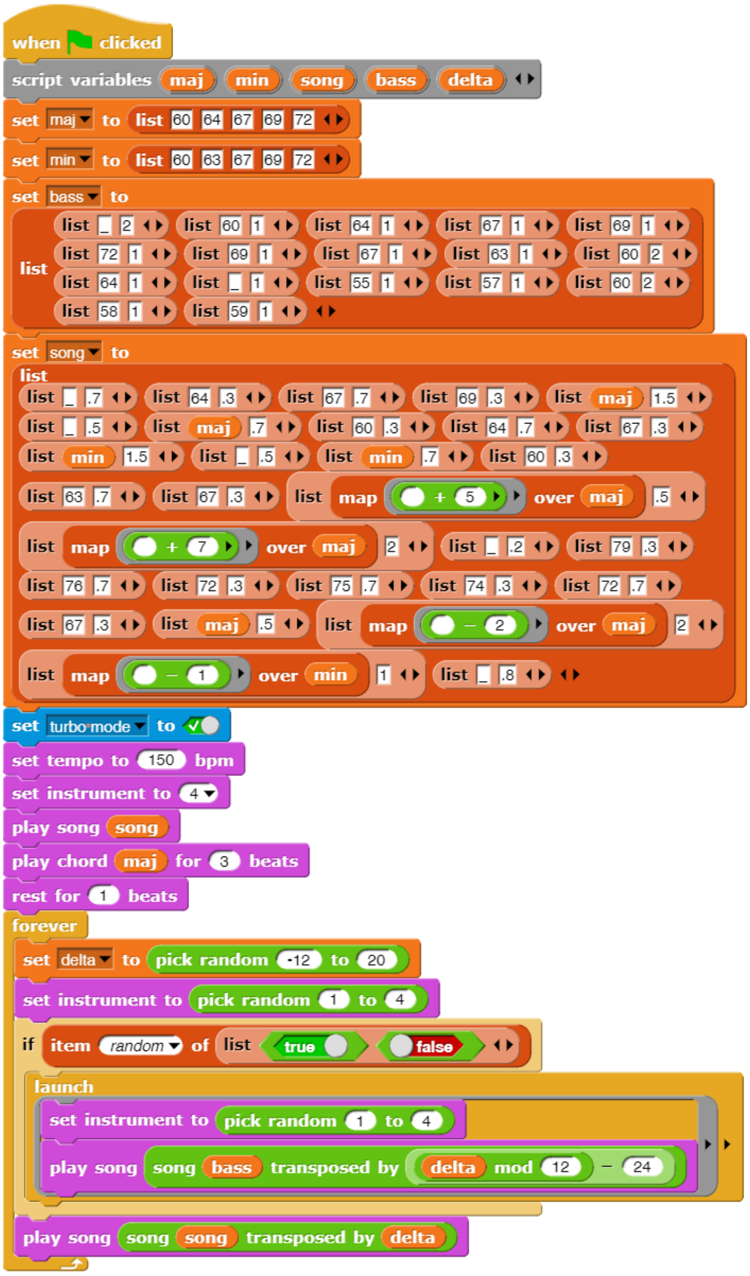
describe bass accompaniment and song using lists of tone/duration pairs

make a few presets

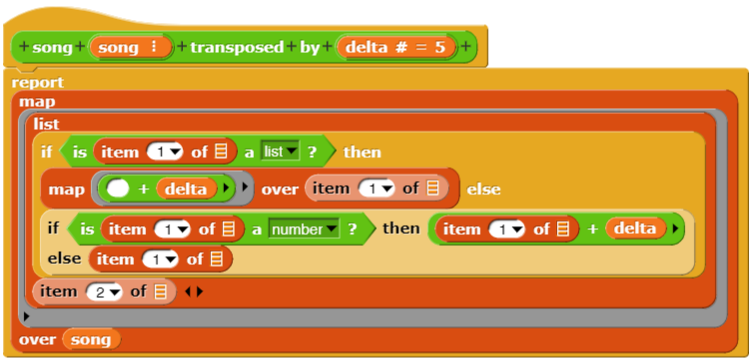
play the song, finish with chord and short break

and now with variations the song and the bass accompaniment play again and again

both play in parallel because of the *launch* block

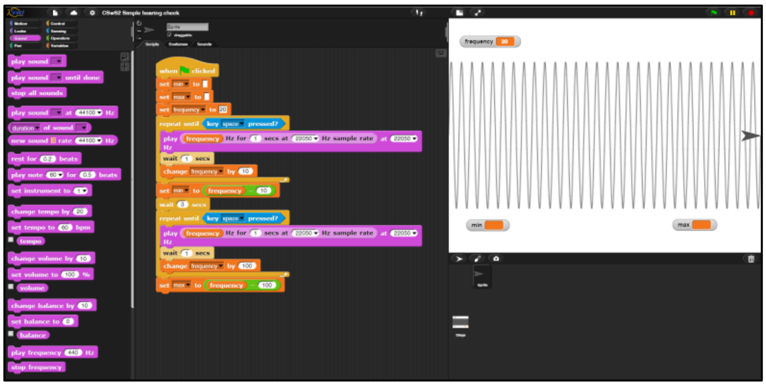


transpose a song



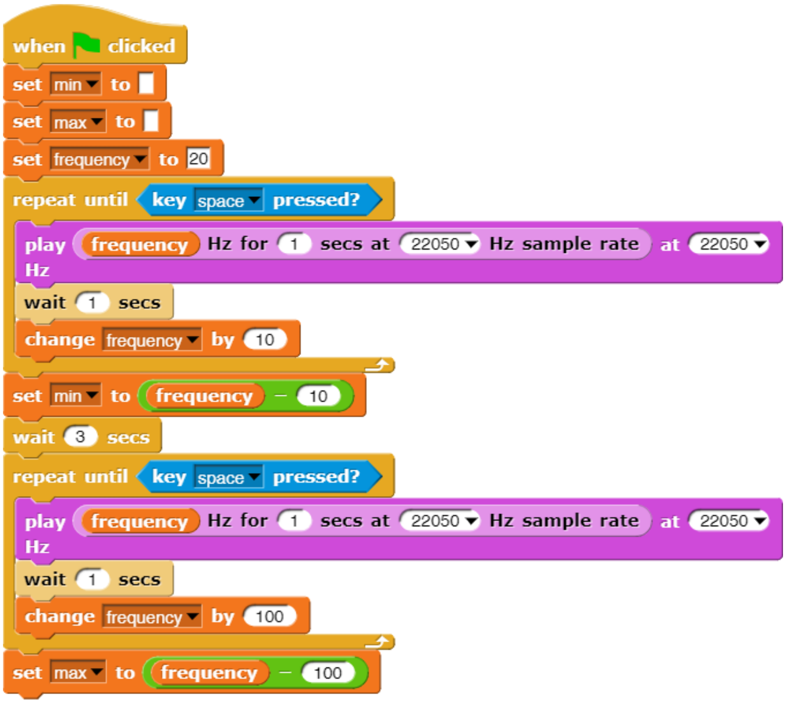
11.4 Project: Hearing Test

Level: *from middle school* Materials: *Hearing volume test*



In a hearing test, the hearing ability is tested at different frequencies or different volumes. In this case, we play tones of increasing frequency until the test person hears something. Then he (or she) presses the space bar. This frequency *min* is noted. Then the frequency is increased until nothing is heard. This frequency is also stored. In the current *Snap!* version, the *Java-Script extensions* in the Settings menu must be enabled for this.

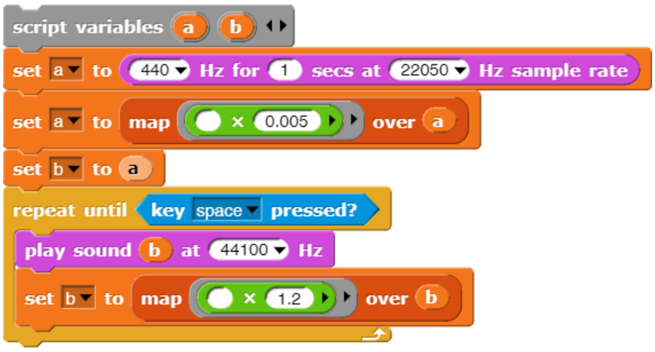
Make sure that the volume cannot become too high!



11.5 Tasks

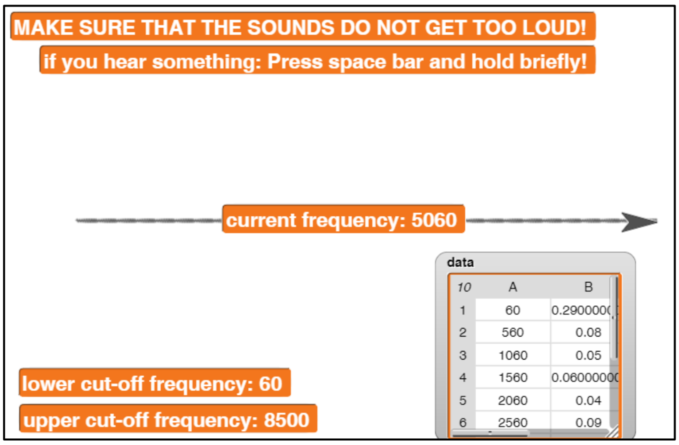
1. Establish experimental conditions in the hearing test that lead to **comparable results**.

2. Change not only the frequency, but also the volume. Since our sounds are described by samples, the **volume** can be changed by simply multiplying the sample values. (*This can be done as in the script or by applying the multiplication block as a hyperblock.*) For example, in the following script, the volume is increased until the spacebar is pressed.



Make sure that the volume cannot become too high!

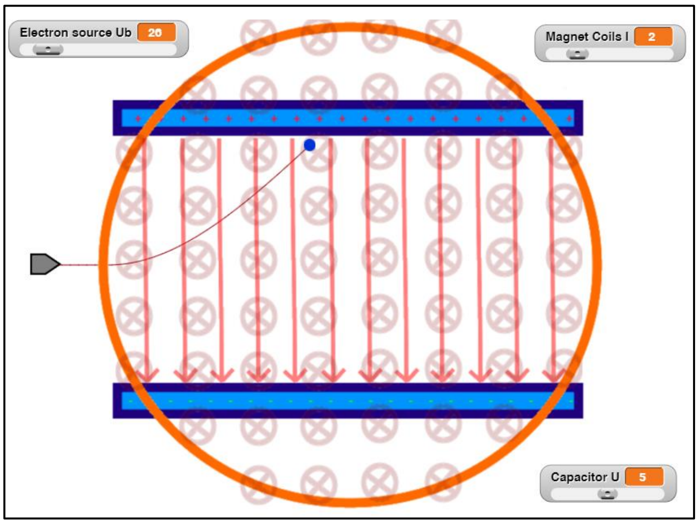
3. Measure the **cutoff frequencies** and the volume required per frequency to hear. Create a diagram from this.



4. Take a trip to an **ENT practice/clinic**. Present your diagrams and have them explained to you whether and what can be read from them. Find out about causes of possible hearing loss.

12 Project: Electrons in Fields

Level: *high school* Materials: *Electrons in fields*

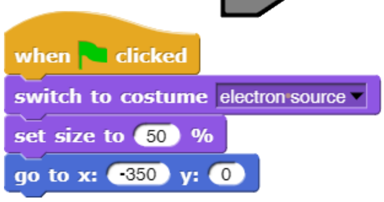


We want to use the knowledge we have acquired so far to realize a small project from the field of - well - physics: Electrons move in a tube in which a capacitor is built in. This tube is brought inside a pair of Helmholtz coils in such a way that the electric and magnetic fields are orthogonal to each other. Both are semi-homogeneous. This is one of the standard high school experiments. All components can be developed independently in different groups, and in very different ways. Only the physics remains the same. That's the way it is with physics. 😉

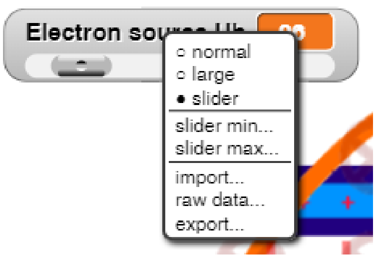
12.1 The Electron Source and the Experimental Setup

Since this is a standard experiment, the required equipment should be available in the physics collection. It is therefore a good idea to set up the experiment properly, photograph it and extract the partial devices from the pictures so that they can be used in the project. Here in the script, only simple drawings were made instead. We need pictures of the capacitor, the coils, the electron source and - for illustration - the generated fields.

First of all, let's enlarge the *Snap!* stage to 800 x 600 pixels. There is a menu item for this in the settings menu of *Snap!* Then we draw a simple picture of an electron source and import it as a costume of the current sprite.



After starting the program with the green flag, our electron source is sent to its place in the correct outfit. If necessary, we can also move it to another place in the experiment. The device has only one characteristic property: the momentary acceleration voltage of the emitted electrons. For this a local variable *Ub* is generated and displayed on the stage. In the context menu of this display *slider* can be selected and the minimum and maximum value can be set. The slider is now used to change the variable value between these numbers while the program is running. We choose a range between *0* and *250* (volts).

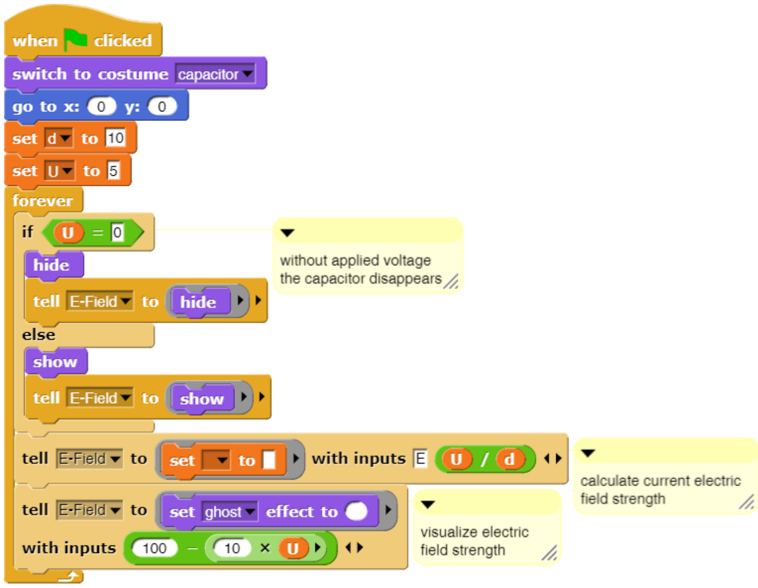


12.2 The Capacitor and the Electric Field

The capacitor in the tube has a plate spacing *d*, which we set firmly so that later a useful electron movement results. After it has also found its place, it runs continuously until the program aborts. If we set the applied voltage *U* to zero, it should disappear, so that we can also study movements only in the magnetic field - there it would only disturb. For *U* and *d* we set up local variables. After that, it tells the electric field *E-Field* its current value. This is done by setting in the context of the *E-field* the value of its local variable *E* with the value *U/d*.

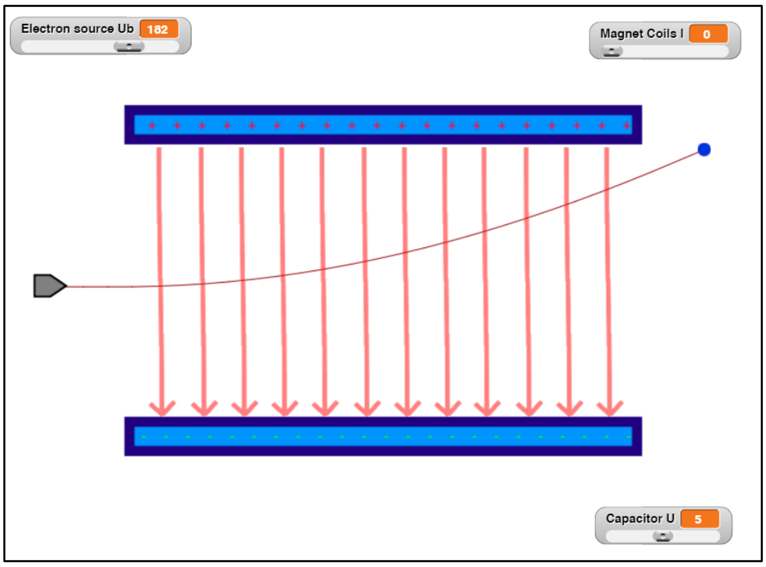
Indeed, it is true: $E = \dfrac{U}{d}$

It is important that the slots in *set <variable> to <value>* are blank, so that they can be replaced by the specified values!

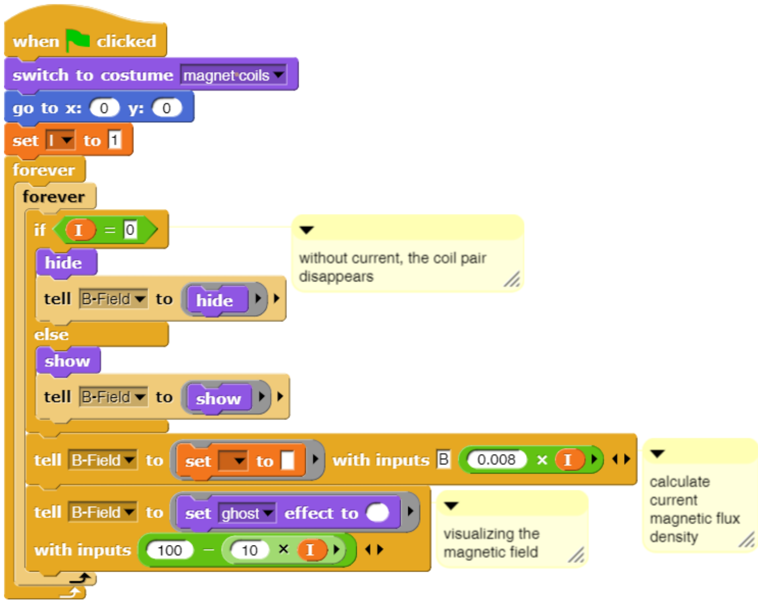


Then, in the same way, he sets the *ghost effect* of the electric field, i.e. its transparency, to a value that depends on the applied voltage. The smaller the voltage, the more transparent the arrows symbolizing the electric field appear.

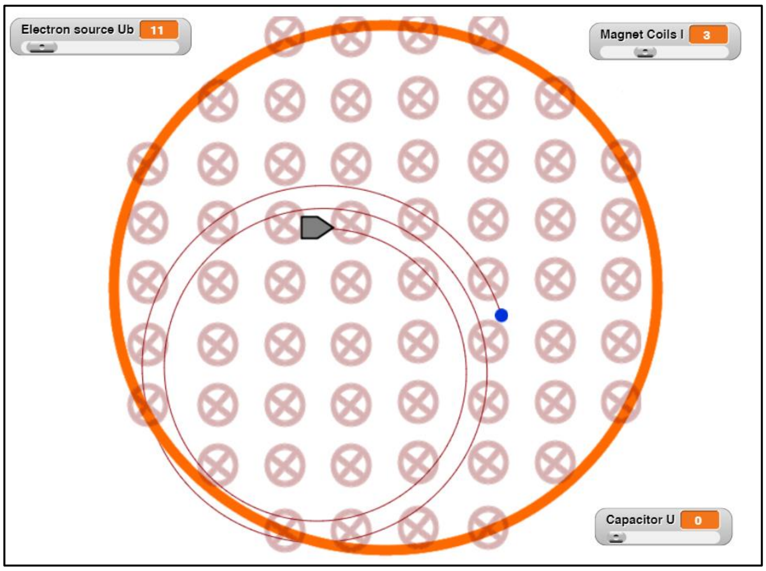
The electric field, another sprite of its own, simply consists of a costume containing a series of parallel arrows that fit between the capacitor plates. It has a local variable *E*, which is set by the capacitor - as described. We display the voltage of the capacitor as a slider variable on the stage.



12.3 The Helmholtz Coils and the Magnetic Field



The Helmholtz coil pair is symbolized by a simple circle on the stage.[63] It contains a local variable *B*, the magnetic flux density, which turns out to be $B = 0.008\frac{T}{A} \cdot I$ for commercially available devices, where *I* is the electric current through the coils. We display it as a slider variable between *0* and *10* (amps). That's pretty powerful. The coils, much like the capacitor, tell the magnetic field what value and transparency it has. Like the electric field, the magnetic field consists only of a picture.



If we switch off the electric field and consider only the electron orbit in the magnetic field, we get an approximately circular orbit, but not a closed one. The spiral results from calculation inaccuracies because the calculated changes are too large. We would have to proceed much more small-step. So, this would still have to be worked on!

[63] You can really make this much more beautiful!

12.4 The Electrons

Now comes the bitter moment when we can no longer avoid physics. So be it. 😉

Two forces act on an electron in the arrangement: the electric and the magnetic. With the electrical one it is quite simple. It acts upwards here because the electron is charged negative: $F_{e,y} = e \cdot E$

The Lorenz force $\vec{F}_L = q \cdot \vec{v} \times \vec{B}$ is orthogonal to the current velocity of the electron and to the field direction. So, we have to work with vectors. The magnetic field has only one component in z-direction, i.e. "into the screen", the velocity has two components in x and y-direction "on the screen".

So, it is valid: $\vec{F}_L = e \cdot \begin{pmatrix} v_x \\ v_y \\ 0 \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ B \end{pmatrix} = e \cdot \begin{pmatrix} v_y \cdot B \\ -v_x \cdot B \\ 0 \end{pmatrix}$

In summary: $\vec{F}_{total} = e \cdot \begin{pmatrix} v_y \cdot B \\ E - v_x \cdot B \\ 0 \end{pmatrix}$, and because is true: $\vec{F} = m \cdot \vec{a}$

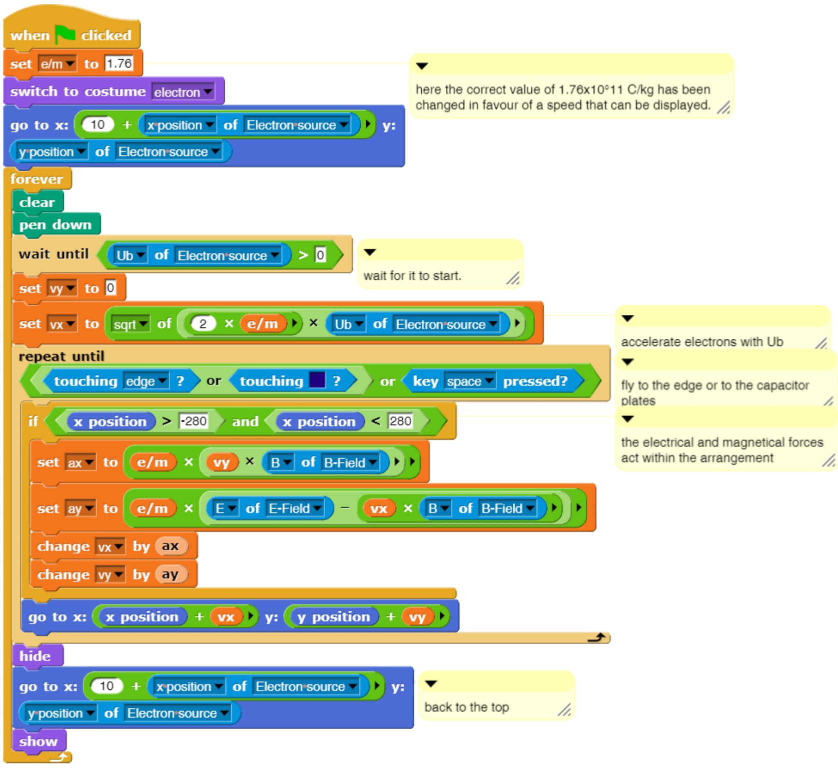
we obtain for the accelerations in the two directions:

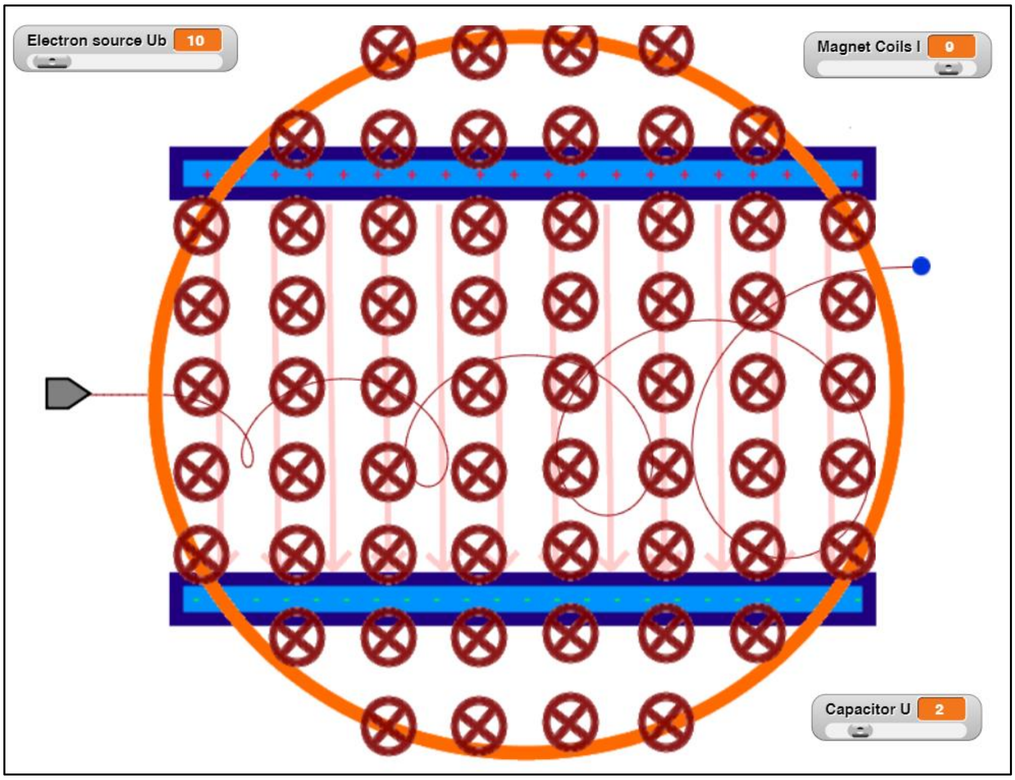
$a_x = \frac{e}{m} \cdot v_y \cdot B$ und $a_y = \frac{e}{m} \cdot (E - v_x \cdot B)$

with the appropriate signs to the coordinate directions of *Snap!.* These accelerations change the velocity components and these in turn change the position of the electron. That's it.

We can transfer these results directly into the script of the electron. We adjust the natural constant *e/m* a little bit for this, because "real" electrons are significantly faster than our screen representatives. Other adjustments are not necessary. So, the electron needs only the "too large" chosen local variable *e/m* and the acceleration and velocity components. To make it easier to follow the trajectory, it is drawn on the stage.

One can observe the sometimes astonishing movements of the particles now nicely. Of course, we must ask what is true and what is due to numerical effects. Projects never end, they give impulses for further questions!





13 Texts and Related Topics

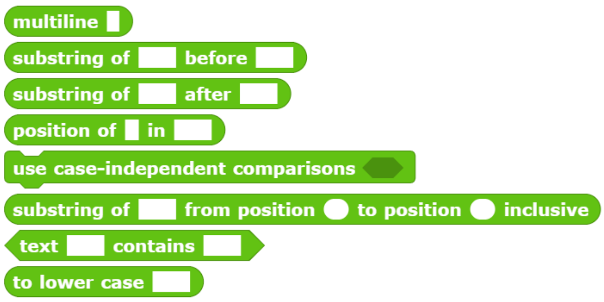
13.1 Operations on Strings

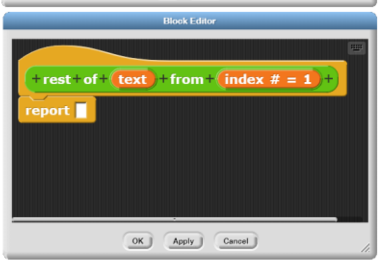
Level: *from middle school* Materials: *Stringoperations*

Like its predecessors, *Snap!* includes a minimized set of methods that work with strings. These include

- *join <string1> <string2>…* : the operator for concatenation of multiple strings. The result is a new string. The operator can be extended with further arguments using the arrow keys.[64]
- *split <string> by <char>* : the operator for splitting a string into a list. The splits are done at the specified characters, typically spaces.[63]
- *letter <n> of <string>* : returns the nth character of a string.
- *length of text <string>* : returns the length of a string.
- *unicode of <char>* : returns the Unicode of a character.
- *unicode <n> as letter* : returns the nth Unicode character.

Other string operations are located in the libraries. They can be imported via the File menu. The new blocks are then located under the *Make a block* button in the *Operators* palette.

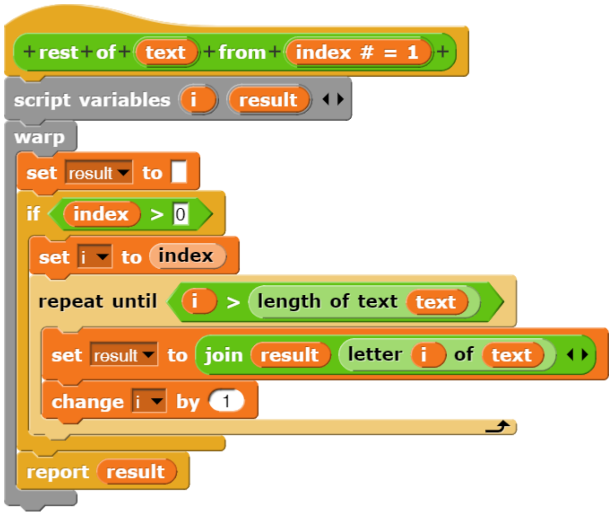




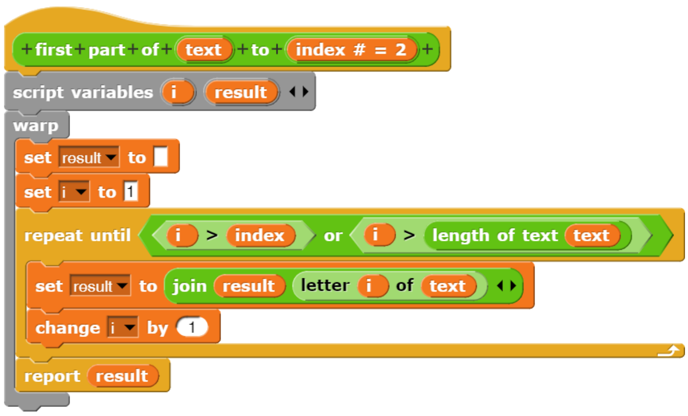
We want to take a different approach here by building any needed methods from the basic operations. First, we want to write a method *rest of <text> from <index>* that returns the rest of a string starting at a certain index. So, we create a new block, this time assigning it to the *Operators* palette so that it appears nice and green with the string operators. Since this is a function, we check "*reporter*", and because of course we want others to benefit from our work, we leave it at "*for all sprites*". We can insert the parameters at the + signs between the words of the method header, as already described several times. We specify the type as *text* or *number* and set the parameter *index* to the default value *1*. Both will be displayed in the method header as *index # = 1*.

[64] The block can additionally perform operations with other data types (see there).

In the script we copy all characters of the text from the index value into a string variable *result*. We return this as a function result using the *report* block. To make the whole thing nice and fast, we wrap it in a *warp* block.

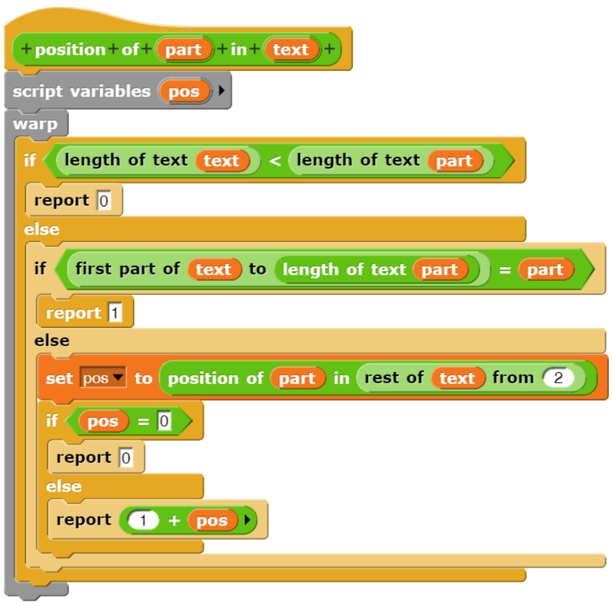


In a very similar way, the function *first part of <text> to <index>* returns the beginning of a string.

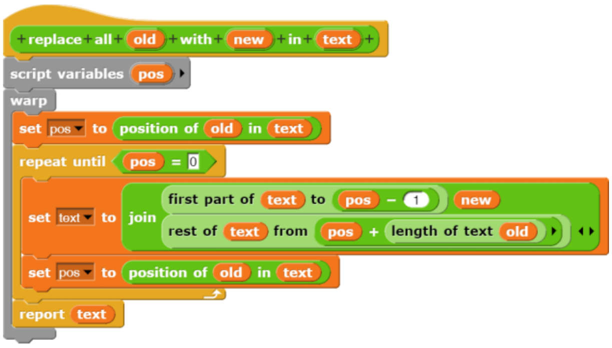


With both functions it is easy to get a snippet from a string.

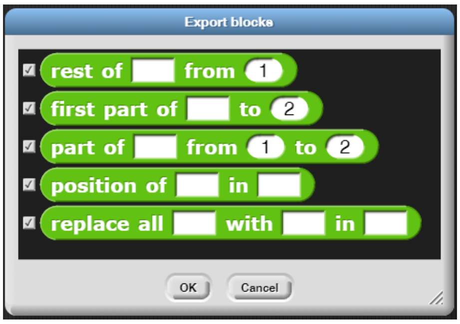
And the position of a substring in another string can also be determined - nicely recursively. If it is not present, then 0 is returned.



This makes it easy to perform standard operations such as replacing in strings.



So that we can delight mankind with these new opportunities, we export the created blocks to a library. To do this, we select *Export blocks ...* in the file menu and then select the blocks to be exported - all of them, of course! We get a file *Stringoperations blocks.xml*, which we save in a suitable place. If necessary, we can load the blocks into other projects via the file menu.



13.2 Vigenère-Encryption

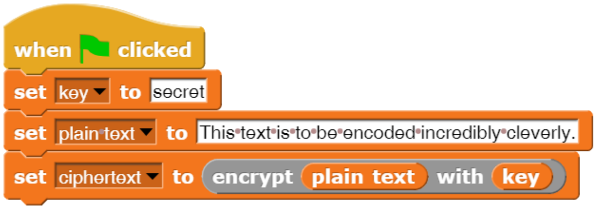
Level: *high school* Materials: *Vigenère encryption*

Vigenère encryption is an extension of Caesar encryption in which each character of the plaintext is shifted by a number in Unicode derived from a key character. Usually, the key is shorter than the text to be encrypted, so you simply extend the key until it is at least as long as the plaintext.

<u>Example</u>:	Plain text:	THISISAFULLYSECRETTEXT
	Key:	NOTHING
	Extended key:	NOTHINGNOTHINGNOTHING

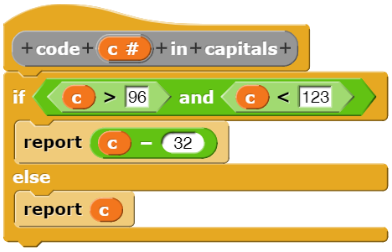
Thus, the first character of the plaintext (*T*) is shifted by 14 characters (*N* is the 14th character), the second character (*H*) by 15, the third (*I*) by 20, and so on. If characters larger than *Z* are obtained, then the characters are shifted cyclically starting at *A* - as usual in Caesar encryption.

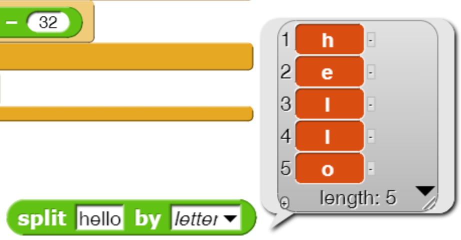
We write a small script that specifies the key and the plaintext and lets us determine the ciphertext using a function. So only the encryption method is interesting.



Since we are working with the character codes, we need the two blocks from the *Operators* palette: *unicode of* <…> and *unicode* <…> *as letter*.

First of all, we want to be able to convert codes from the lowercase range (97 ... 122) to uppercase codes when needed. This is done by subtracting the value 32 from the character code if necessary. Then we generate a list of character codes from the passed plaintext, a character string, which is to be called *textcodes*. A list is created from a string by applying the *split ... by ...* block. We pass the code of this function to the *map <function code> over <list>* block, which can be recognized by the gray ring around the function block. That is, the function is not executed first, as usual, and then its result is passed, but the program code of this function is passed to be executed in the *map-over* block. In this case, the "mapped" function consists of first determining the *Unicode* of a character and then passing it through the *code in capitals* function. From this list, we still throw out any invalid codes with a value less than 1. We store the code lists of plaintext and key in the variables *textcodes* and *keycodes* respectively.

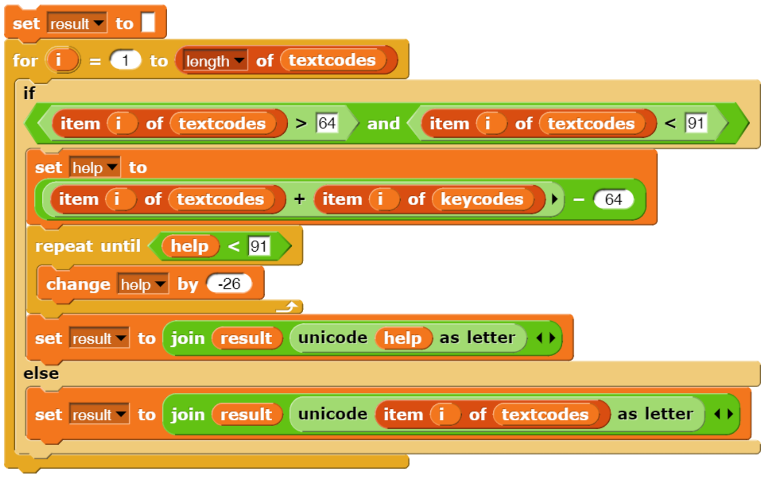




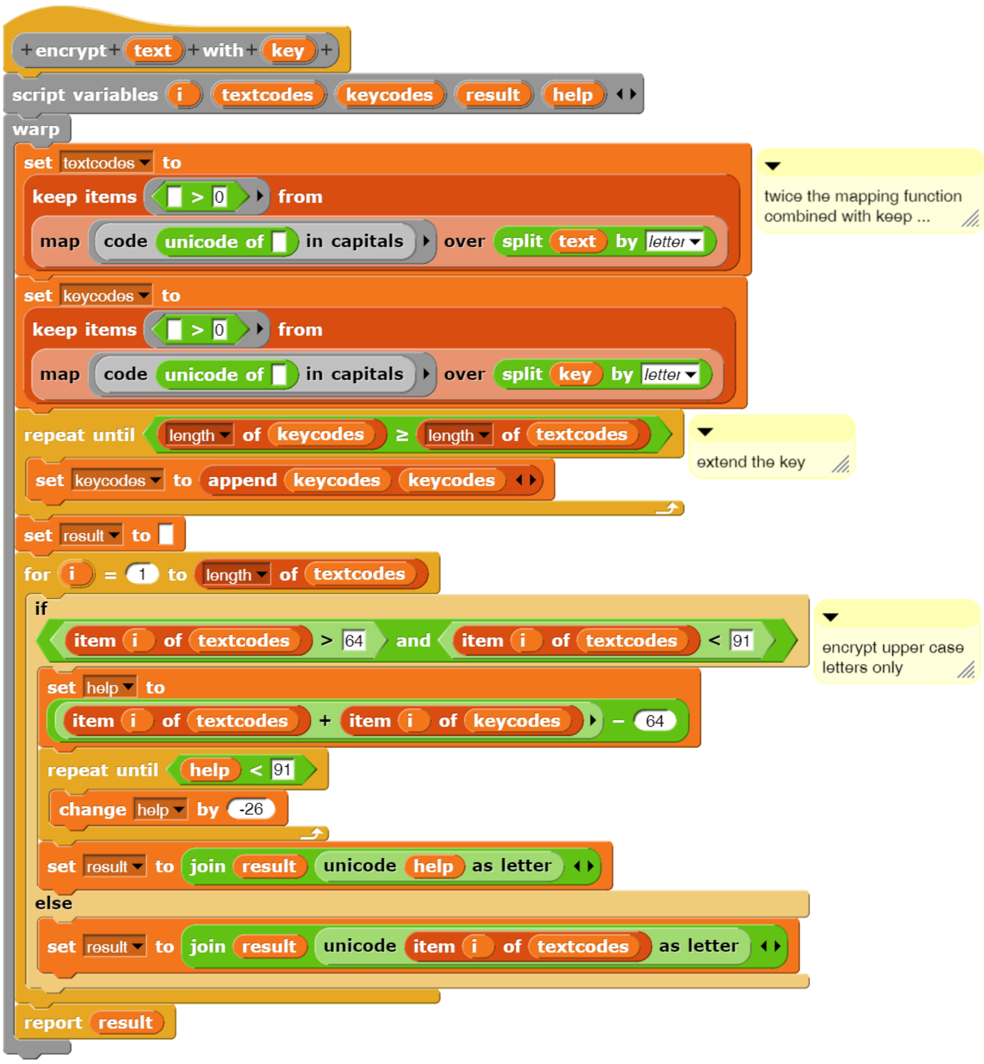
Next, we extend the *keycodes* list by the codes of the *key* until the list is at least as long as the *textcodes* list. This is done here by doubling the *keycodes* list using the *append* block each time.

Now we just have to apply the Vigenére procedure, in this case only to the codes of the letters. Instead of "mapping" a function, this time we use the *for* loop.

With their help we go through all characters of the *textcodes* list and encode them as specified.

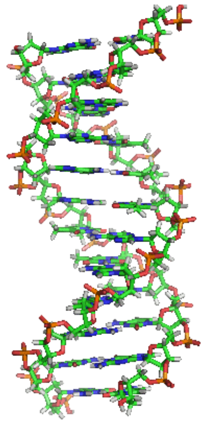


The complete process:



13.3 DNA-Sequencing[65]

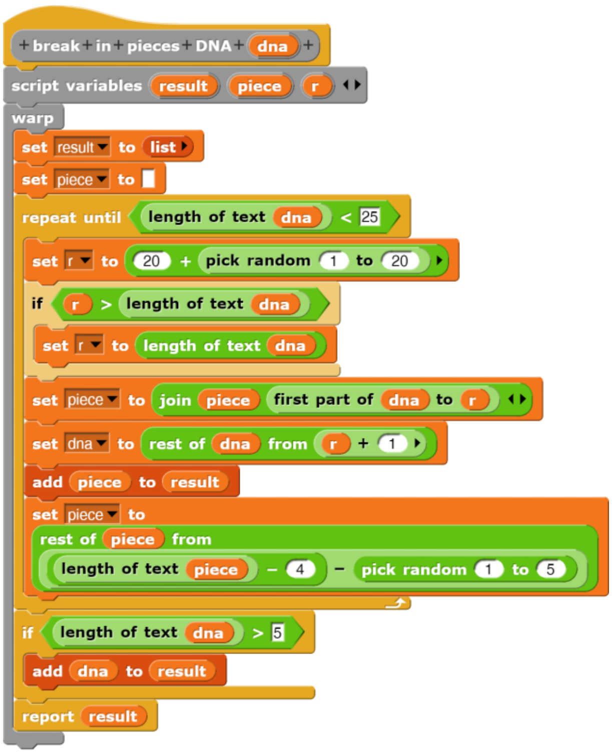
Level: *high school* Materials: *DNA analysis*

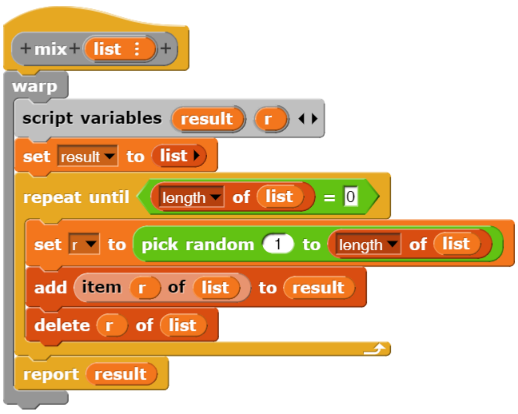


DNA-Helix

In bioinformatics, partial sequences are extracted from a broth of biomolecules containing fragments of DNA chains. From these, the entire DNA is reassembled. Here, we use a highly simplified model in which the partial fragments are represented by strings consisting of the characters *A*, *C*, *G*, and *T*. The partial fragments are then reassembled. The fragments "overlap" partially, so that the original DNA can be reconstructed from matches at the chain ends.

First, we need "DNA". Sequences can be found on the Internet. But since the meaning of the sequence is not important here, we simply generate it randomly. The product, a long string, we chop up, i.e. we split it into pieces of different length, which partly overlap. We accomplish this task by inserting a piece of the end of the predecessor at the front of a chunk. In the first section, this piece is empty. We use the string library that we created in chapter 13.1.

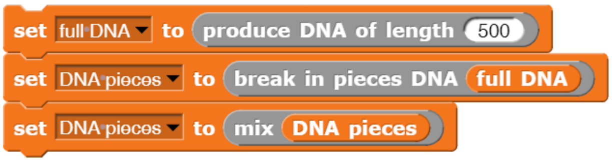




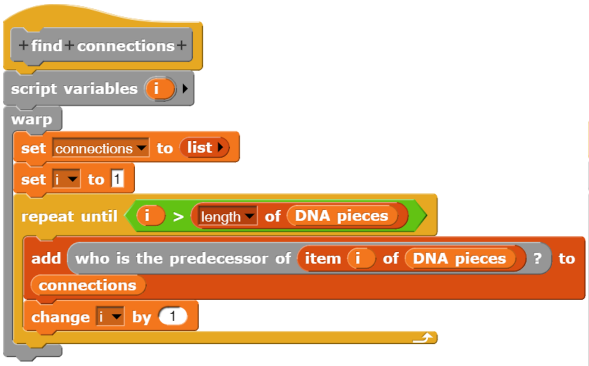
The sections are still in the correct order, so reconstruction would not be a problem. We change that by mixing up the order.

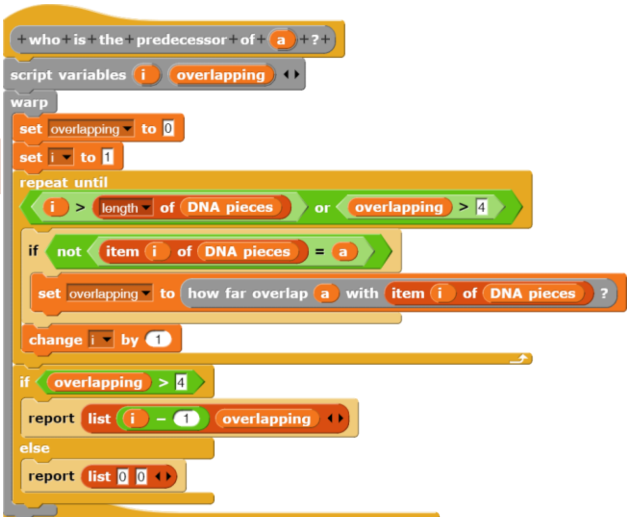
[65] A short description can be found e.g. under http://molgen.biologie.uni-mainz.de/Downloads/PDFs/Genomforsch/Modul10B_Skript2015-Hankeln.pdf. Picture from https://de.wikipedia.org/wiki/Desoxyribonukleinsäure

We then use the following command sequence to obtain the "soup" of DNA pieces we are looking for.

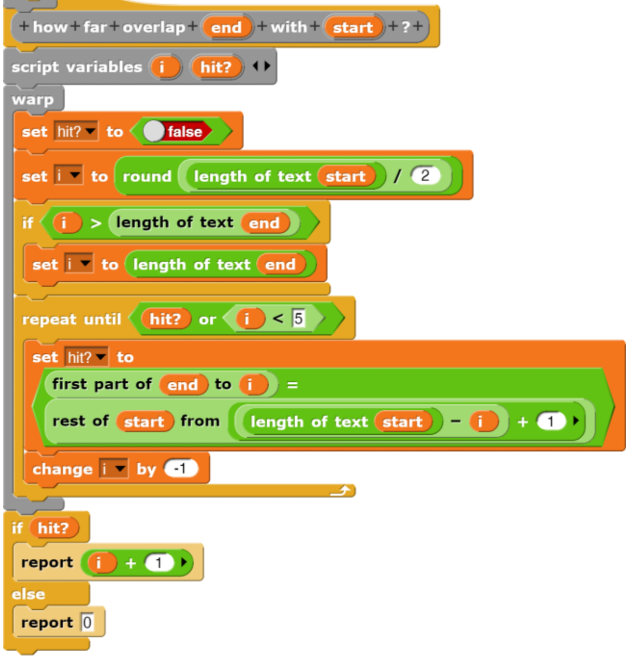


To reconstruct the original DNA from this, we need to determine which fragments were once connected. We create a list called *connections*, in which we enter the predecessors and the length of the overlap. Since the first fragment has no predecessor, its overlap length is zero.





One piece of DNA was "attached" to another if a sufficiently long overlap can be found. Since similarities can also be random, we define "sufficiently long" as "*5*". For a given sequence, there are four ways to "guess" the correct character for each place. So, the probability of generating the character correctly by chance is *0.25*. For five characters it is then $0.25^5 = 0.00098$. This is sufficiently "unlikely" for us.



So, the only remaining problem is to determine whether and if so, how far two DNA sequences overlap. We place them (mentally) on top of each other from the middle of the first one and then move the second one step by step "to the right" until we either detect an overlap or until we are too close to the end of it.

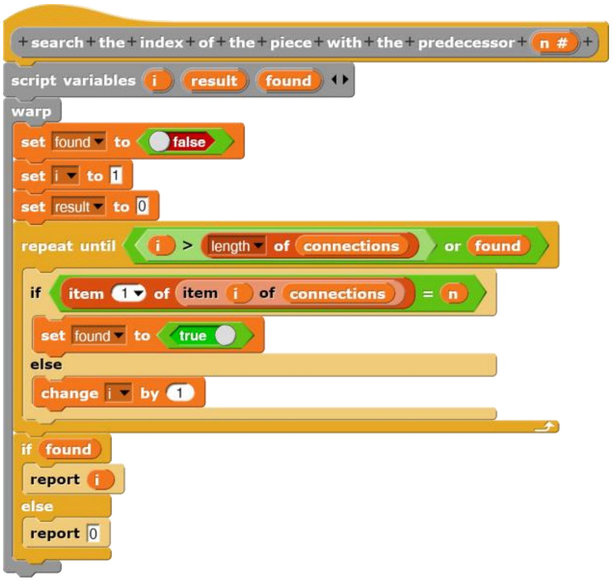
Table view

16	A	B
1	12	10
2	4	6
3	2	6
4	5	6
5	7	9
6	16	8
7	14	8
8	15	8
9	0	0
10	0	0
11	8	7
12	3	8
13	9	10
14	13	10
15	6	7
16	1	10

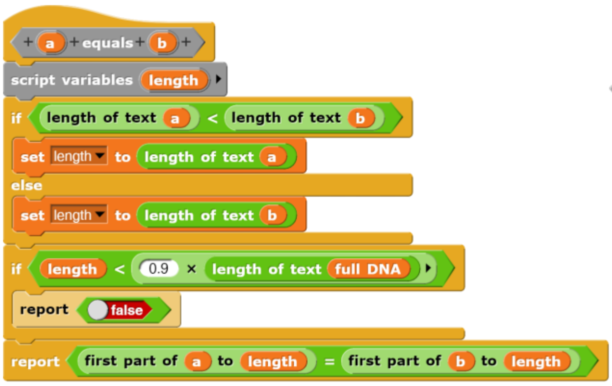
OK

Now we should be able to reconstruct the original DNA from the list of connections. We do this by searching through the list of connections, starting with the value *0* of the first piece. To do this, we search the connections for the element whose first entry corresponds to the value *n*. We return this index if necessary. We return this index if necessary.

Once we have found a piece of DNA, we append it to the previous finds and continue searching. The process ends when we get a zero as continuation index. Then we are either finished or an error has crept in during the search for overlaps. With the short overlap length, this happens sometimes.



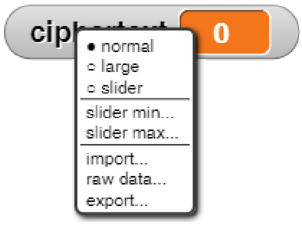
Finally, we check whether the DNA reconstruction was successful. The reconstructed DNA should already be approximately as long as the original - and of course it should also match the original in this area.



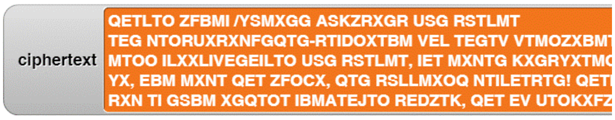
It does - most of the time.

13.4 Text Files, Server, and Frequency Analysis

Level: *high school* Materials: *Textfiles server and frequency analysis*

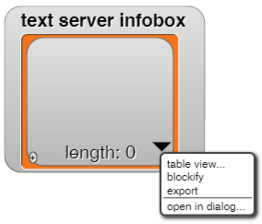


From obscure sources we have received the information that there is an incredibly secret text in the *ciphertext.txt* file on our computer. We even learn in which directory it is located. To be able to edit the text from *Snap!* we create a variable *ciphertext* and display it on the stage. As content it shows the zero. We select *import...* from the context menu of the displayed variable, navigate to the named directory and select the secret text. It appears in the variable.

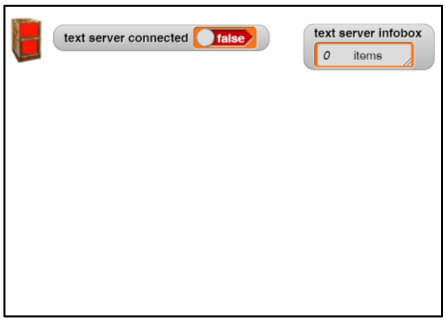


To be on the safe side, we want to save the text at another place immediately. We select the item *export...* from the same context menu and get the file *ciphertext.txt* at the bottom-left of the window, like saving a project. We can find it in the download directory of our computer. The described procedure is simple; however, it cannot be controlled by the program, but is executed "by hand".[66]

Text files are a simple but reliable tool to exchange data between different computers. For this to work, we need an *http server* (which may be the same computer if necessary) running a script that has the desired functionality - here: loading and saving text files. In this case we want to choose the server *snapextensions.uni-goettingen.de*, where the script *handleTextfile.php* is located. We draw two costumes for a *text server* sprite, indicating whether we are connected to the server - or not. The data exchange with the server should be logged in a variable *infobox*. By clicking the green flag our variables shall be initialized, where the one named *connection* gets a rather cryptic value.

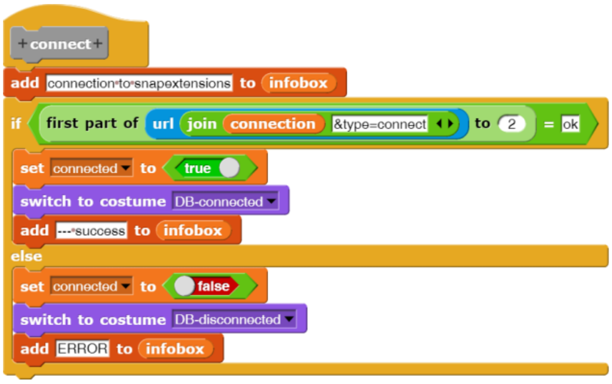


This consists of the server address, a login script and some variables - just PHP. We change our info box to "*table view*" using the context menu, which looks a bit better. The output window looks like this:

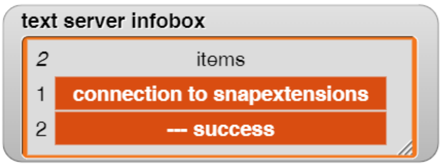


[66] But in the library SciScap! you can find corresponding blocks.

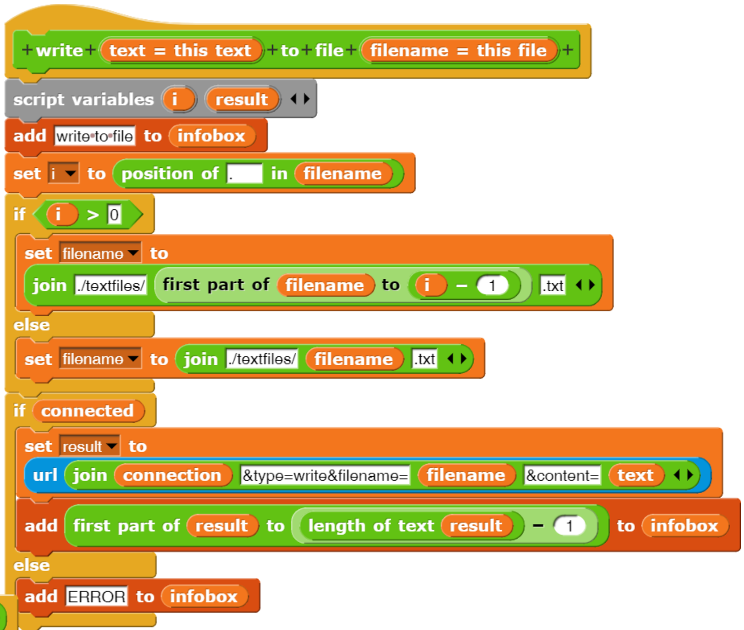
We need a connection to the server. This is done using the *url* block, to which we pass the required data. We log the success or failure in the info box.



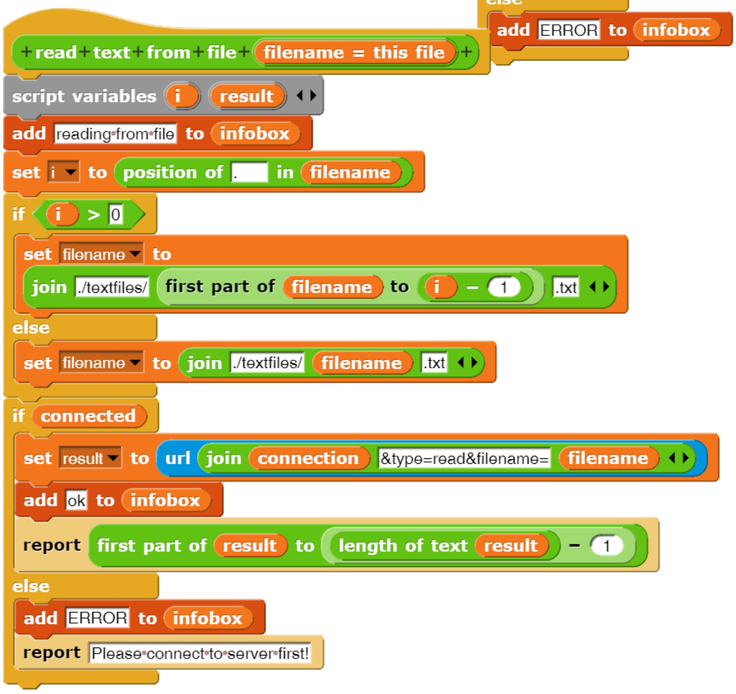
After executing this block, the connection to the server is established, but the text in our info box is only partially visible. We therefore click with the left mouse button on the column heading *items* and drag the column in width until all text is readable.



We want to write data to a file on the server. We specify the text to be written and the file name as parameters. First, we append the extension "*.txt*" to the file name if necessary and make sure that the file is stored in the subdirectory *textfiles* on the server. Then the *url* block passes the required data.

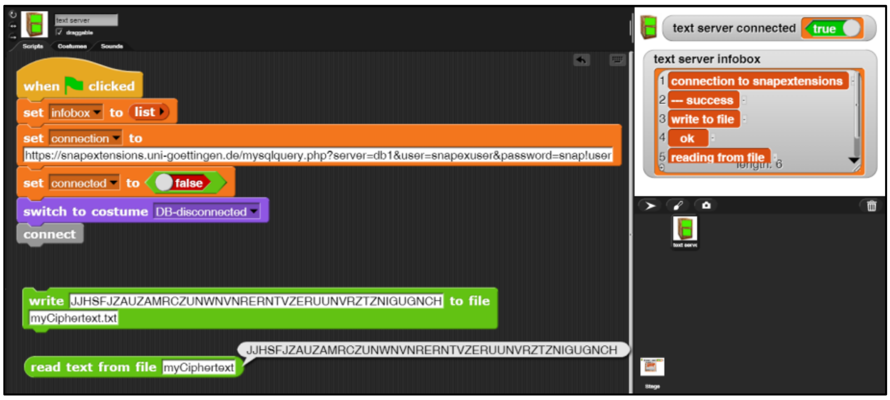


Reading from a file is done accordingly.



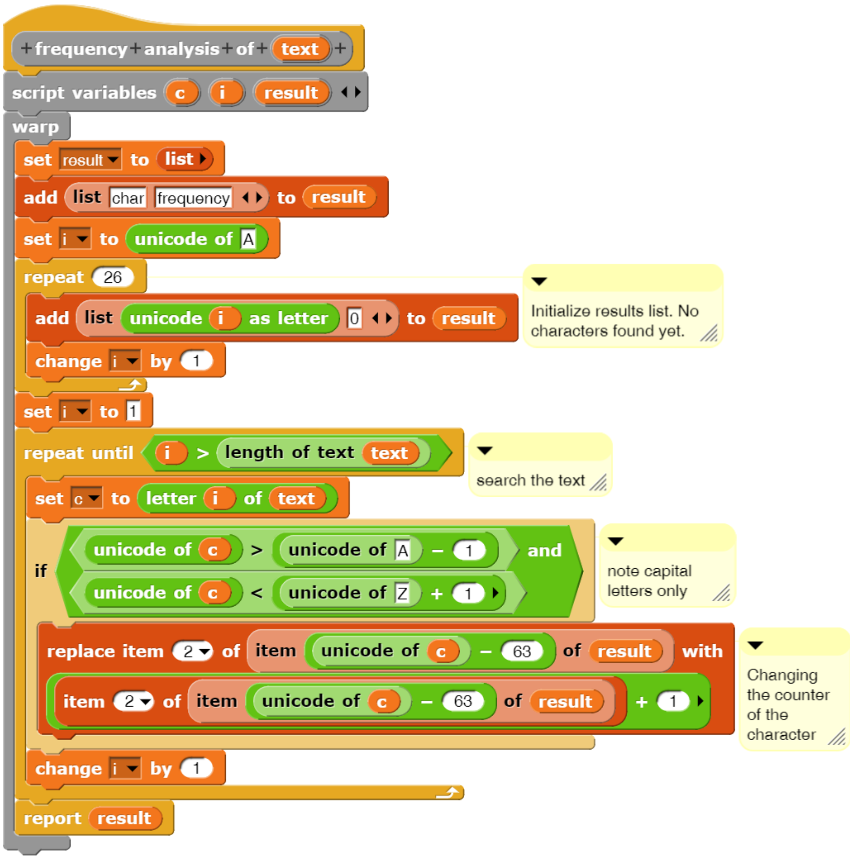
We export the text server sprite to an XML file and can thus use its functionality in other projects as well.

After a connection setup, a write and a read operation, our workspace looks something like this:



It doesn't help, we now have to decode the ciphertext. To do this, we perform a frequency analysis - i.e. we count how often the individual letters occur in a text. We can see from the ratios of the first three most common characters that this is a German text.

Since in German the *E* is the most frequent letter and it would be mean if the text had been written in another language, we store the list of frequencies in a variable *frequencies*.



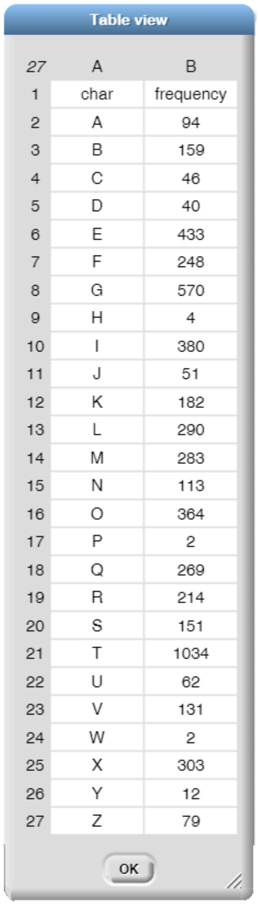
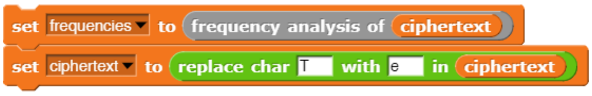


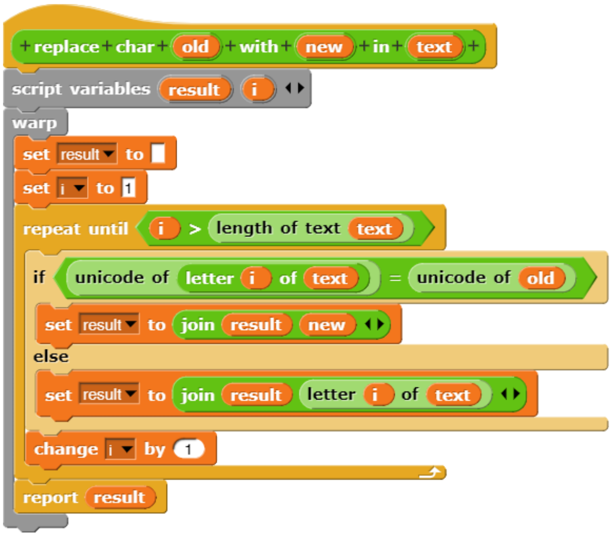
Table view

27	A	B
1	char	frequency
2	A	94
3	B	159
4	C	46
5	D	40
6	E	433
7	F	248
8	G	570
9	H	4
10	I	380
11	J	51
12	K	182
13	L	290
14	M	283
15	N	113
16	O	364
17	P	2
18	Q	269
19	R	214
20	S	151
21	T	1034
22	U	62
23	V	131
24	W	2
25	X	303
26	Y	12
27	Z	79

OK

Then we try to replace the capital *T* in the ciphertext with a lowercase *e* - because *T* is the most common. Our *replace* block is really not meant for so many replacements, so we quickly write a new one. In this new block, we distinguish between upper and lower case in text comparisons and therefore use the *Unicodes* of the characters.





Because the result is not too impressive, we need more substitutions. We assume an *n* behind the *G* and also perform this substitution.

We can look at the ciphertext quite well if we break it down into lines with. 

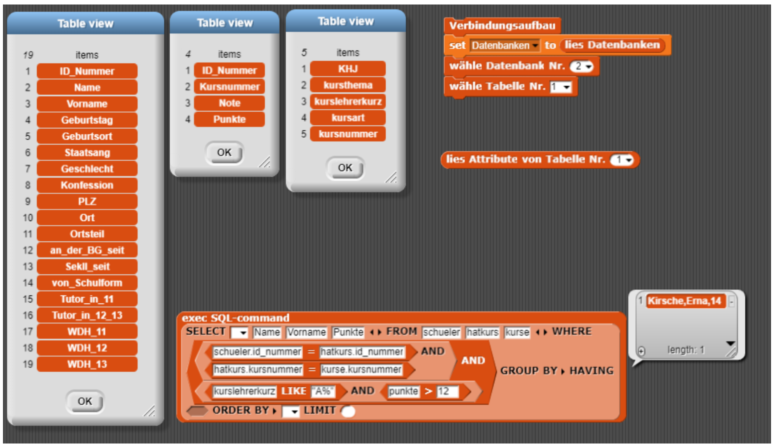
For example, we find words like *eEn* in line 2. We therefore consider the *E* to be an *i*.

That was a good idea! Let's keep searching and trying substitutions, then we'll eventually find the secret! You just have to persevere - there are only 23 letters left!

13.5 SQL-Databases

Level: *from middle school*

Materials: *SQL, SQL example*

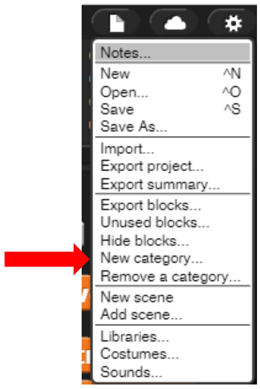


An important IT system task is the access to external data sources. On the one hand, the Internet is available for this purpose, and on the other hand, the use of SQL databases is common. Since the use of this type of application is somewhat complicated in many computer languages, it is often treated separately from algorithmics. This makes this subfield of computer science rather boring: you create ER diagrams on paper or query databases with special client applications, e.g. *PHPmyAdmin*, but you don't exploit the results further. With the help of *Snap!* this can be done differently!

Again, we need a server running either on another computer or also on our own, and on which - in this case - in addition to an *http server* and an *SQL server*[67], there is a PHP script called *mysqlquery.php*, to which we send the data required for an *SQL query* for the SQL server using the parameters *type*, *query*, *command*, ... The result of the query is then either an error message or a table with results. If necessary, the script prepares this table so that *Snap!* can display it as a list. The source code of this script can be found e.g. on *https://snapextensions.uni-goettingen.de*.

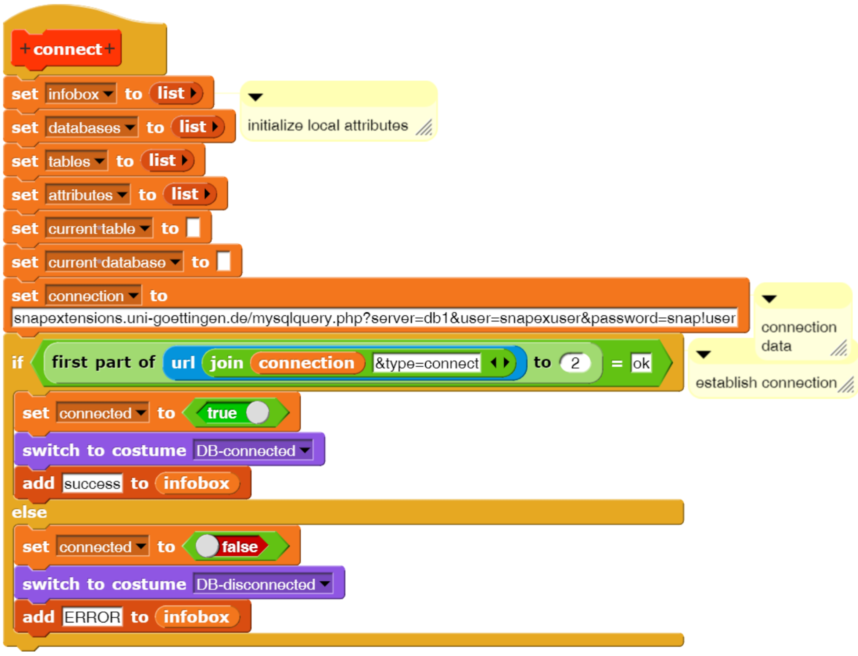
Similar to the last section, we create a sprite called *SQLserver*, which indicates by its costume if there is a connection to the database. Some attributes like *connection*, *connected*, *current table*, etc. store the current state, and a variable *infobox* logs what is happening. This sprite is saved as *SQLserver* and can be loaded when needed.

The new blocks necessary for SQL queries are declared globally so that they are easily accessible for queries outside the server sprite. They are stored in the *SQL blocks.xml* file and must be loaded additionally. Since this is a completely different *category* of blocks than the ones present in the standard palettes, we give the system a new palette called *SQL*, in which we place the SQL blocks.

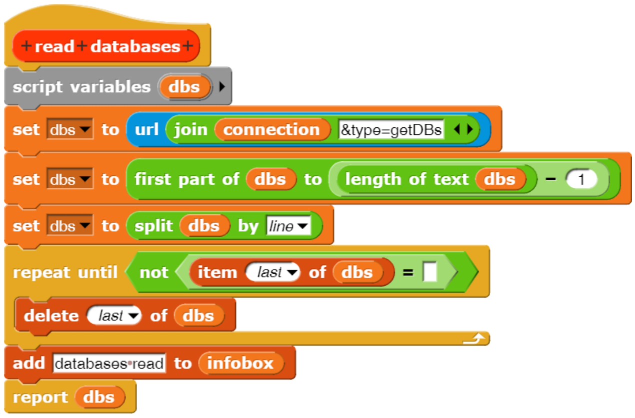


[67] The project „In the supermarket " also uses a SQLite-Server.

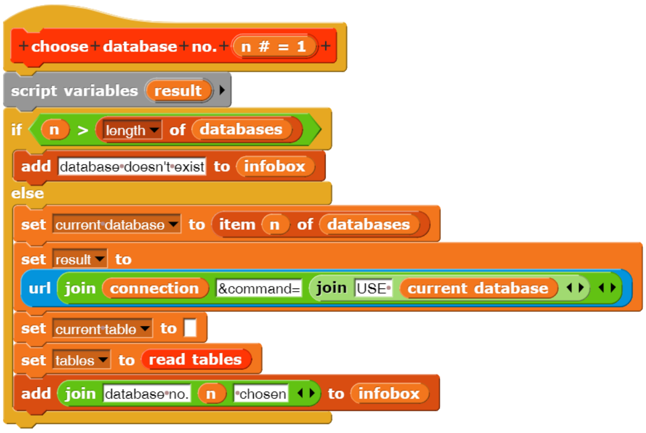
First of all, we need an access to the external SQL server. For this we set up a block *connect*. There, the local attributes are initialized, and the connection data is stored in the variable *connection* so that it does not have to be reentered each time. Then the connection is established and the success or failure is noted in the variable *connected*.



The reporter block *read databases* is used to ask the SQL server for the existing databases. These are returned as a list. For the actual query, only the value "*getDBs*" must be appended to the connection data as "*type*".

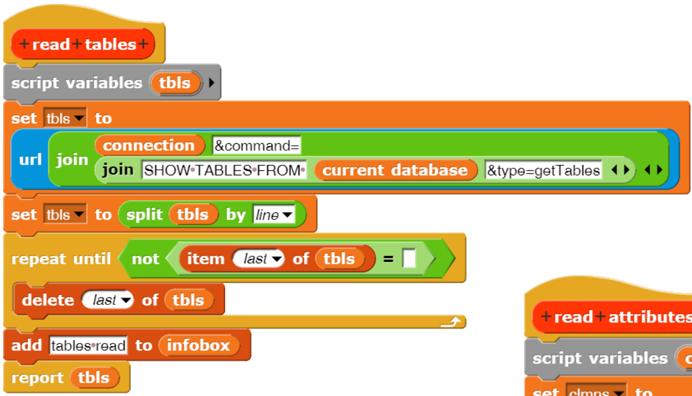


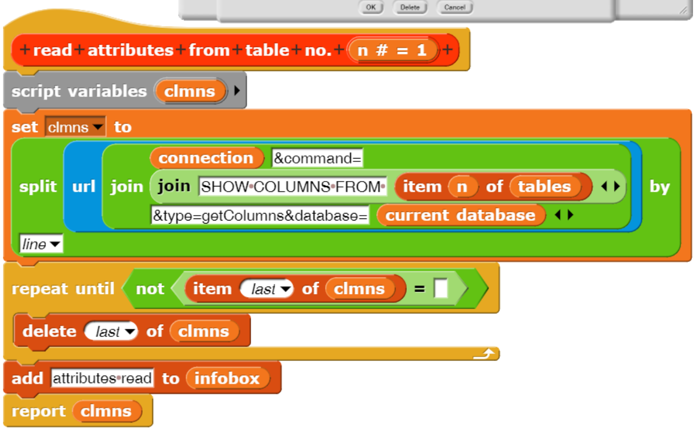
The connection establishment and the selection of a database can be saved as a block sequence. The last block selects the specified database. Interesting is the small arrow next to the parameter. If you click on it, a selection list with the possible values appears.



A selection list can be created in the block editor by right-clicking in the dark area. You will then get a small context menu with the item *options...* In the pop-up window *Input Slot Options* the possible input options are entered.

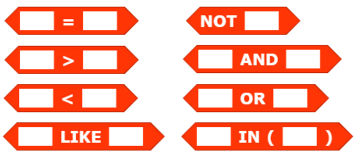
In a very similar way, for the selected database it is determined which tables it contains and from which attributes the tables are constructed.





Thus, with the help of the new blocks we can find out which tables are present and what attributes they contain. In the context menu of the obtained list, the result can be permanently displayed using the "*open in dialog*" option. This way we can clearly arrange on the screen the values needed for queries.

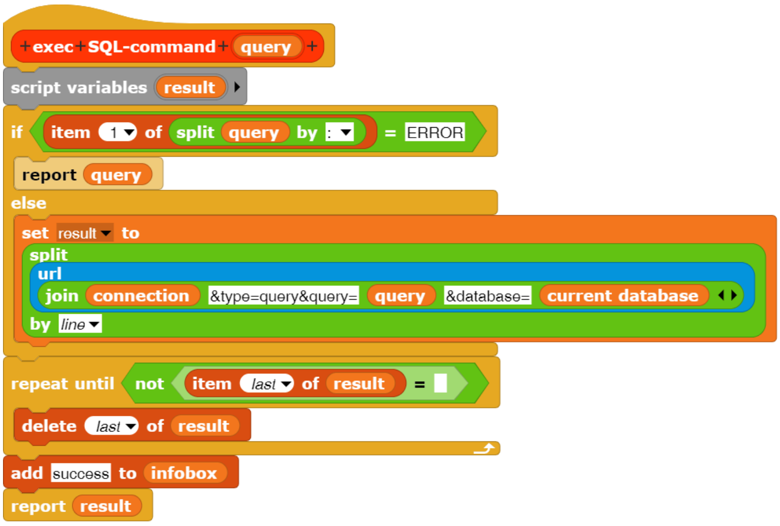
We have now created the prerequisites for issuing queries to the database. For this we still need SQL aggregate functions and operators. These can be used to interactively compose SQL queries using the data from the "Table views" and two types of SELECT blocks.



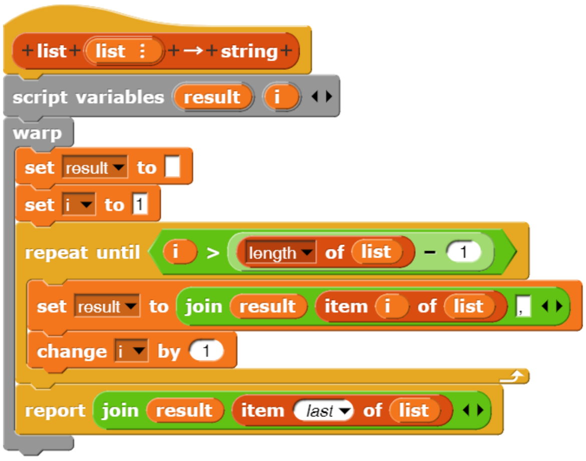
It should be noted that only the texts of the queries are generated by the blocks! The queries are not (yet) executed.

These blocks can now be used to create - and control - SELECT queries.

For the execution of such queries, we have one - last - block available. An SQL command is passed to this block either as text or as the result of a SELECT block. In the obtained response list, any empty entries are deleted.

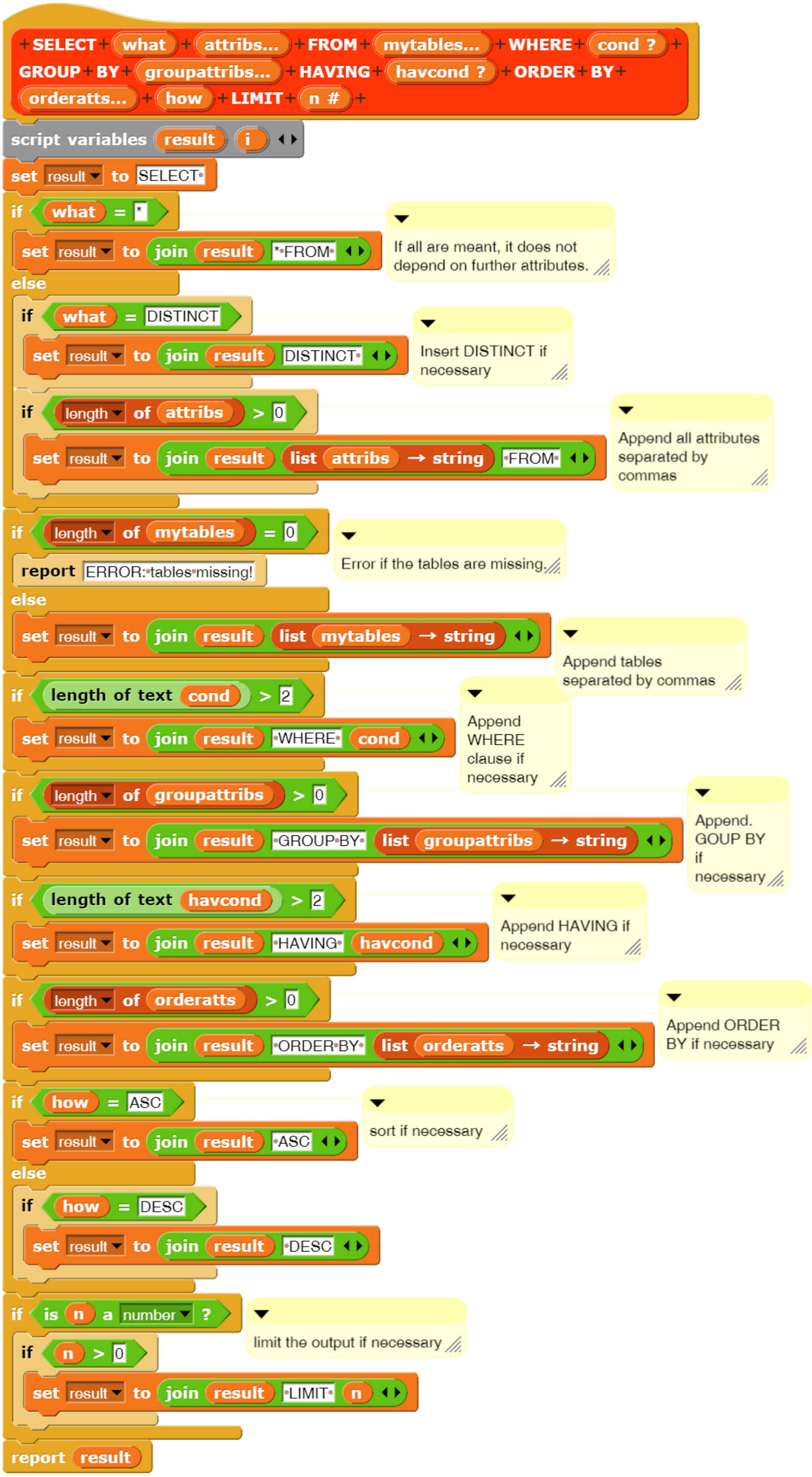


The simple SELECT block assembles an SQL query from the parameters. It uses a reporter *List → string* for this.

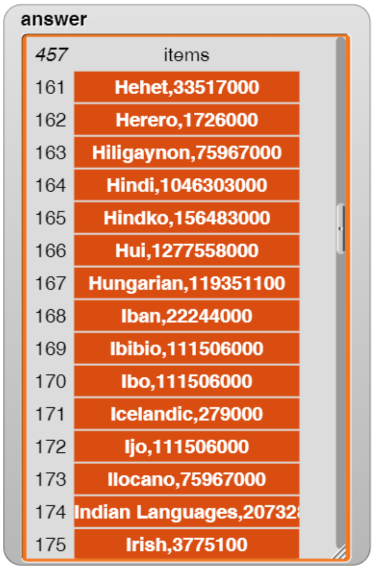


With this we can start a first attempt:

With the full SELECT block it is not more complicated - only longer.



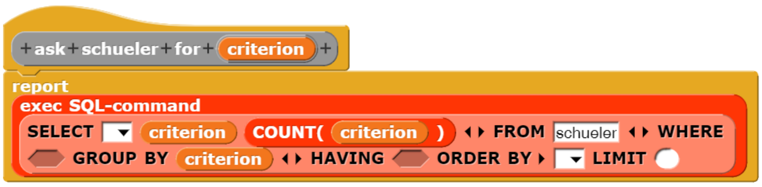
We can now work with this: How many people speak which language?



Amazing!

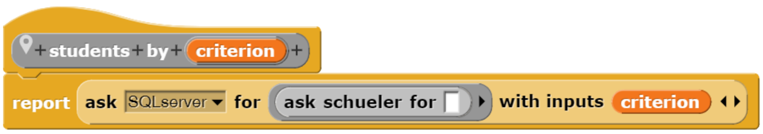
The resulting SQL library is intended for testing SQL commands interactively and then - if successful - incorporating them into new blocks that allow the database to be used without SQL knowledge. We illustrate this with a simple query.

For a new project, we first import the *SQL blocks* library, then the *SQL server* sprite. In addition, we create an *SQL user* sprite. This then asks the SQL server to establish a connection.



After that, query blocks can be created, which e.g. determine the data important for school statistics.

These can then be used for special purposes.



13.6 Tasks

1. A simple form of **block ciphering** is to insert the text to be encoded into a table with several columns from left to right and from top to bottom. If the last row is not filled, then any characters are inserted. The encrypted text is obtained by reading the table from top to bottom and from left to right.

   Example:

```
THIST    →    TEIRLRHXSEYEITIDSTSINIEXTSCBCX
EXTIS
ISINC
REDIB
LYSEC
RETXX
```

   What is the key? Realize the procedure.

3. **Genetic algorithms** simulate nature's evolutionary process by randomly generating new candidates to solve a problem again and again. In this case, **palindromes** are sought, words that are the same when read forwards and backwards. The procedure consists of an initialization in which a random initial population is generated. In this case, a set of random words. Then a loop is run over and over again in which candidates for recombination of individuals are selected based on a fitness function. From two candidates (at least) one new one is generated. Afterwards random changes (mutations) take place. In the resulting new generation, the "best" candidates are selected for the next run on the basis of the fitness function (selection).

4. The determination of the **Levenshtein distance** between two character strings is used to determine the "degree of relationship" of the character strings. Typically, these are DNA strands from the characters A, C, G and T.

   a: Find out about the process.

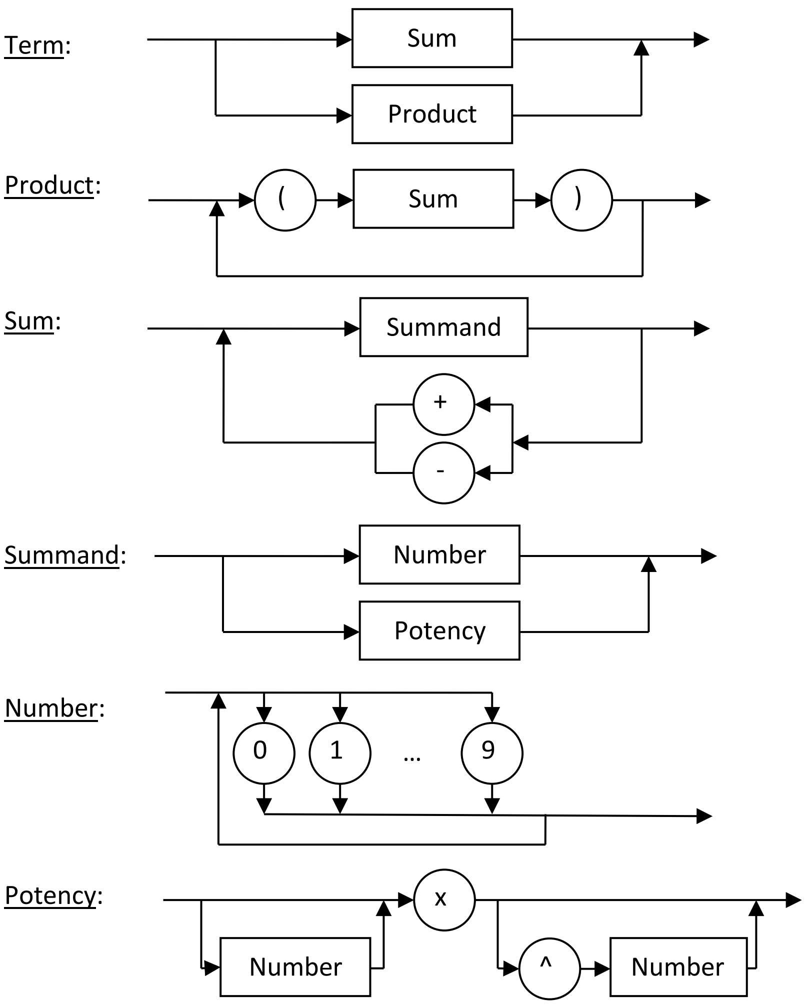
   b: Realize the procedure.

14 Computer Algebra: Functional Programming

Level: *high school*   Materials: *Computer algebra*

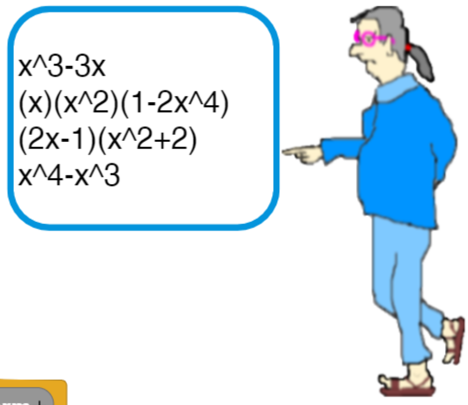
14.1 Function Terms

We want to develop a small "*Computer Algebra System*" (CAS), which on the one hand illustrates the *top-down method* and on the other hand shows how to *program functionally* with *Snap!* For this we have to define what we want to understand by function terms e.g. via syntax diagrams.

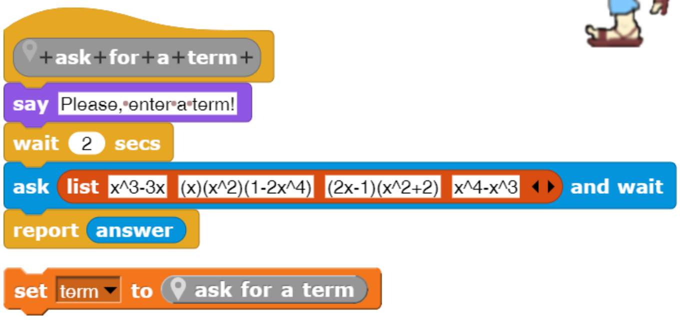


Function terms are therefore e.g.: 3 4x (2x-1)(x^2+2) (x)(x^2)(1-2x^4)

14.2 Parse Function Terms



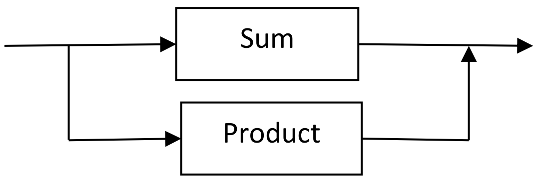
To work with function terms, of course, we need someone who knows something about them. We therefore draw *Gundolf de Jong*, a talented young mathematician, and then make him smart. First of all, *Gundolf* has to be able to read in function terms. To do this, he asks the user for an appropriate input using the block *ask <question> and wait* from the *Sensing* palette. We don't use the simple form here, where something always must be entered, but we pass a selection list, from which one of the options is chosen by a mouse click.



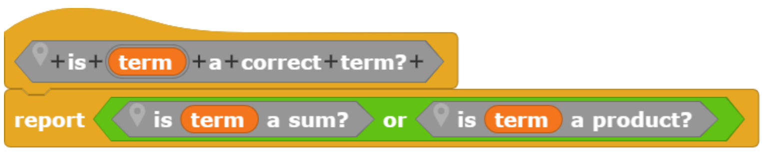
We move the whole thing to a method of *Gundolf*, which we define as a function. So, we select the oval block shape in the block editor. If we have declared a variable, e.g. named *term*, then we can assign the result of the input to it.

Next, we verify that the input is correct. We move the corresponding methods into a sprite called *Parser*. In this sprite we want to program functionally on the one hand, but on the other hand we want to solve the problem in a top-down way.

We create the global block (*for all sprites*) *is <term> a correct term?* as a *predicate*, which accordingly can only return the results *true* or *false*. After that we have a nice title, but unfortunately no content yet. Nevertheless, we can already use the block in scripts - just like other blocks. On the one hand this allows recursive operations, on the other hand it is suitable for top-down development. Since according to the syntax diagrams correct terms are either sums or products, we move the problem there by creating two corresponding predicates - still empty - locally (*for this sprite only*), because the rest of the problem is none of the business of external observers.



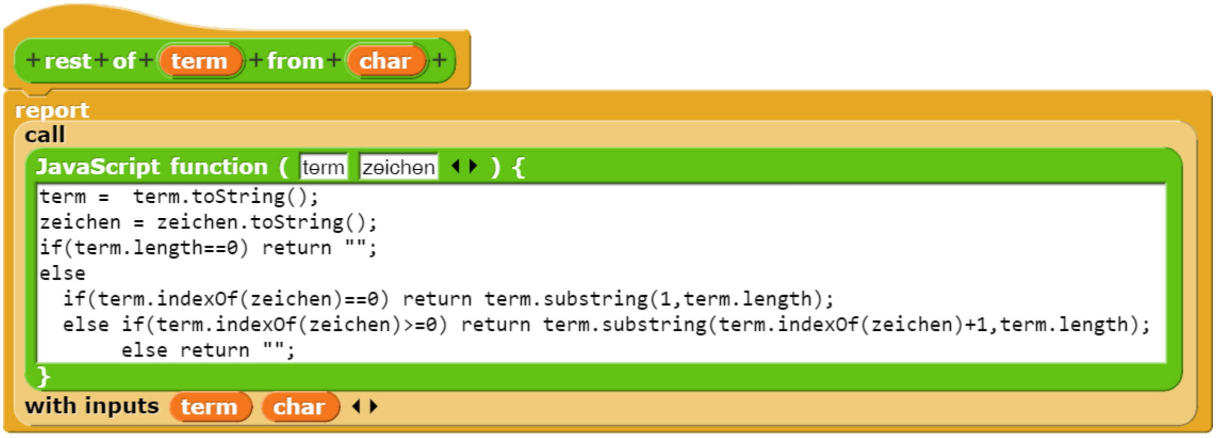
Snap! now evaluates logical expressions "*lazy*": the second expression is evaluated only if the first one does not already determine the result. We can therefore specify the predicate *is <term> a correct term?* completely with empty block hulls.

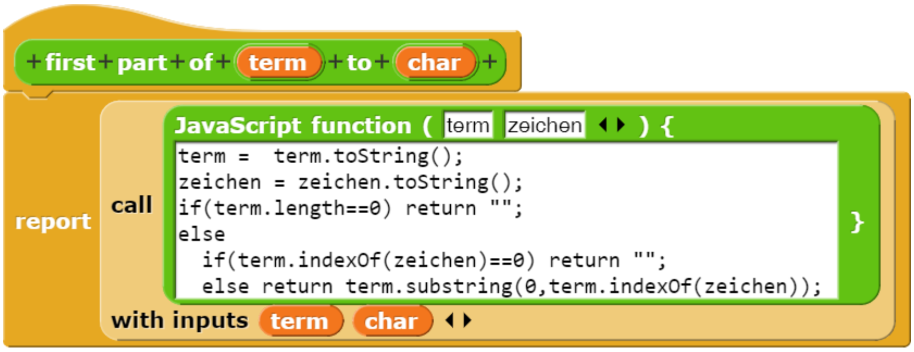


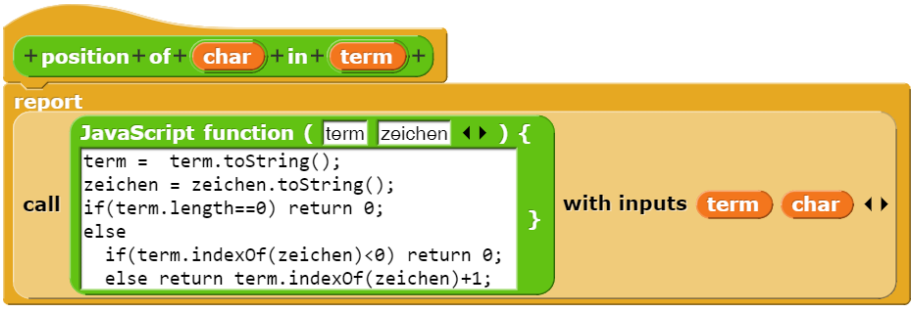
We continue this procedure for all elements of the language definition. The sum consists of either a single summand or a summand followed by the correct operator (+/-) and a sum. We can write this down directly if we have a predicate *is <term> a summand?* that is empty for now.

We have to be careful that our terms - strings - are not accidentally interpreted as numbers. For this reason, we have always set the type of the input parameter *term* to "*Text*". If we forget this, then the *string* "*123*", for example, could be interpreted as the *number 123*. The second element of the string is e.g. a *2*, but there is no second element in the number *123*. A corresponding access would lead to an error.

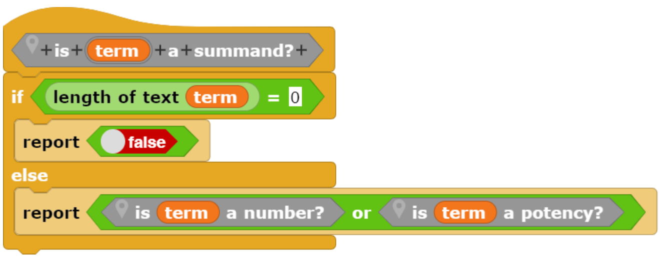
We need one more thing. The entered term is no longer examined as a whole, but we may have to split it into two parts: the *first part of <term> to <char>* and the *rest of <term> from <char>*. In addition, there is the determination of the position of a character in a string: *position of <char> in <term>*. In this case, we want to implement them as *JavaScript* methods, because time matters a bit.[68]





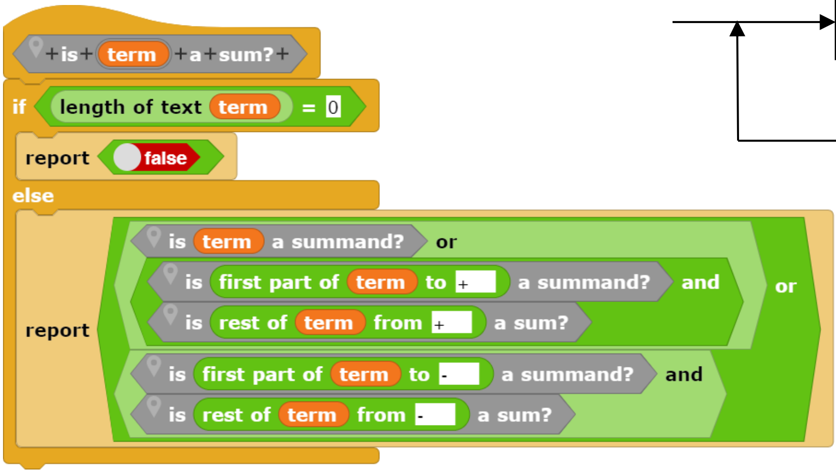


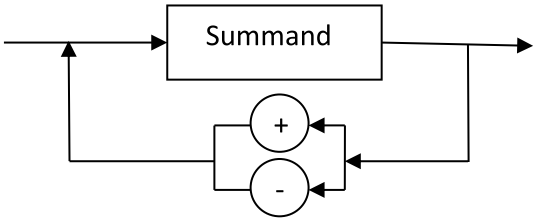
With this we write the predicate *is <term> a summand?* - with an additional security check.



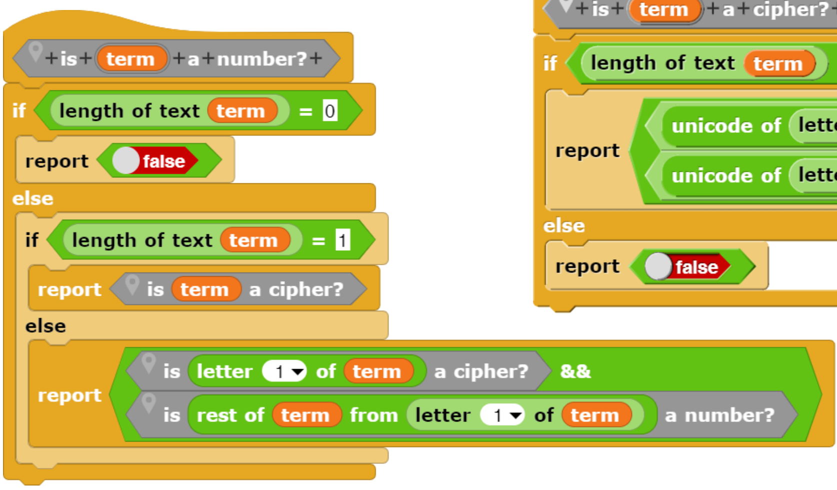
[68] For this, JavaScript usage must be explicitly allowed in the Settings menu.

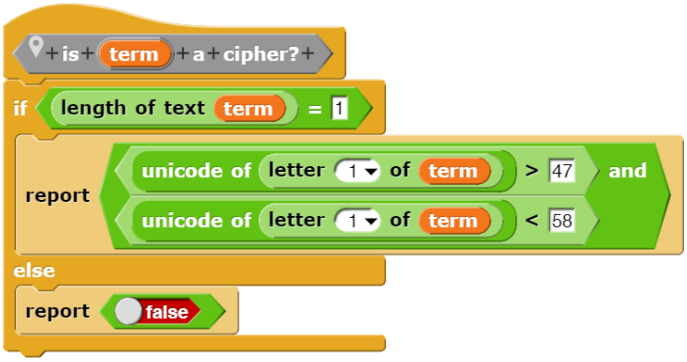
And now we can finally create the predicate *is <term> a sum?*.





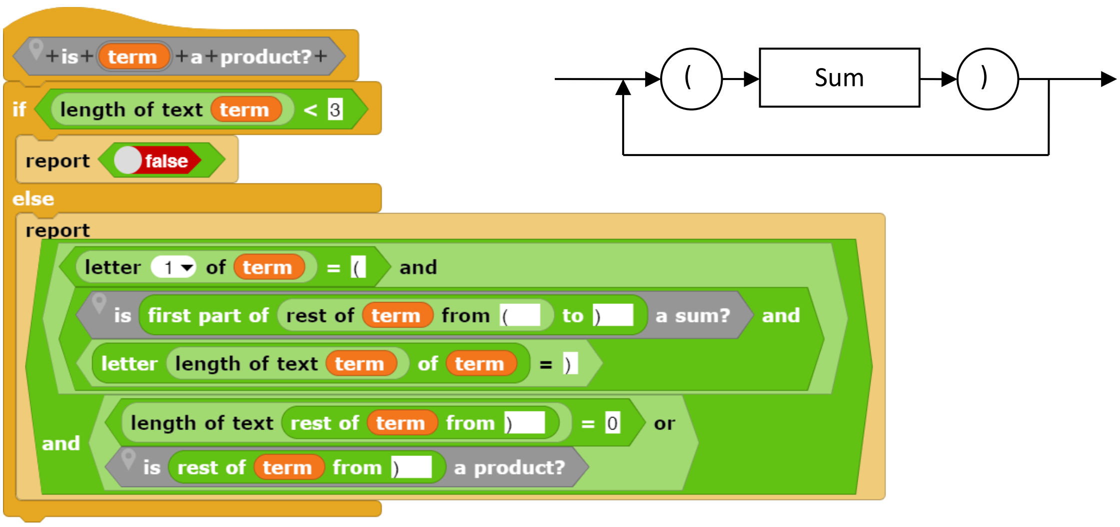
We are nearing the end. *is <term> a number?* is very easy to write if you know *is <term> a cipher?*:

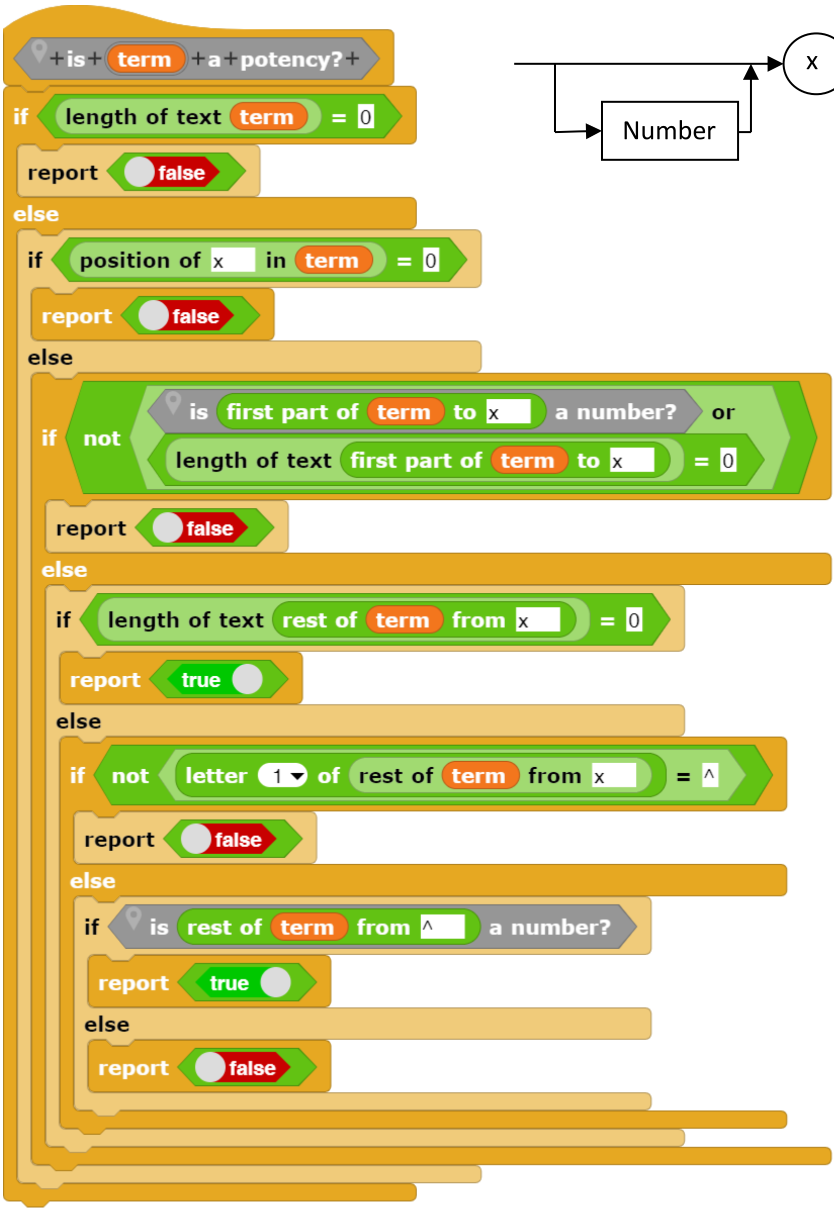


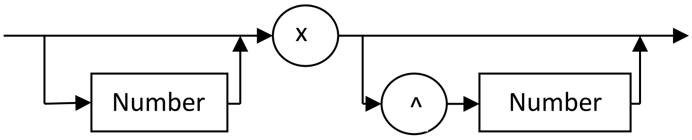


And how do you check a potency? This is also in the syntax diagram - we just have to copy all possibilities (see next page).

All that is missing now is the product, which can be formulated in direct analogy to the sum, because a product consists of either a parenthesized sum or one followed by a product.

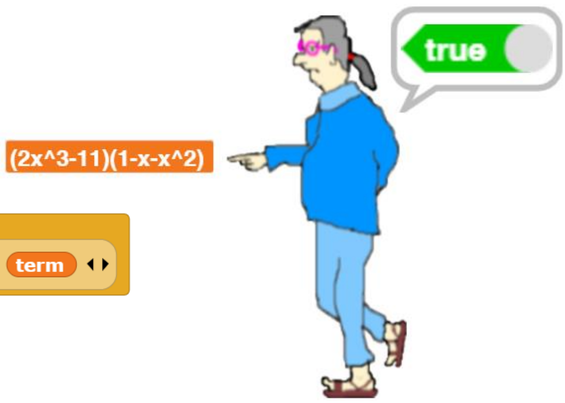


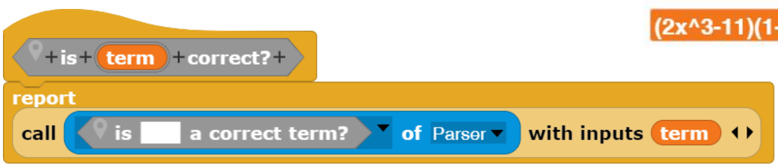




We can use it to check ("*parse*") whether a term entered corresponds to the selected syntax. If this is the case, we can continue working with it. Our mathematician *Gundolf* asks the *Parser* here.

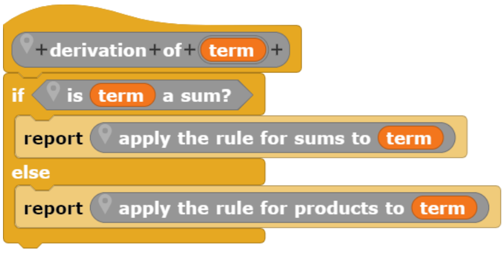
He of course wraps this query in its own block to make it appear that he himself could answer such a question.



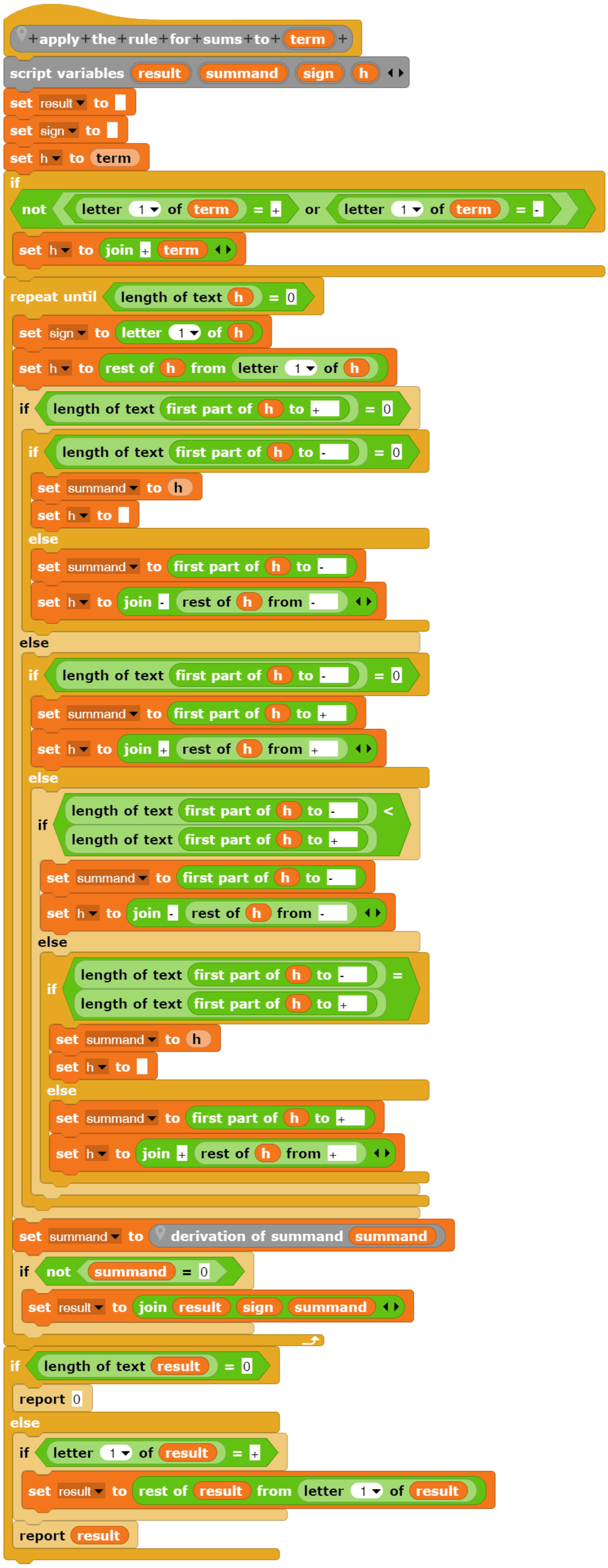


14.3 Derive Function Terms

We want to determine the first derivative of correct function terms. We collect the necessary methods from the *Parser*. Since there are only two possibilities for the inner construction of terms, the first approach is simple.



When applying the rule for sums, we have to determine the summands and derive them. Because we have defined numbers without a sign, we treat this separately in each case, i.e. we add a "+" if necessary and then split off the sign again. Subsequently, the different possibilities are treated according to the rules of mathematics.



Deriving individual summands is not particularly difficult.

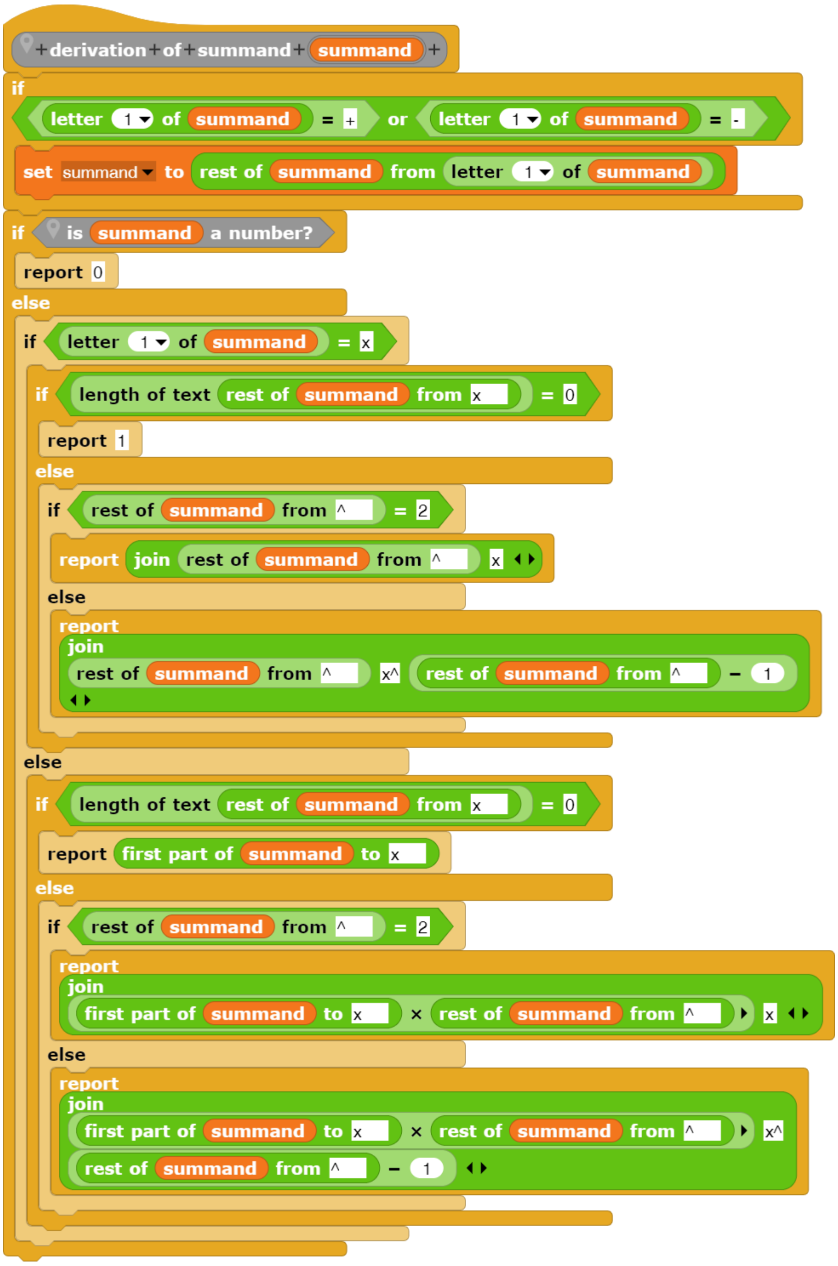
Numbers result in zero.

The derivative of x is one.

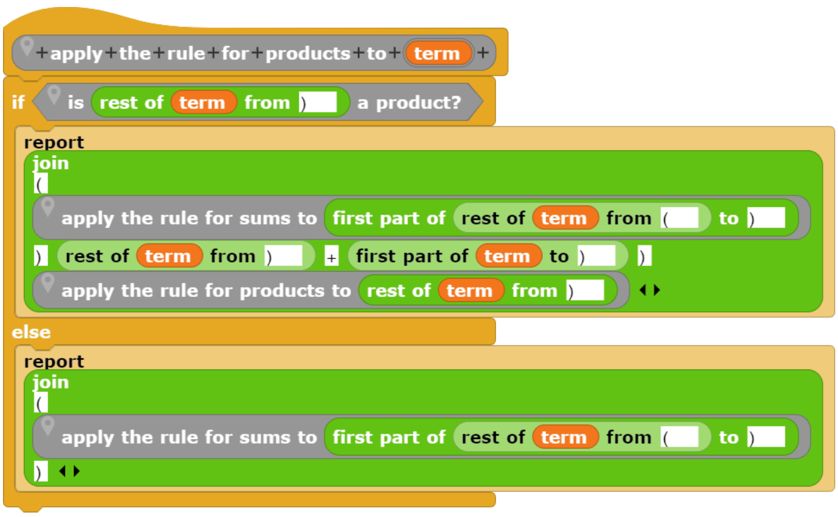
The derivative of x^2 is 2x,

otherwise, we get n*x^(n-1).

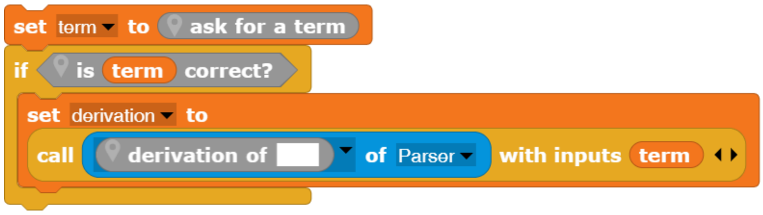
Accordingly, a factor before x is considered.

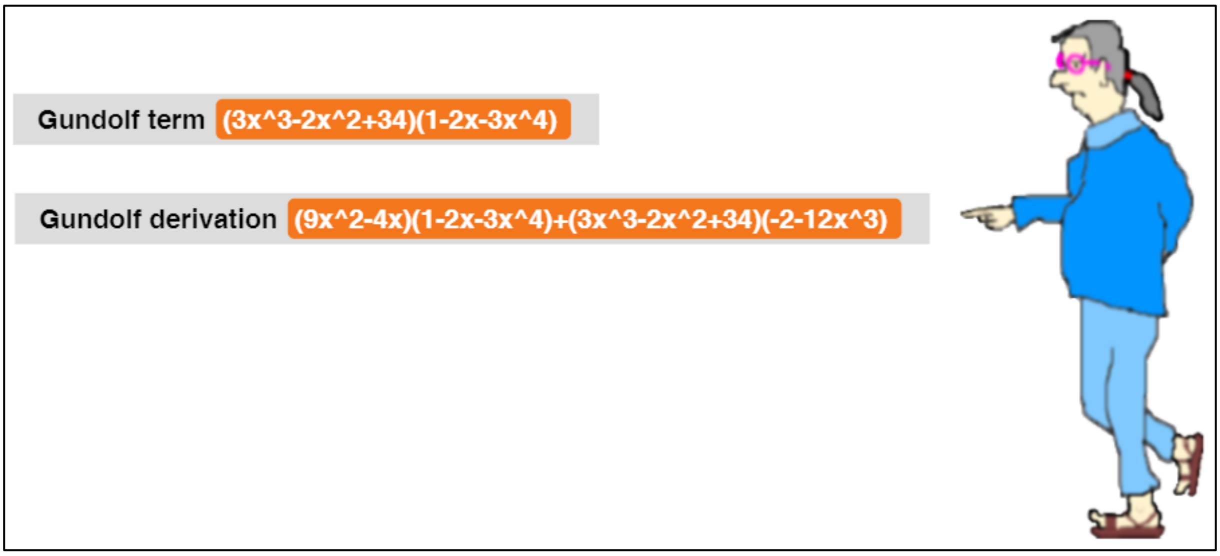


The only thing missing is the rule for products. We can simply write it down - adding some brackets.



The result is even quite readable:



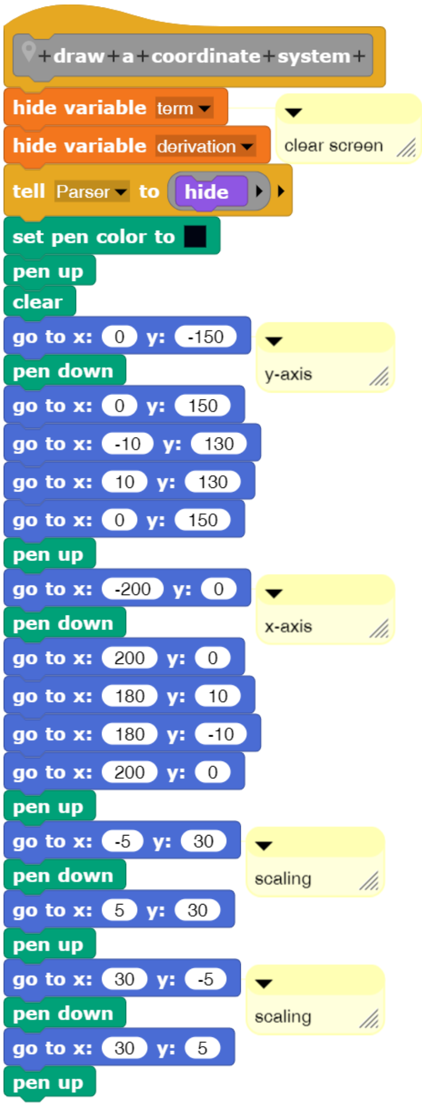


Note that the derivatives do not necessarily correspond to our highly simplified definition of function terms and therefore often cannot be "processed" further.

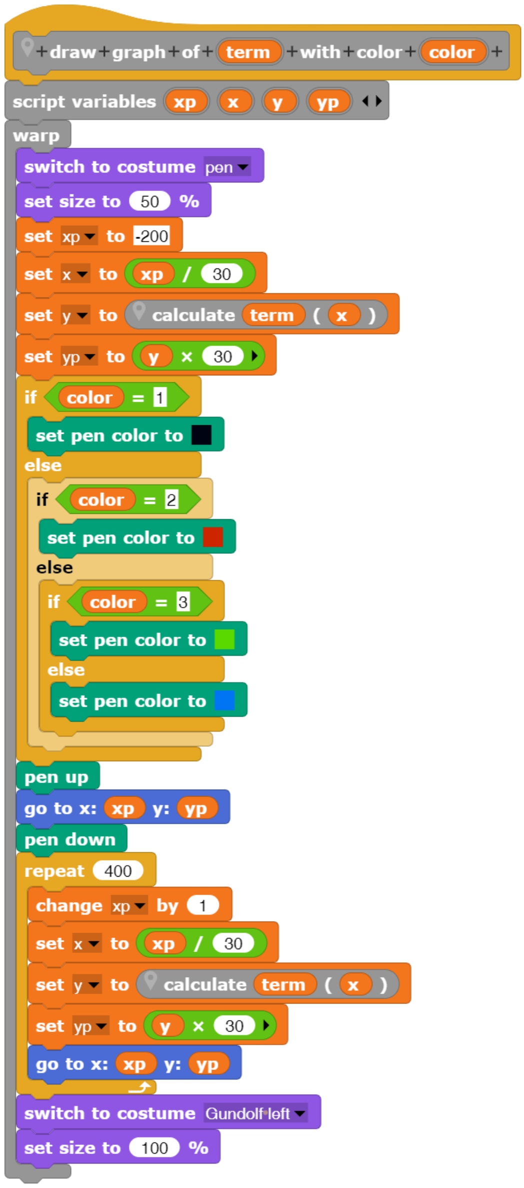
14.4 Calculate Function Values and Draw Graphs

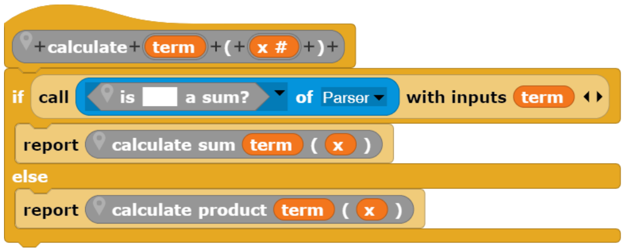
If we can parse function values, then of course we can calculate them. The procedure is quite similar to parsing, and it is made much easier if we already know that the entered term is correct. We leave this work to Gundolf, who was actually quite useless up to now - except for the self-representation. As a mathematician he should be able to calculate!

We want to calculate function values and then draw the graphs of the function and its first derivative. For this Gundolf must be able to draw at least a graph.

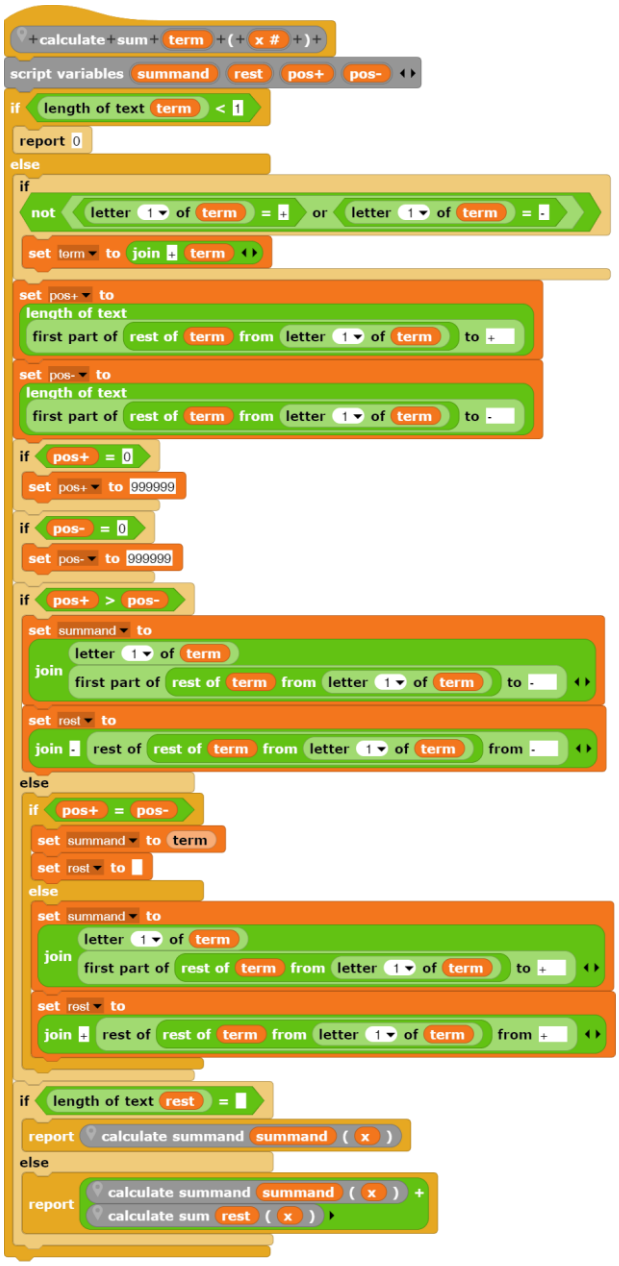


In these scripts all blocks are already present - except of one. The calculation of a function term at the position x is still missing. We give the corresponding scripts without comments because they are very similar to those of the parser.

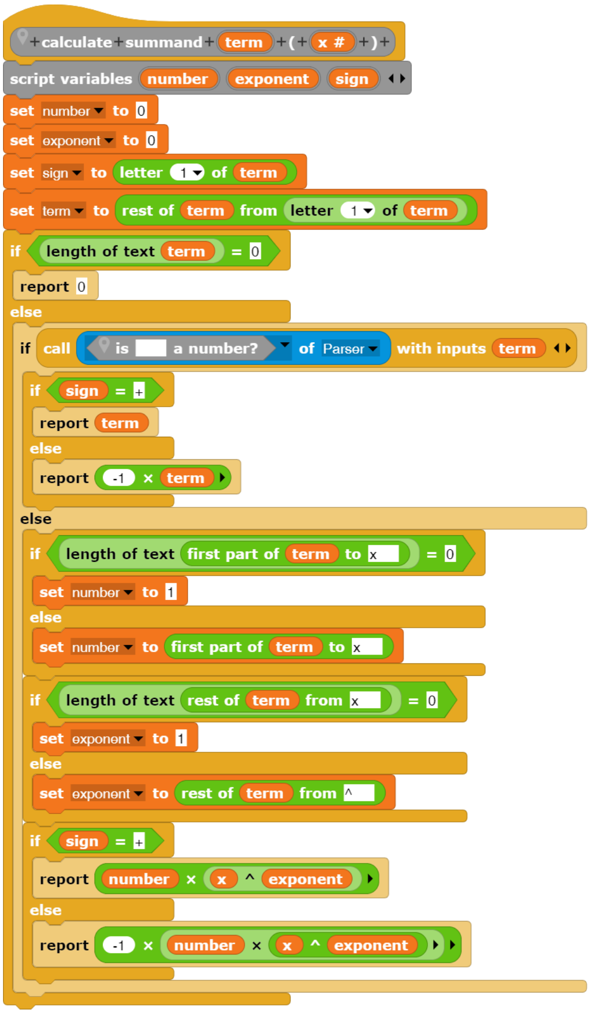




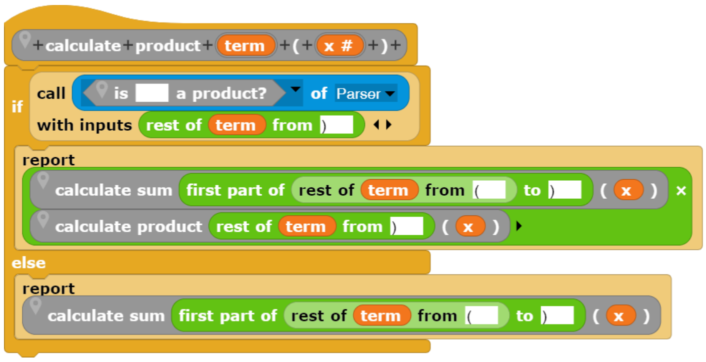
+calculate+ term +(+ x # +)+
if call is a sum? of Parser with inputs term
report calculate sum term (x)
else
report calculate product term (x)



+calculate+sum+ term +(+ x # +)+
script variables summand rest pos+ pos-
if length of text term < 1
report 0
else
if
not letter 1 of term = + or letter 1 of term = -
set term to join + term
set pos+ to
length of text
first part of rest of term from letter 1 of term to +
set pos- to
length of text
first part of rest of term from letter 1 of term to -
if pos+ = 0
set pos+ to 999999
if pos- = 0
set pos- to 999999
if pos+ > pos-
set summand to
join letter 1 of term
first part of rest of term from letter 1 of term to -
set rest to
join - rest of rest of term from letter 1 of term from -
else
if pos+ = pos-
set summand to term
set rest to
else
set summand to
join letter 1 of term
first part of rest of term from letter 1 of term to +
set rest to
join + rest of rest of term from letter 1 of term from +
if length of text rest =
report calculate summand summand (x)
else
report calculate summand summand (x) +
calculate sum rest (x)

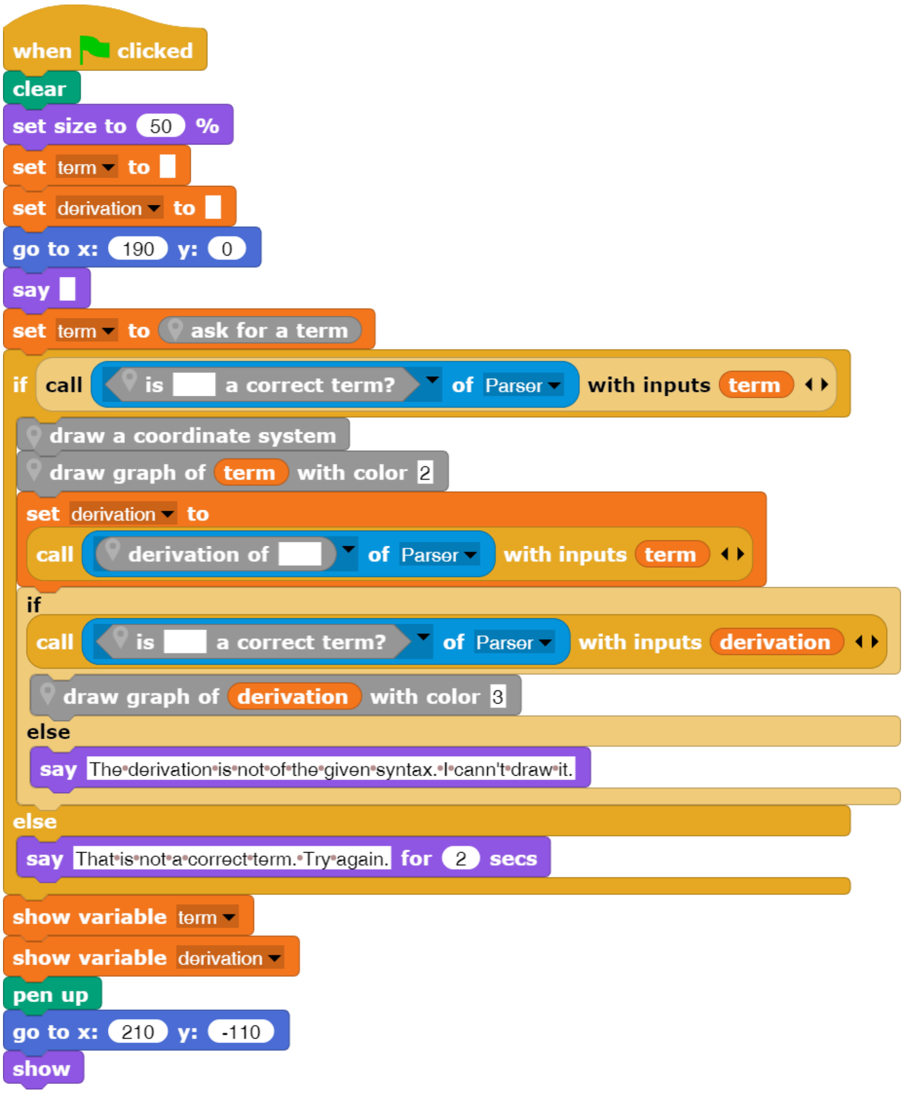


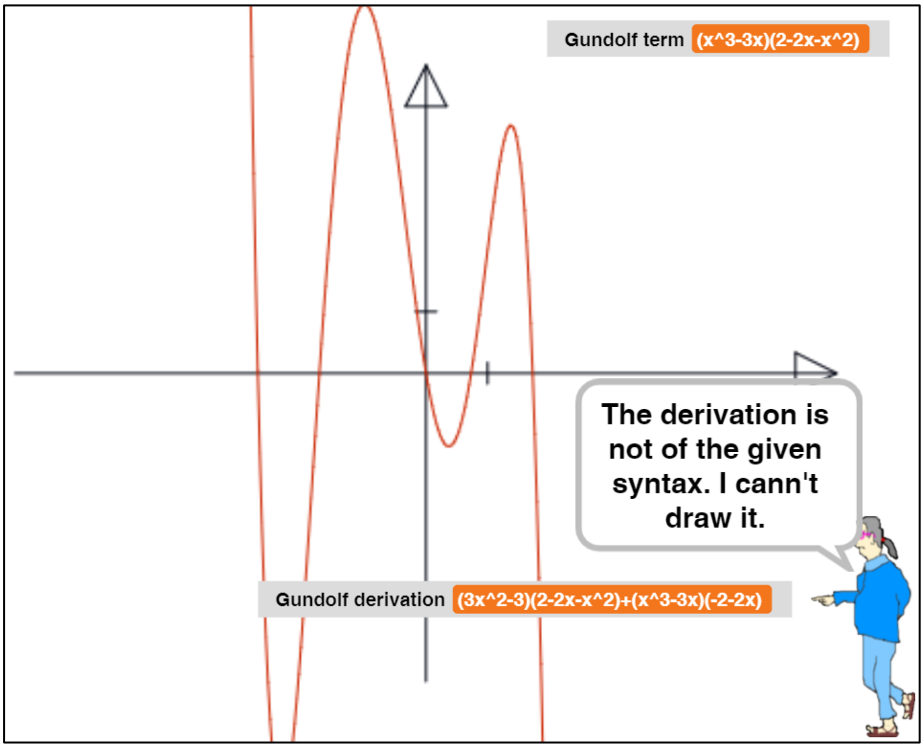
+calculate+summand+ term +(+ x # +)+
script variables number exponent sign
set number to 0
set exponent to 0
set sign to letter 1 of term
set term to rest of term from letter 1 of term
if length of text term = 0
report 0
else
if call is a number? of Parser with inputs term
if sign = +
report term
else
report -1 × term
else
if length of text first part of term to x = 0
set number to 1
else
set number to first part of term to x
if length of text rest of term from x = 0
set exponent to 1
else
set exponent to rest of term from ^
if sign = +
report number × x ^ exponent
else
report -1 × number × x ^ exponent



+calculate+product+ term +(+ x # +)+
if call is a product? of Parser
with inputs rest of term from)
report
calculate sum first part of rest of term from (to) (x) ×
calculate product rest of term from) (x)
else
report
calculate sum first part of rest of term from (to) (x)

With their help Gundolf can shine now:





14.5 Tasks

1. a: Make the outputs a little more **readable**.
   b: Combine results in the derivation so that they correspond to the given syntax and the graph can be drawn.

2. a: Define **signed numbers** and change the processing of the terms accordingly.
   b: Proceed accordingly for **floating point numbers** (numbers with decimal points).

3. a: Define **advanced function terms**, which can contain quotients, using syntax diagrams.
   b: Enable parsing of these function terms by writing appropriate predicates.
   c: Form derivatives by implementing the quotient rule as a string operation.

4. Perform task 3 accordingly for **trigonometric functions**.

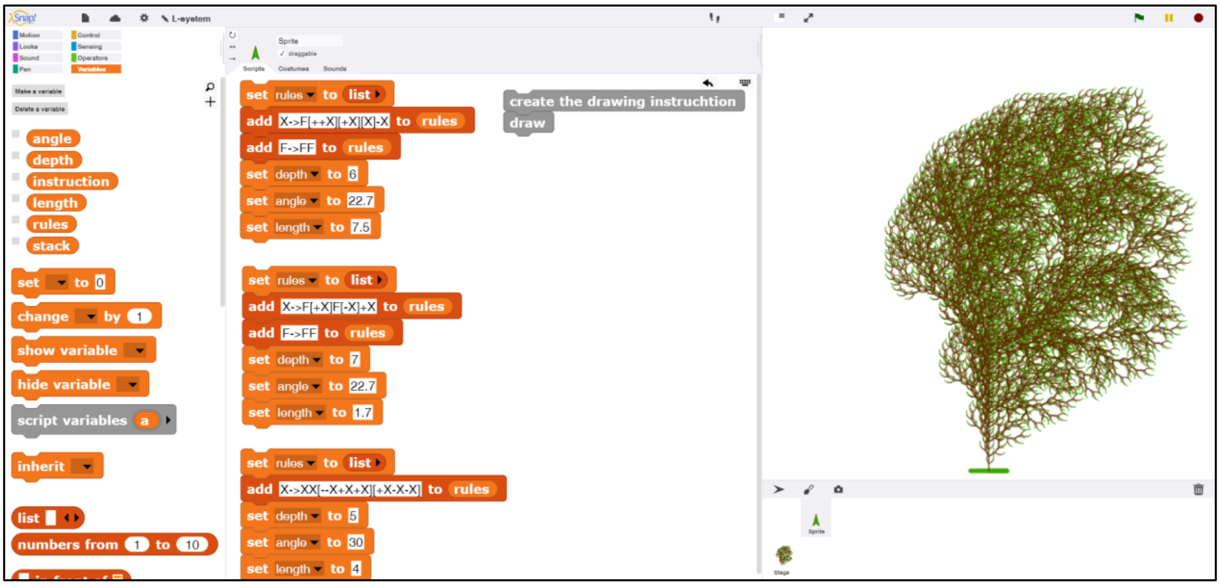
5. Allow function terms that require the use of the **chain rule**. Implement appropriate predicates and string functions.

6. a: Let the graphs of the **other function types** draw after they have been parsed.
   b: Allow a selection of the graphs to be drawn (function, first and second derivation).

7. Introduce a "**function calculator**": a function term is entered first. If this is correct, values can be entered repeatedly, and the corresponding values are determined.

15 Artificial Plants: L-Systems

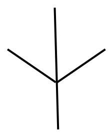
Level: *high school* Materials: *L-systems*



15.1 L-Systems

In systems according to Aristid Lindenmayer[69], plants are described by a rule system that generates the drawing instruction for a *turtle* from an *axiom*, a first character, by substitutions. One can imagine it in such a way that - starting from a shoot - the plant is drawn up to the next branching. Its position is stored on a *stack*, then the branches are described one after the other, returning to the branching after each branch. The turtle can only move forwards (*F*) and rotate through a fixed angle (+ and -). Saving the turtle position and direction and restoring this state is symbolized by square brackets (*[* and *]*). A simple plant with a triple branching can be described by

Axiom: *X* Rule: *X* → *F[-X][+X]FX*

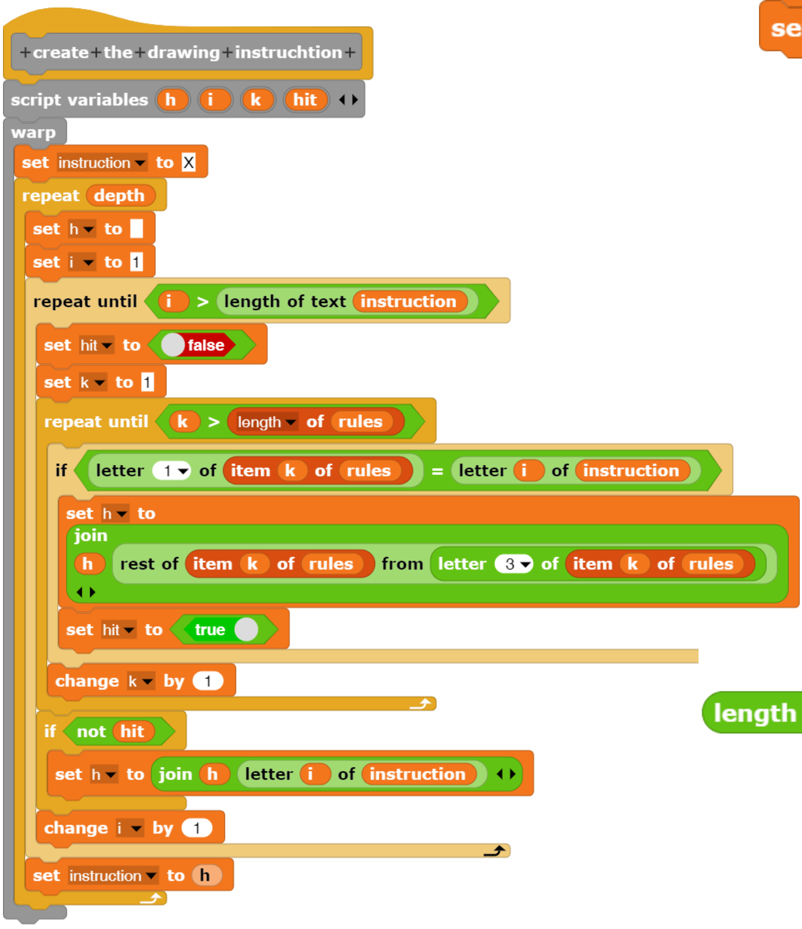


If this rule is applied several times, then the plant can grow at the positions where an *X* has been inserted. So that the older parts of the plant grow along, a rule *F* → *FF* is often inserted.

[69] https://de.wikipedia.org/wiki/Lindenmayer-System

15.2 Create the Drawing Instruction

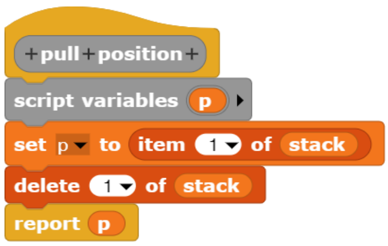
First of all, we need a rule system, i.e. a list variable *rules*, to which the desired rules are added line by line as character strings. The character to be replaced is at the very beginning, followed by "->" and the replacement starting with character *4*. The *recursion depth*, the given *angle*, and the *length* of the line length (for *F*) are also assigned to variables.



When generating the drawing instruction, we start with the *axiom X*. Then we create an auxiliary string *h* in which the replacements are performed per pass: whenever a character to be replaced is found in the old drawing instruction, we append the substitution to *h*. Finally, *h* replaces the drawing instruction, and the next replacement pass is started. The result can become quite long!

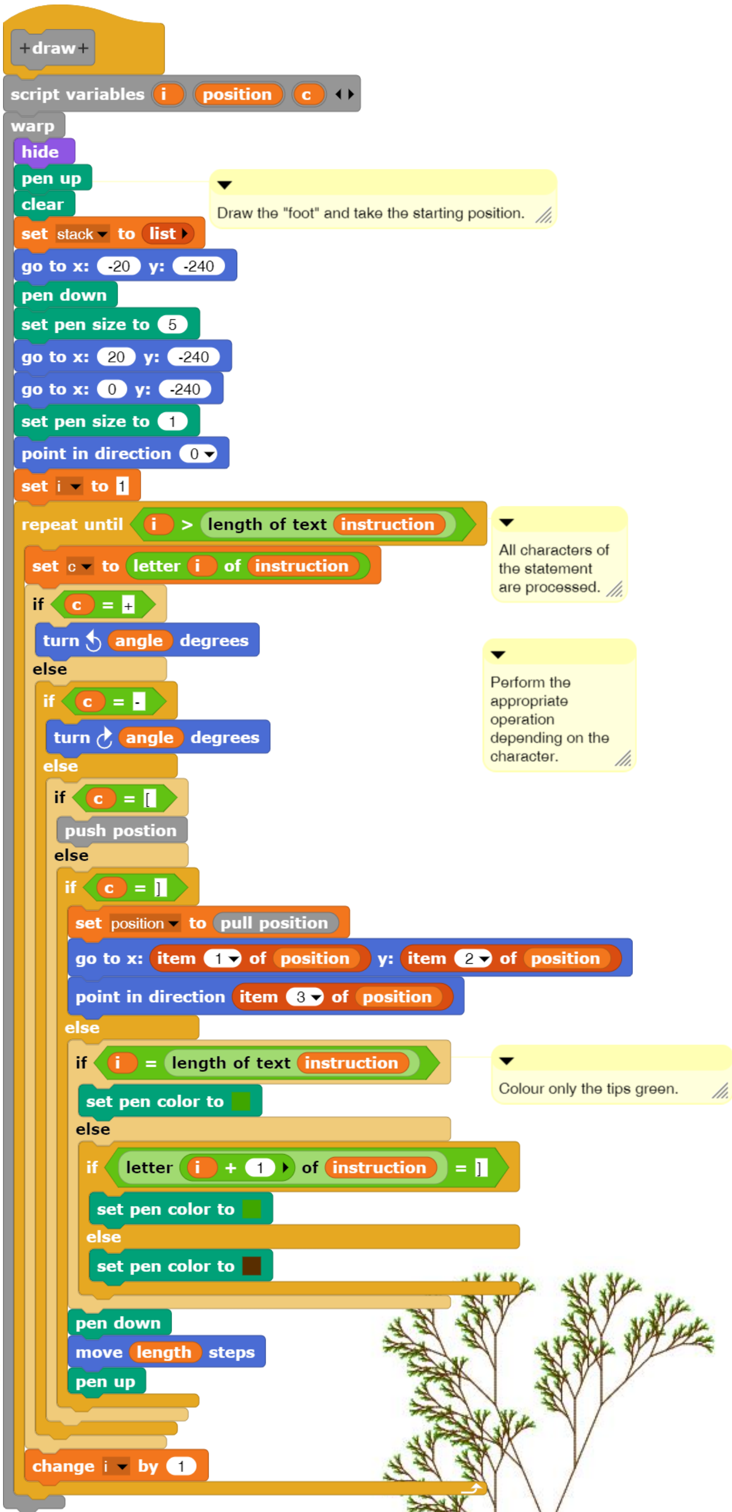
15.3 The Stack Operations

As a stack for storing the turtle positions we use a simple list. Operations are executed only at the beginning of the list - already we have a *stack*. The storing is usually done by an operation *push*. We store a three-element list with *x*- and *y-position* and the current *direction*. By *pull* the last stored position is retrieved and removed from the list.

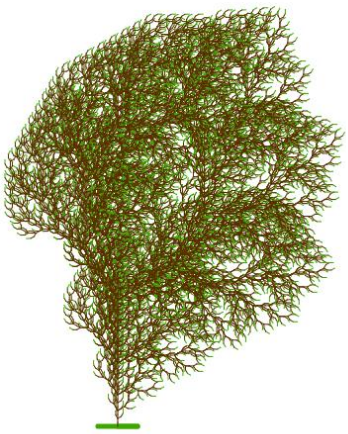


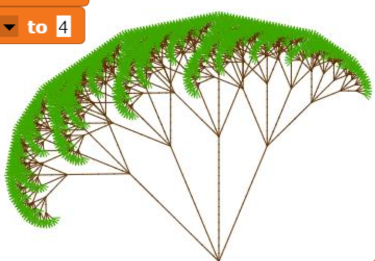
15.4 Drawing the Plants

Drawing the plants is very easy, since all our sprites can be used as turtles. We enlarge our stage to 500x500 (select *stage size...* in the settings menu) and let the Turtle draw the "foot" on which the plant grows. We then step through the string of drawing instructions one character at a time, performing the appropriate Turtle operation or stack operation for each character. As a little gimmick, we draw the "tips" of the plant in green. *(Tips can be recognized by the fact that the next step is to go back to the last turtle position, i.e. a pull operation follows).*



Examples:





15.5 Tasks

1. a: Search the web for **grammars** for L-systems. Create the appropriate plants.
   b: Select a plant species, e.g. a certain tree species, and study its construction thoroughly using pictures. Pay particular attention to growth areas. Then describe their structure using an L-system grammar. Check the result using the program.

2. a: Why are the grammars considered so far "**context-free**"? What does this mean for the plants produced??
   b: Check the web to see if grammars other than context-free are used to describe artificial plants. If yes: why actually?

3. a: In the program the tips of the branches (as "leaves") were dyed green. Replace these green pieces with more beautiful **leaves**.
   b: Transfer the principle to drawing the thickness of the branches. Just come up with something! 😉

4. Plants don't always grow the same: there are storms, raging children, hobby gardeners, weather disasters, ... Bring some **randomness** into play to produce differently shaped plants of the same type.

5. a: The stack operations were always performed at the top of the list. Could one also take the end? If yes: why?
   b: Would something change if you insert at the beginning of the stack and remove the positions at the end? If yes: why?

6. The users of the L-system program can enter anything else as grammar. Check their entries with a **parser** before trying to create the plant.

7. a: How would the rules for L-systems be changed if we wanted to create **three-dimensional plants**? What did this mean for the drawing of the plants? Are there turtles for three-dimensional drawing?

   b: Find out about topics where artificial plants are used on the net.

8. Do they also draw artificial animals? Artificial people? If yes: where? How do they do that?

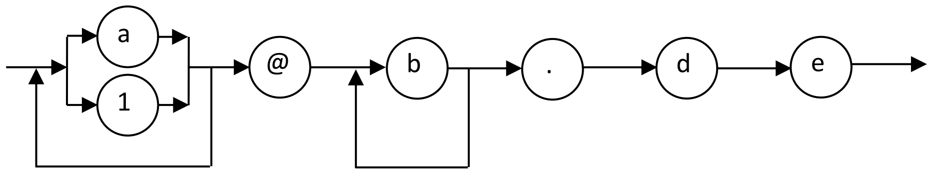
16 Automata

16.1 Correct Mail Addresses

Level: *from middle school* Materials: *Email addresses*

We want to use a finite automaton to check whether a mail address is correct. To do this, of course, we first need to know what "correct" means. We give a syntax diagram:

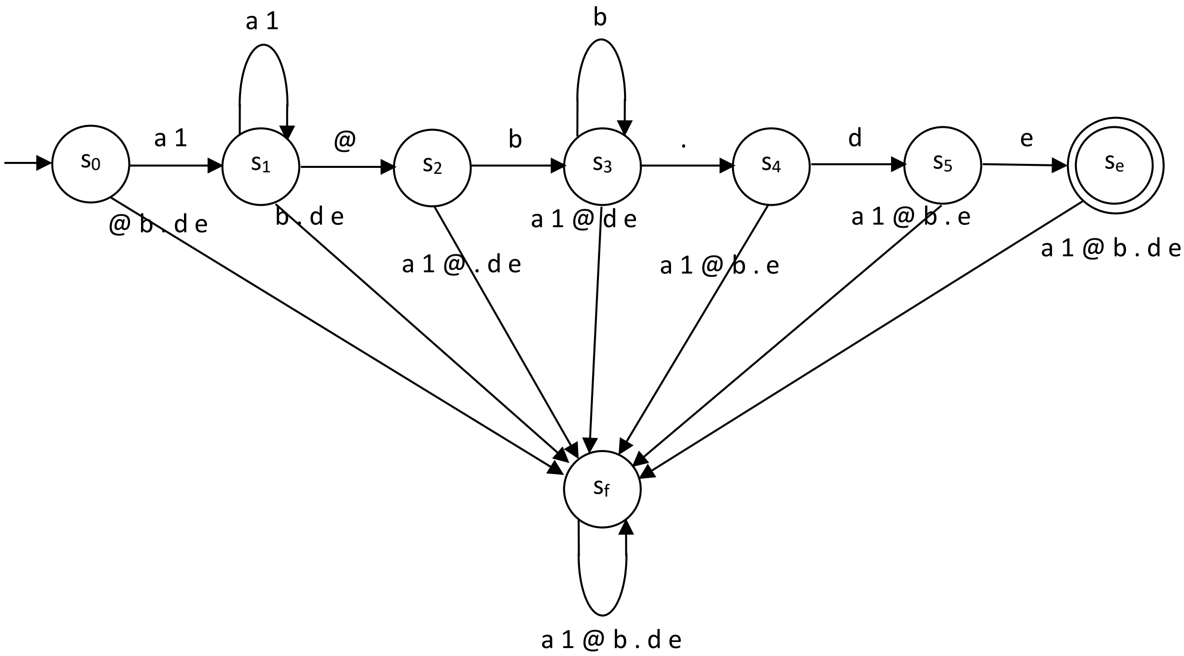
<u>Mail address</u>:



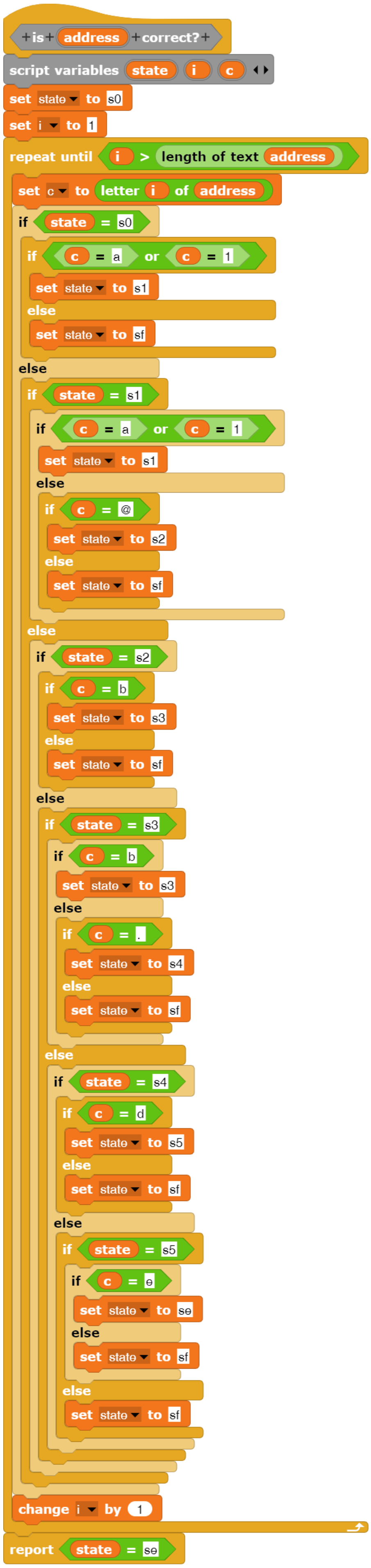
In this simplified form, the usernames thus consist of the characters *a* and *1* (as substitutes for letters and special characters) in mixed order, then the usual @ follows. The mail server name consists only of *b*s, and - separated by the dot - *de* follows as the domain name.

Correct email addresses are e.g. *a@b.de a1a@bbb.de*, wrong would be e.g. *1@c.com*.

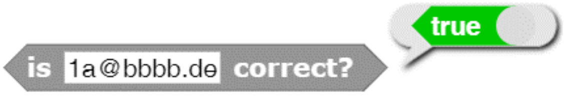
Translated into a finite automaton, we obtain its state diagram (the input characters are enumerated on the edges of the graph with spaces as separators):



The translation into a *Snap!* script can well be done as a *predicate* because the response of the automaton is *true* (the final state s_e was reached) or *false* (another state was reached, typically the error state s_f). In the script the checked address is stepped through character by character. Starting from the initial state s_0, it is checked whether the current character is valid. If this is the case, then the script switches to the next state specified in the state diagram, otherwise it switches to the error state. The script is quite long, but consists only of nested alternatives, which are a direct translation of the state diagram.

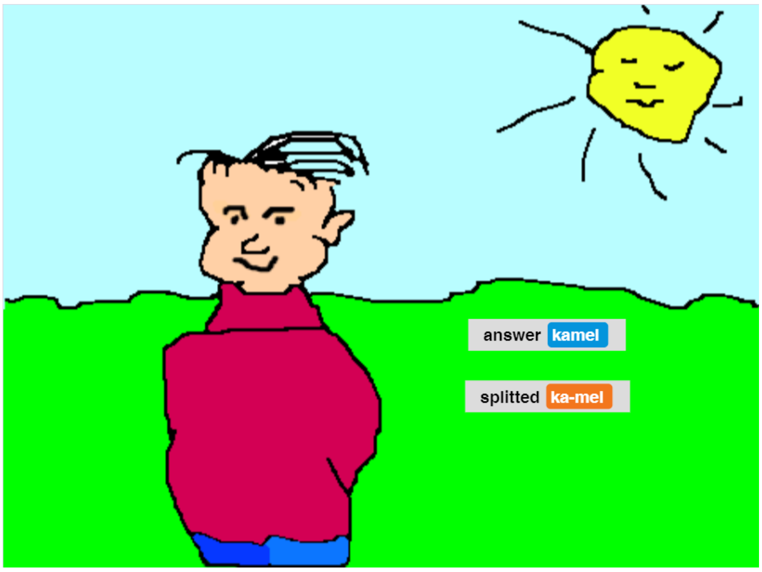


When checking the mail addresses, the created predicate can then be used.

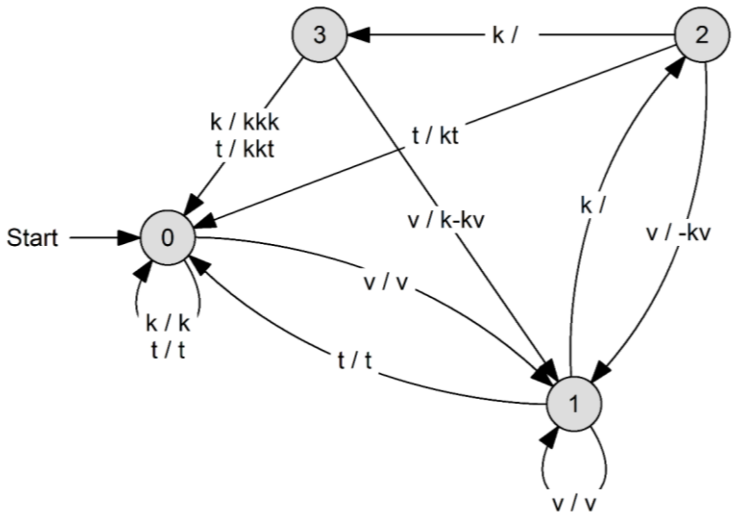


16.2 Hyphenation: Kevin Speaks[70]

Level: *high school* Materials: *Kevin speaks*



Mealy automata can be used to implement simple hyphenation, which works surprisingly well. In addition, we want to get the sprite *Kevin* to pronounce the words we input. The second sounds harder than it is: if we have the syllables, then for each syllable we can create an image with the mouth position whose name corresponds to the syllable (e.g. *AU.png*) and record the spoken syllable to it (e.g. as *AU.wav*). We drag these files to the *Costumes* and *Sounds* areas of *Snap!* and call them from there.

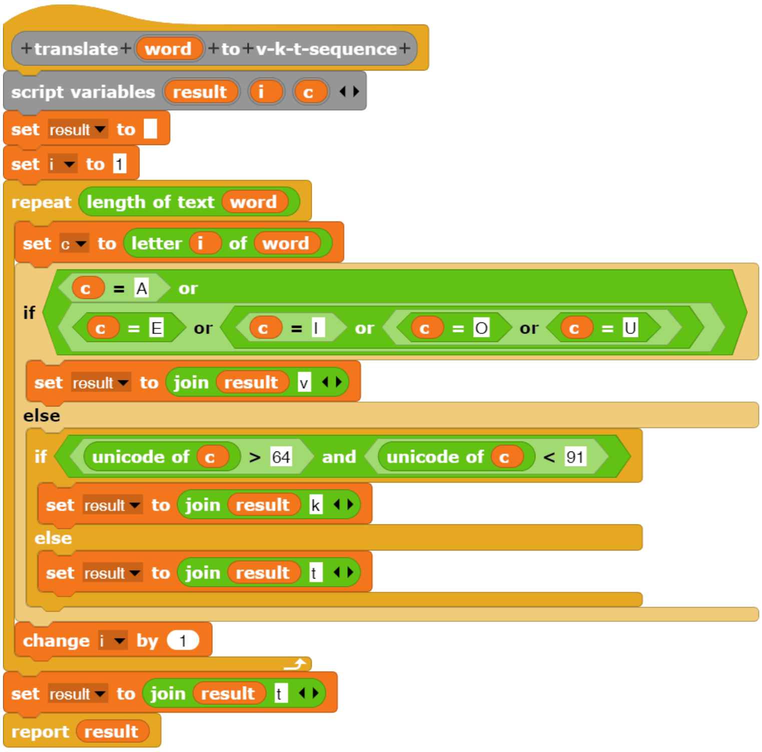


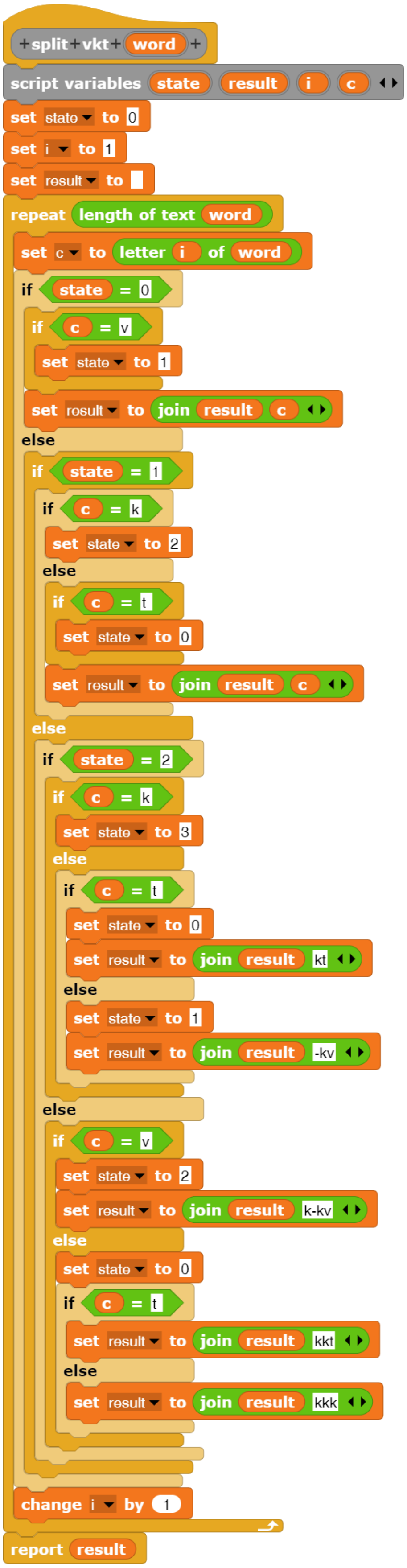
We start from the very simple Mealy automaton shown here. Its input alphabet consists of vowels (*v*), consonants (*k*) and other separators (*t*). It inserts some hyphenation characters, but of course it works incompletely and partly wrong. It separates the character strings *vkv* into *v-kv* and *vkkv* into *vk-kv*.

Using *ask and wait*, we enter words, which will then be hyphenated. Since users of programs never follow the guidelines, we first make sure that only uppercase letters appear in the word. To do this, we must be able to convert at least a single character to uppercase if necessary. We have already written the function for this in the Vigenère encoding, as well as the one for the conversion of whole words.

A word transformed into uppercase can be similarly transformed into a sequence of the characters *v*, *k* and *t*. The vowels are very easy to find, the consonants are letters that are not vowels, and the rest are treated as separators. For practical reasons, a *t* character is added last. Thus, there is always at least one character - and we always reach the state *0* last in the automaton.

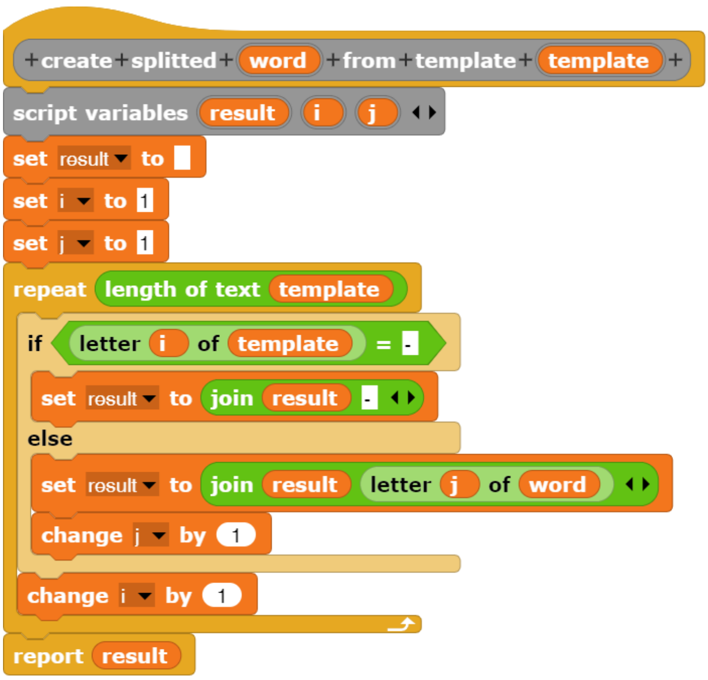
[70] Based on an idea of Wilfrid Herget.



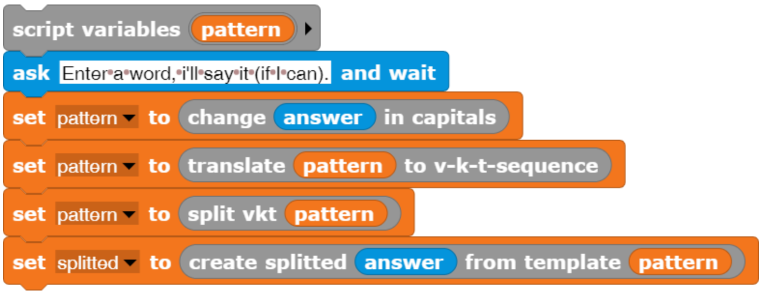


Now we hyphenate. We read character by character of the sequence from the characters *v*, *k* and *t* and write down our automaton: Depending on the state, we specify which next state is taken and which characters are output.

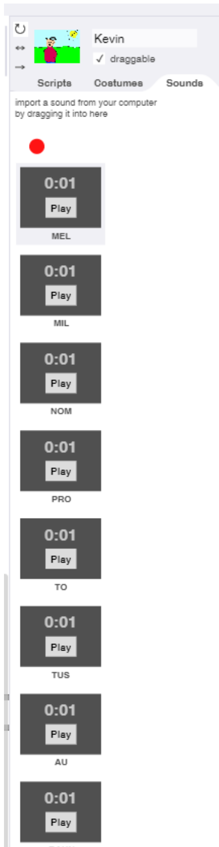
Finally, we have to transform the *vkt* sequence back into the original characters - with the separators in between. To do this, we run through the *vkt* sequence with the separators (index: *i*) as a *template* and build the result sequence from the characters of the entered word (index *j*). However, we change *j* only if *i* does not point to a separator (-) in the pattern.



We can now use these blocks step by step to hyphenate a word:



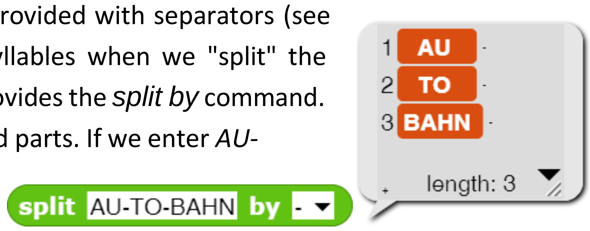
And of course, we can combine such sequences of statements in a new block.



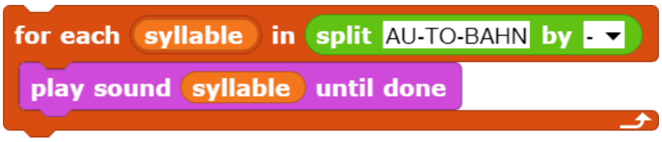
The words broken down into syllables are to be pronounced by the computer, similar to navigation systems, automatic time announcements or other "computer voices". If we store syllables instead of whole words, we need much less memory, because the syllables can be used several times. (But it does not make it more beautiful!).

First, we choose some words: e.g. *autobahn, autonomous, automaton, pronoun, pro-mille, camomile, camel, cactus*. We record their syllables in the recorder of the *Sounds* section and change their name to the syllable's name in capital letters.

Since the entered words were provided with separators (see above), we get (roughly) the syllables when we "split" the word. For this purpose, *Snap!* provides the *split by* command. The block generates a list of word parts. If we enter *AU-TO-BAHN* and hyphenate at the "-" character, we get:



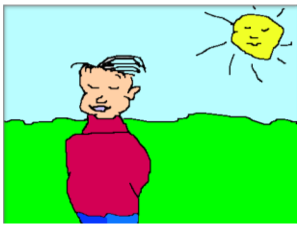
Since our sound files are named the same as the syllables, we can play them with *play sound until done* by choosing the syllable as input parameter of the block.

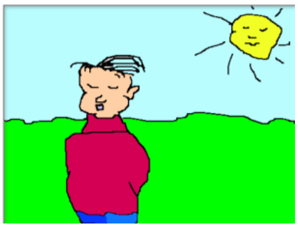


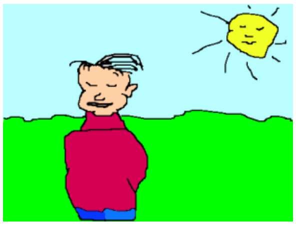
So, we can make the computer speak out words by

- hyphenate the word entered,
- and break it down into its syllables,
- and have the syllables of this list "pronounced" one after the other.

For each of the different syllables we draw a costume for Kevin.


AU


TO


BAHN

We display these costumes while speaking the syllables.

Words are pronounced by calling this script with the corresponding syllables.

16.3 Coupled Turing Machines [71]

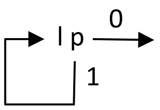
Level: *high school* Materials: *Coupled Turing adder*

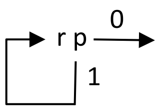
If one describes Turing machines by state graphs, then the importance assigned to this model seems to be strongly exaggerated to the learners, because the problems, which can be described by a still readable graph, are then nevertheless rather small. Much more powerful tools can be generated in the model of coupled Turing machines, where the initial state of the next machine corresponds to the final state of its predecessor. From very simple systems, increasingly powerful constructions can be created. A kind of macro-language emerges, in which topics of computability and decidability can be formulated, but above all can be experienced enactively.

Our system of elementary Turing machines works on a *Turing tape* which contains only ones and zeros. The zeros serve as separators, so that numbers are to be represented by sequences of ones. The number *n* is coded accordingly by *n+1* ones, so that also the zero has a code. In the *standard position*, the head of the Turing machine is above the one furthest to the right. All groups of ones must be separated by exactly one zero and there are two zeros at the left edge of the tape. After working, the machine is again in the standard position. From this position the next machine starts its work.

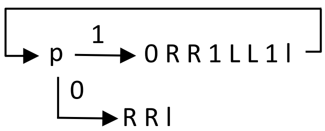
The *1-* and *0-machines* are available as *elementary machines*, which write the corresponding character on the tape at the head position. Apart from that, they do nothing. The small left machine *l* shifts the head of the Turing machine one position to the left, the small right machine *r* to the right. In addition, there is a *checking machine p*, which checks which character is present at the current head position. Depending on the result, it branches into one of two states, to which further machines can then be coupled. That was it.

Because they are often needed, we design two new machines, the *large left-hand machine L*, which runs to the left across a group of ones, and correspondingly a *large right-hand machine R*. These can be implemented as follows.:

L: 

R: 

The *copying machine K1* copies a group of ones to the right.

K1: 

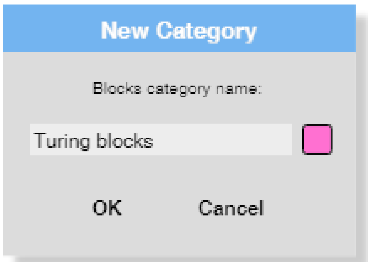
If the *copying machine K2* copies a group of ones over a second one to the right, then we can already calculate sums with the help of a *Turing adder A*.

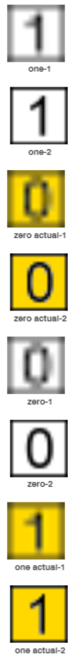
A: *K2 K2 L 1 R l 0 l 0 l*

Try it for yourself!

[71] based on Eckart Modrow, Theoretische Informatik mit Delphi, emu-online, 2005

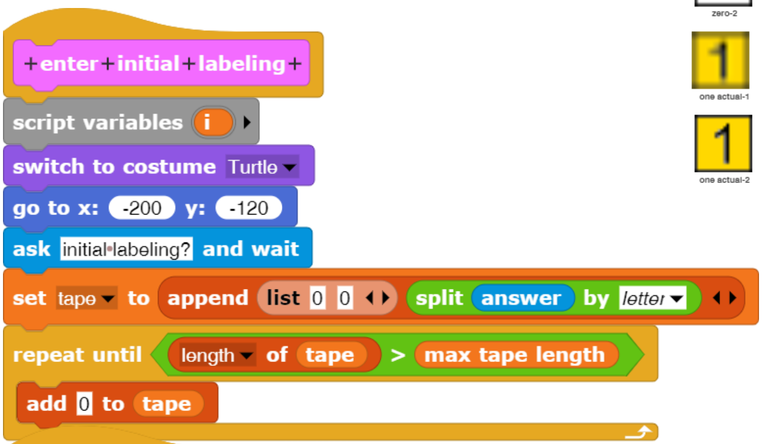
Instead of testing the machines on paper, we want to develop a macro language that can be used to generate our coupled Turing machines. Since we only want to use new *Turing blocks* to be developed, we introduce a new category for this - in tasteful pink.



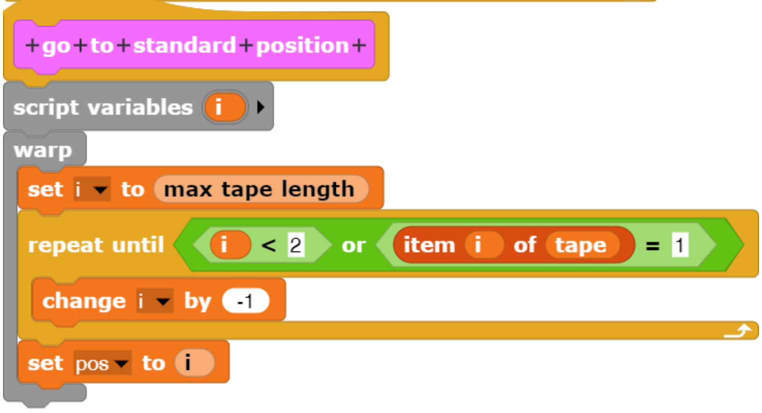


For the simulation of the machines, we need a working tape, which is built from ones and zeros. We choose a list called *tape* for this, since it can be easily changed in length. For the display, we create some images with ones and zeros of different sizes, using the yellow versions to indicate the head position (*pos*). The working speed and the cell size (*cell type*) should be changeable on the screen. Altogether we need the variables *tape*, *max tape length*, *pos*, *cell type* und *pause(ms)*.

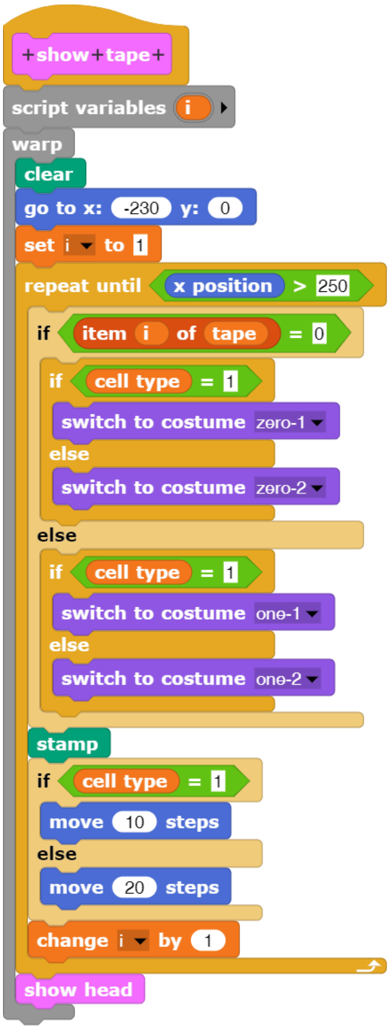
The initial caption must be requested and then a corresponding band must be generated and displayed. We do this by splitting the input string character by character into a list. In front we append the required two zeros, and we fill up the band with zeros up to the maximum length, if necessary.

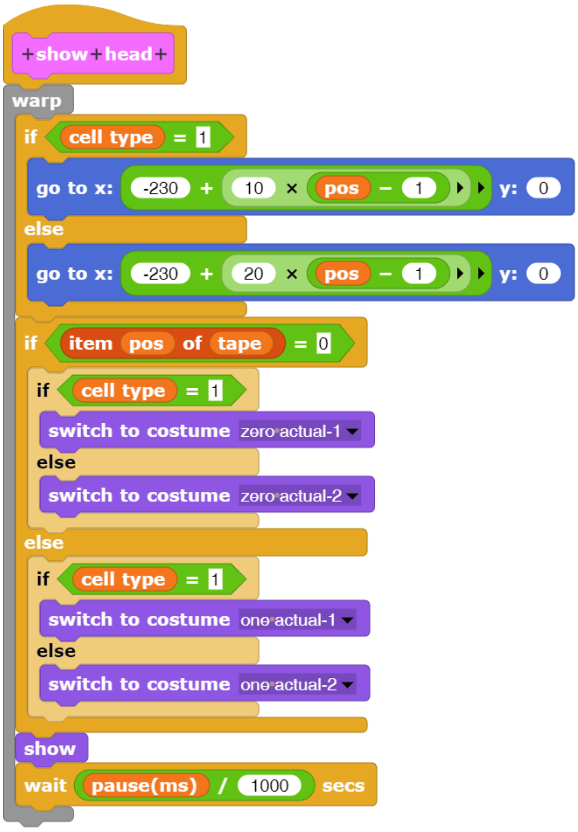


On this tape, the standard position must be taken, determining the value of the variable *pos*, which indicates the position of the head. We search, starting from the right, the first one.

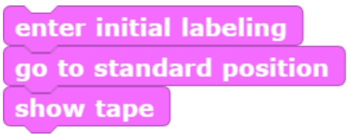


Then the tape is displayed by stamping images of the costumes side by side on the stage. To display the head position, we calculate its screen coordinates and change to one of the yellow costumes.

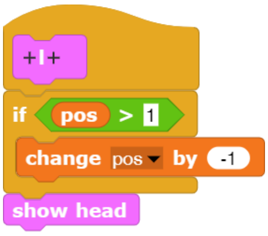


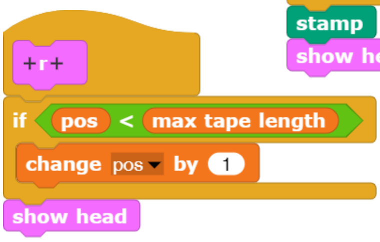


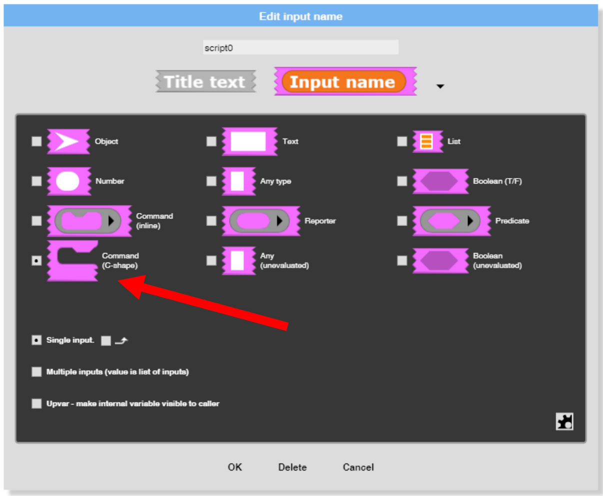
In total we get as start command sequence:



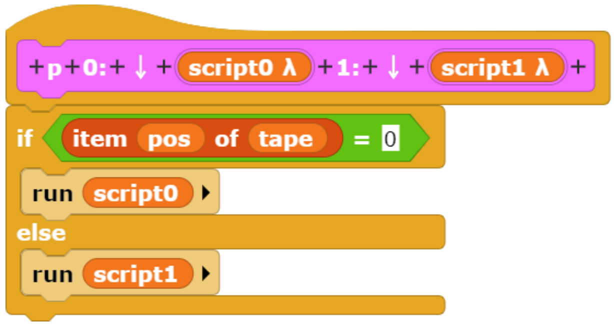
The elementary machines can now be created quickly:





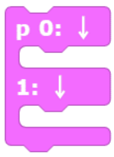


The generation of the test machine *p* is somewhat more complicated because this must be able to execute two different scripts - depending on the tape labeling. These scripts must not be evaluated BEFORE the call of the machine as parameters, but two scripts are passed, which are to be executed AFTER the call depending on the tape label. The "parameter values" are therefore scripts. When typing the parameters, we select *Command (C-shape)* to prevent evaluation. The parameters are marked as scripts by a λ.

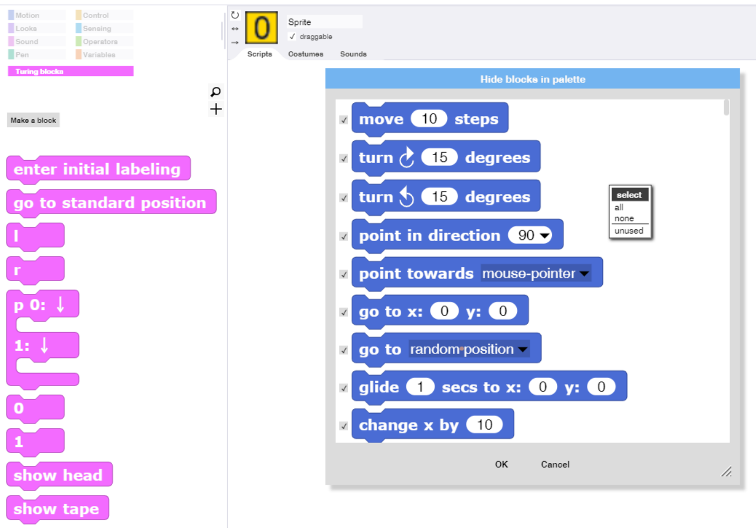


The downward arrows after the zero or the one can be found in the selection list that appears behind the small arrow after the name of "*Title text*" parameters.

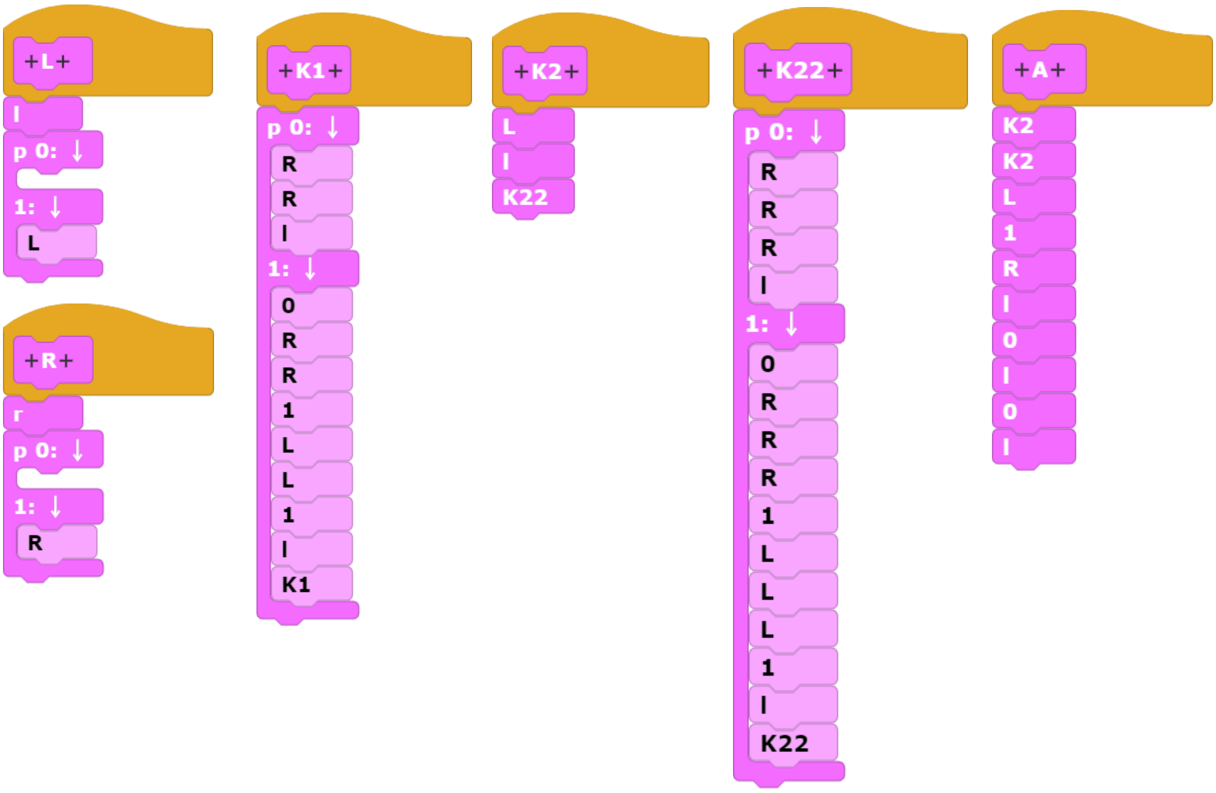
The new block, a control structure, then has the following appearance:



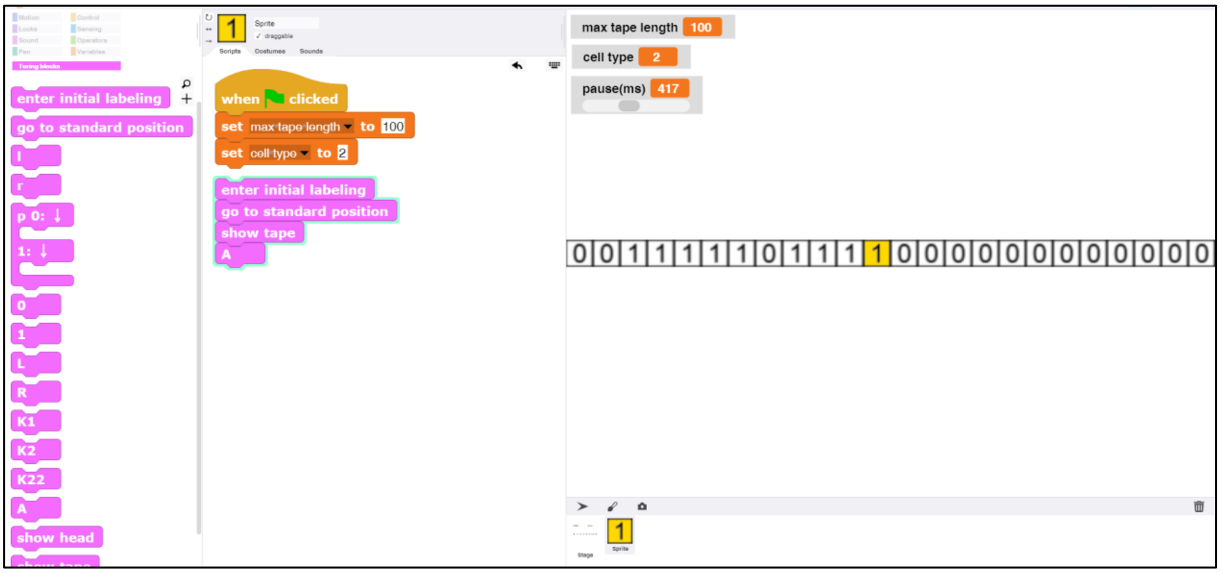
We now want to build our coupled Turing machines from these new blocks. In order not to be tempted to use the standard blocks, we right-click on a palette (*hide blocks...*) and select all blocks except our new ones. The standard palettes are then empty.



With these machines, the others can be developed "quite normally recursively" in *Snap!* as blocks.



The work of the machines can be followed on the screen at different speeds and thus checked. Afterwards they are used as new blocks for more complex problems.



16.4 Cellular Automata: Iterated Prisoner's Dilemma[72]

Level: *high school* Materials: *Cellular automaton*

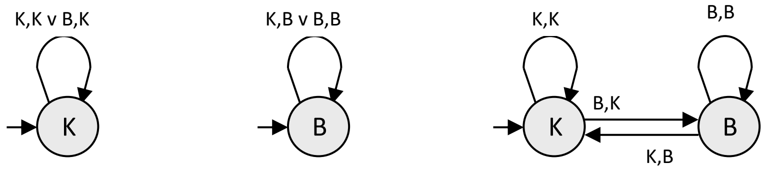
We want to build a cellular automaton based on the Prisoner's Dilemma[73], but slightly modified for trading on the Internet. The behavior of the trading partners is simulated by automata, which sit on a grid closed in both dimensions and trade with the partners within a *Von Neumann neighborhood*[74]. They exchange goods for money - as is common on the Internet. There are different types of business partners:

- *Naive* always cooperate, i.e. provide the correct equivalent value.
- *Fraudsters* never cooperate.
- *Shrewd* people cooperate at first and then react in the same way as their partner did last time.

We describe the behavior of trading partners using state diagrams:

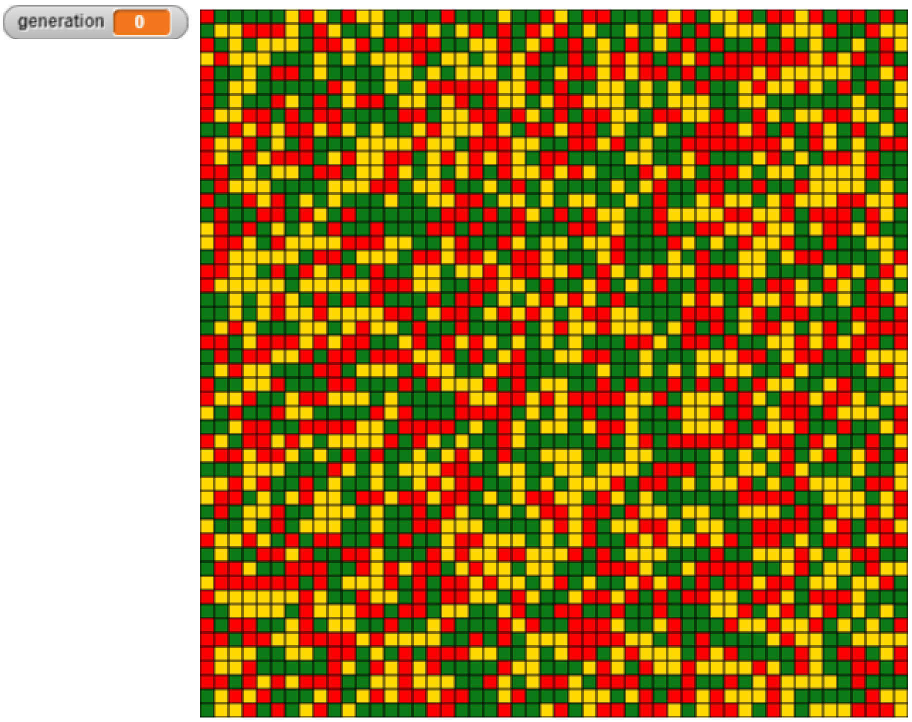
K: *cooperate*

B: *cheat*



The "*naive*" The "*fraudster*" The "*shrewd*"

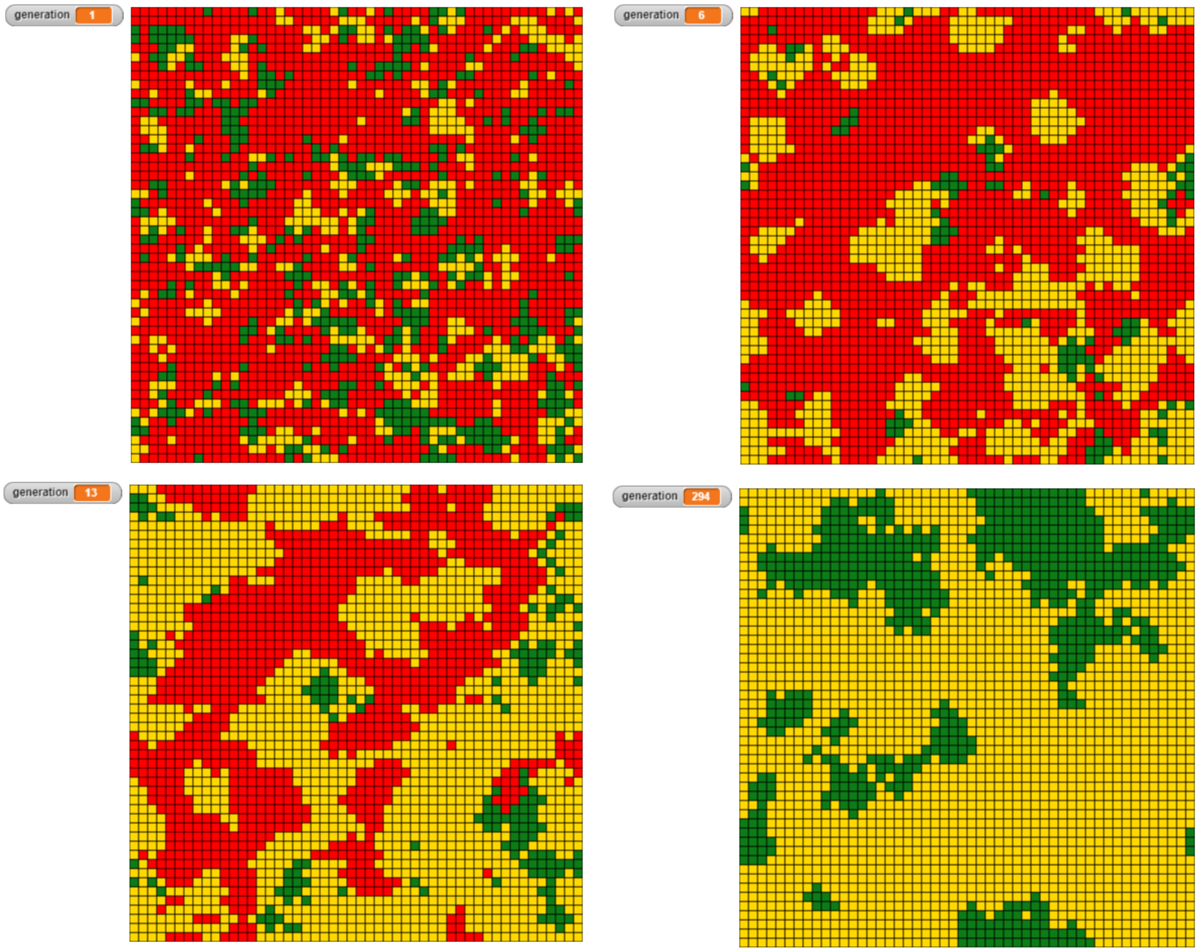
If we arrange such automatons in a grid, distribute them randomly and color them according to their state (green as "*naive*", red as "*fraudster*" or yellow as "*shrewd*"), we get an image similar to the following:



[72] based on Eckart Modrow, Zelluläre Automaten, LOG IN 127 (2004)
[73] https://de.wikipedia.org/wiki/Gefangenendilemma
[74] https://de.wikipedia.org/wiki/Von-Neumann-Nachbarschaft

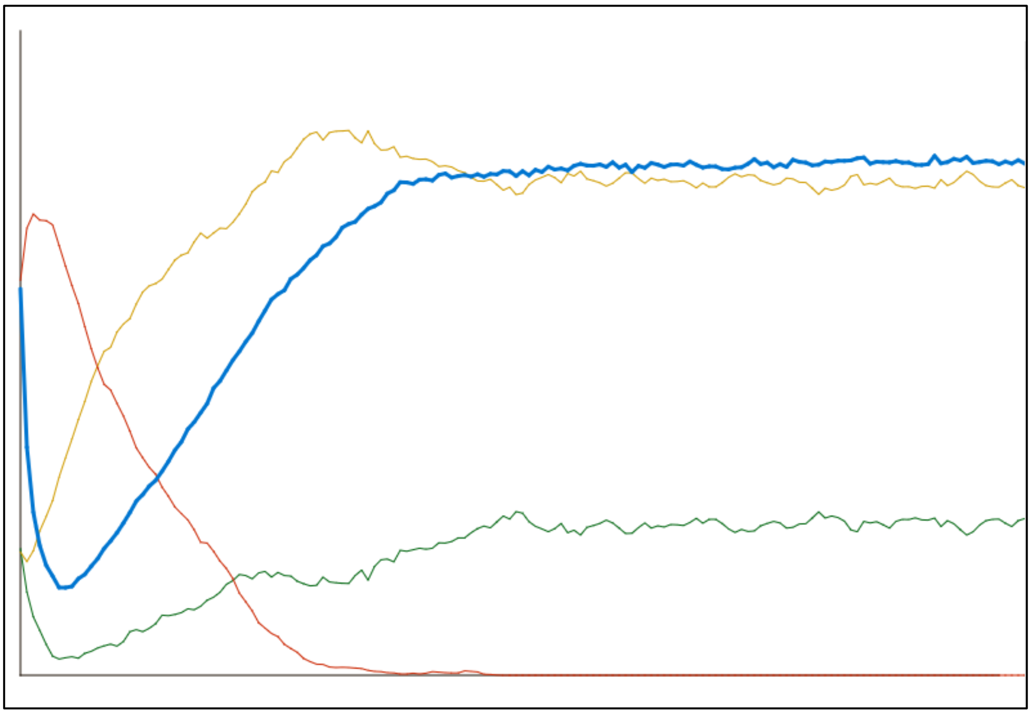
The further procedure is simple: First all partners trade once with their neighbors from the Von Neumann neighborhood, i.e. with the neighbors above, below, left and right. Afterwards all partners evaluate the success of their neighbors. As opportunists, they take over the status of the most successful neighbor or maintain their status when they were better themselves.

In the first generations, the "*fraudsters*" usually prevail. But then clusters of "*naive*" or "*shrewd*" people form and a wild "battle" begins.



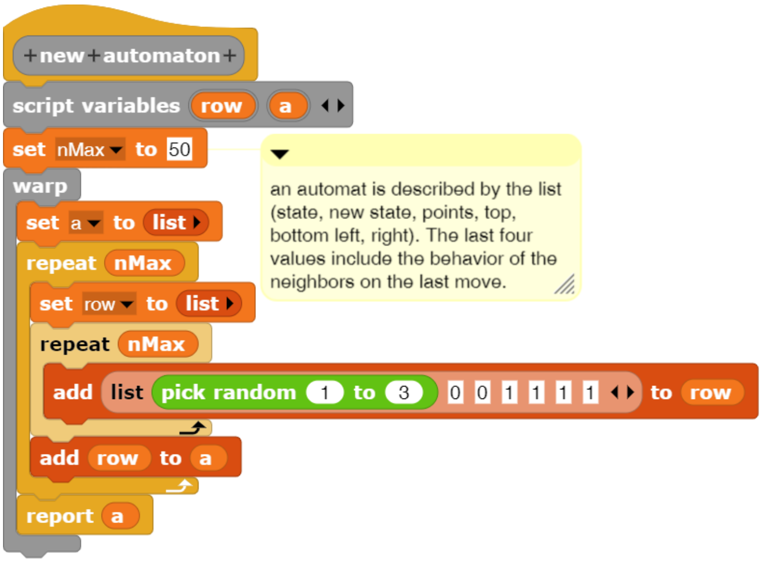
It is true that the "*naive*" are hard pressed by the "*fraudsters*". But they do quite well in groups. The "*shrewd*" usually prevail over the "*fraudsters*" - depending on the configuration - and cooperate with the "*naive*". In the end, the "*shrewd*" usually win - but not always. In groups, the "*fraudsters*" cheat each other and win nothing, while the "*shrewd ones*" assert themselves against them and are more successful with the "*naïve* behind their backs". The processes depend strongly on how the different behavior is weighted.

Global variables are suitable for evaluating the system, e.g. a "gross national product" as the sum of all trading points. Observing the sometimes surprising processes provides starting points for discussing ethical questions. Even if the example cannot, of course, be directly applied to social systems, for most people we have found a new argument for cooperative social behavior, which is not derived from transcendental or philosophical considerations, but from efficiency. It is in clear contrast to the egocentricity of primitive Darwinism, which often dominates public discussion in this respect. A diagram may serve as an example in which, on the one hand, the total numbers of the three types of automata (*naive, fraudulent, shrewd*) were plotted, and, in addition, the sum of the total trading points achieved by all types, i.e. the "gross national product", is somewhat thicker in blue. One can see very nicely that "social prosperity" (if one wants to derive this from the "trading volume") is contrary to the number of "egoists" - of course under the conditions set. Among them, fraudsters usually die out for lack of success, and the naive harmonize magnificently with the shrewd - if they are among themselves. If the behavior is weighted differently, fraudsters can be quite successful. So, it depends on the rules of the game who succeeds. You should think about them, not just in a simulation!



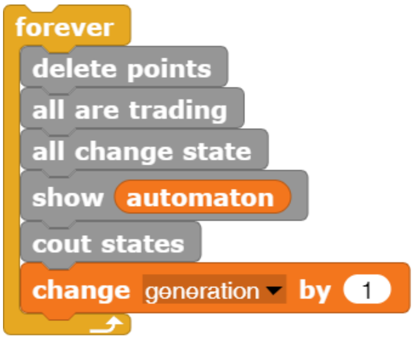
From a programming point of view, the system is rather simple, but sometimes extensive due to the change of viewing direction.

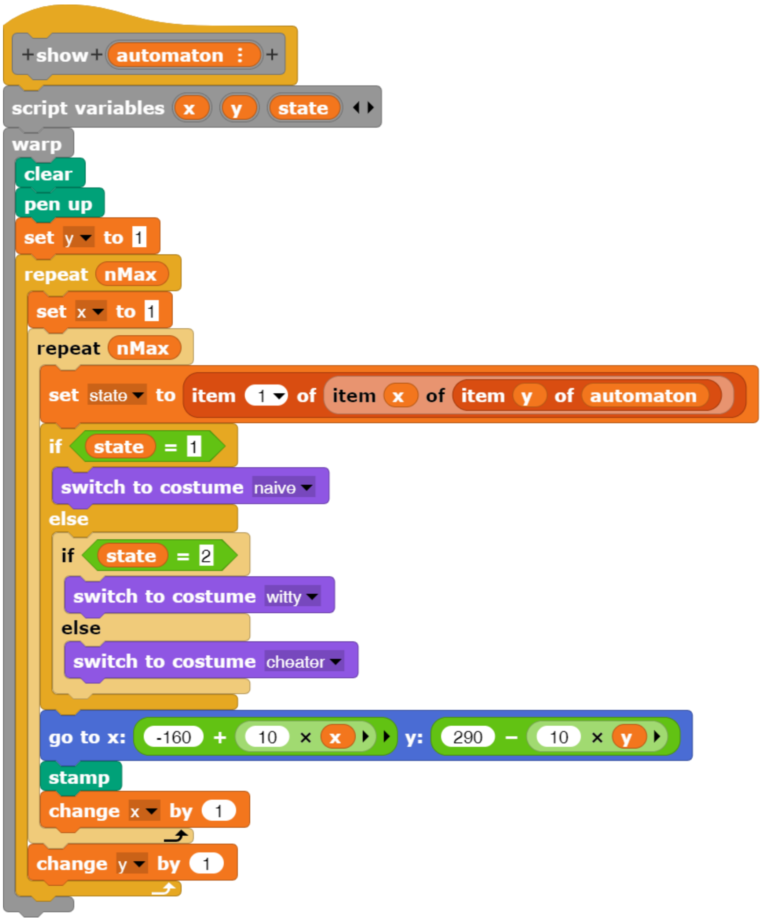
A new automaton can be described by a list of lists, whereby the automatons at the grid places correspond to sequences of numbers, which contain on the one hand their state and the reached trading points, on the other hand the "memory" about the past behavior of the neighbors.



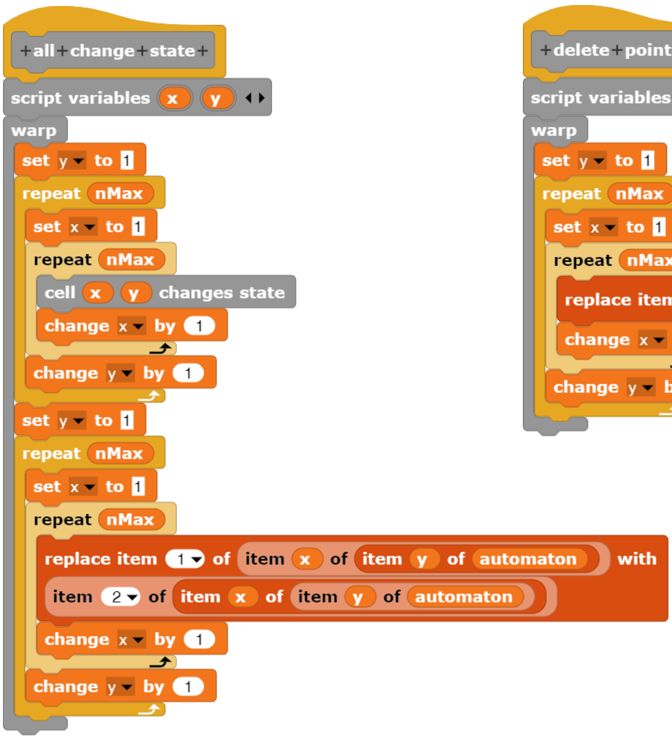
The cellular automaton can be displayed by stamping different colored costumes (small rectangles) next to each other on the work area. This has been changed to the size 800x600 pixels before.

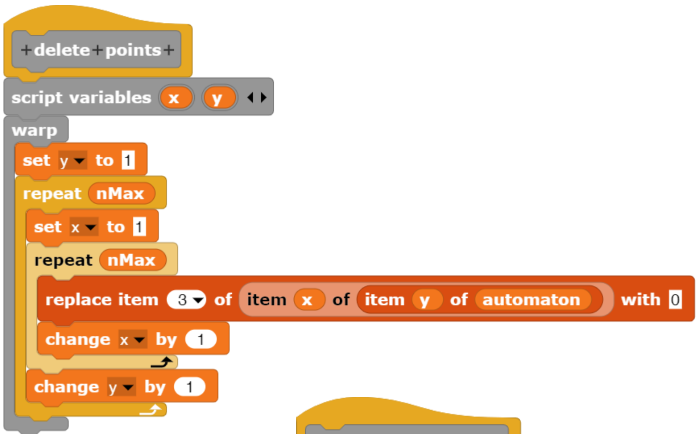
Once the machine has been created, the new generations are created from the last generation in each case.



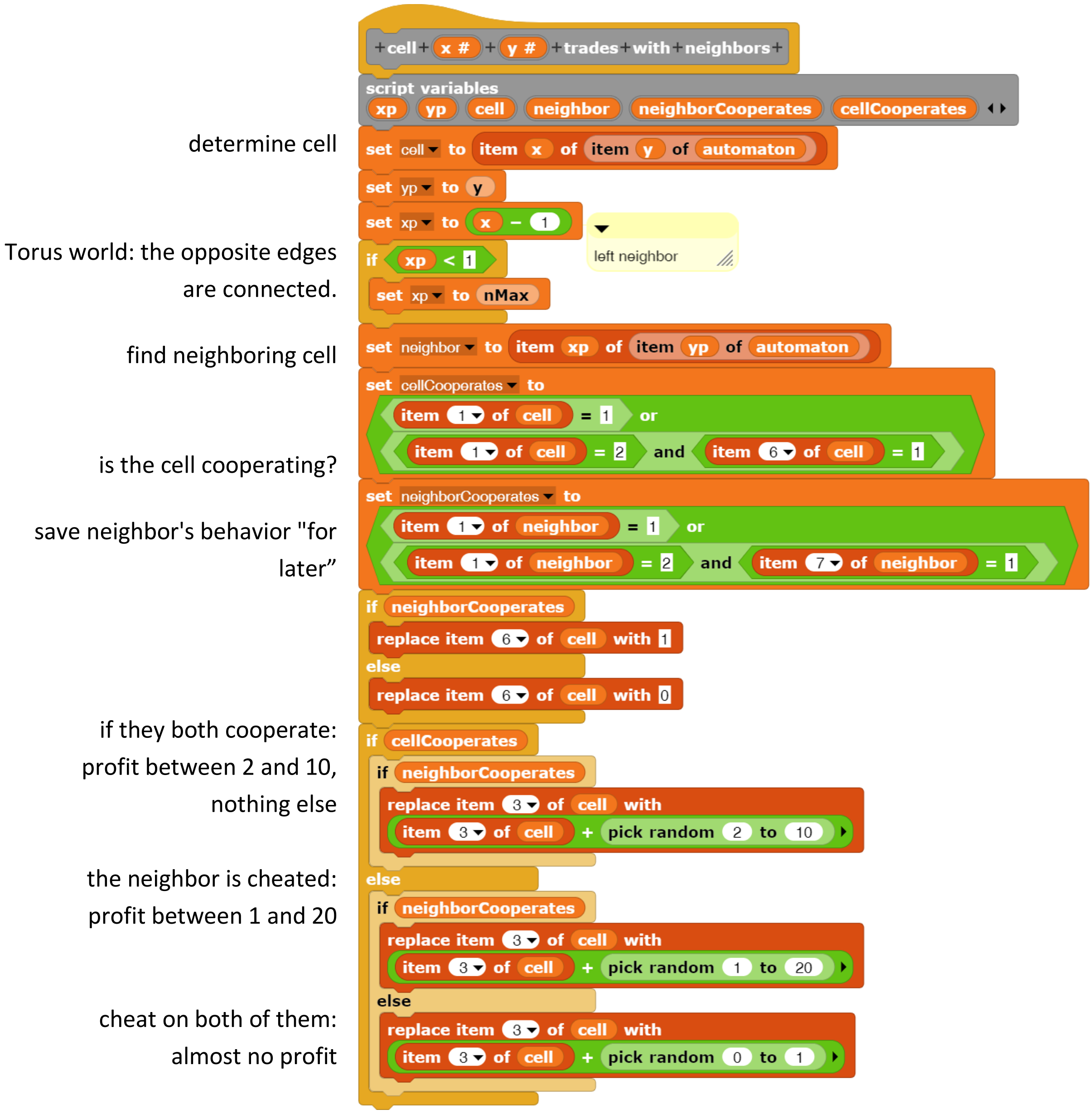


The scripts have a very similar structure: all grid locations are iterated.



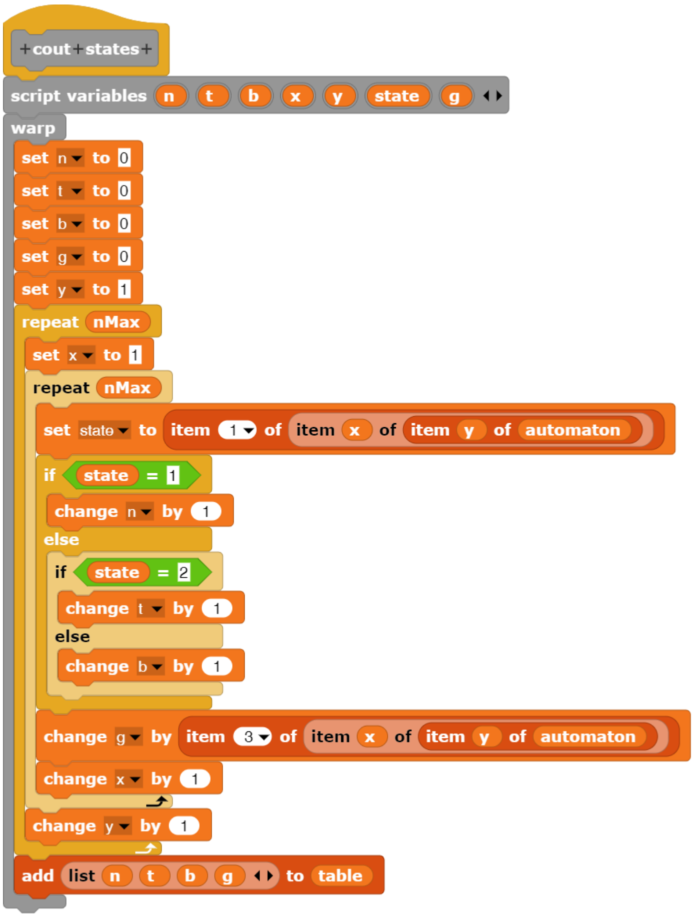


The trade of a cell with its neighbors depends on the one hand on the states of the partial machines, and on the other hand on their previous behavior. Since this data is stored in the machine values, it is easy to retrieve. Shown is the trade with the left neighbor:

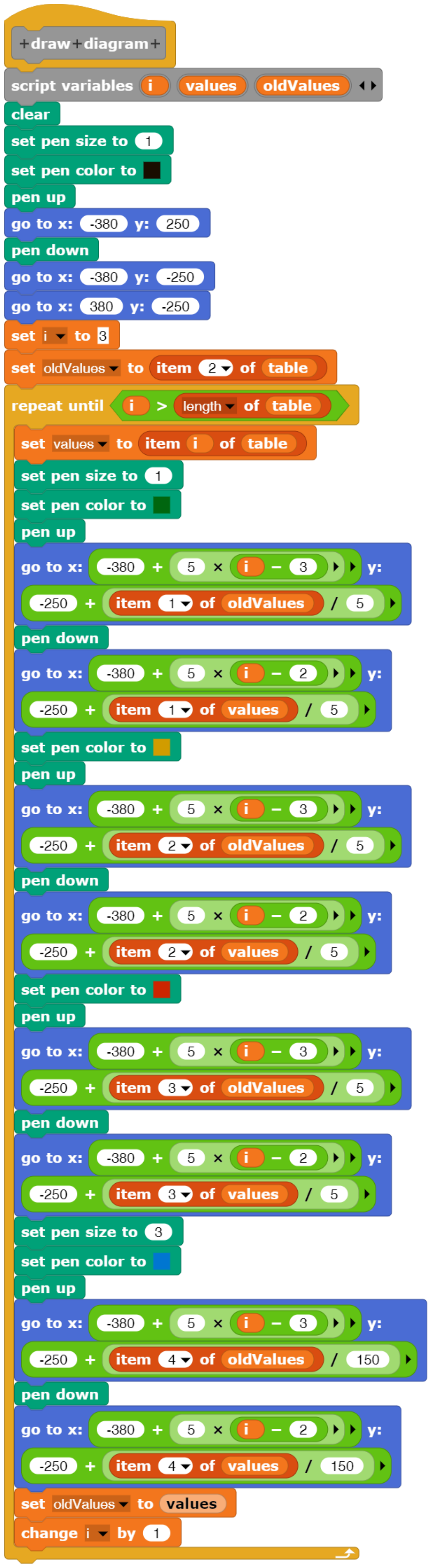


Trade with the other three neighbors is almost the same. The differences are only in the positions of the stored behavior.

Once the values of a generation have been determined, they can be counted and compiled in a list - and this results in a diagram.



34	A	B	C	D
1	Naive	TitForTat	Cheater	overall
2	489	428	1583	45677
3	320	428	1752	26541
4	243	485	1772	18849
5	177	589	1734	16254
6	161	684	1655	14581
7	130	786	1584	14178
8	125	882	1493	13819
9	103	993	1404	14482
10	119	1133	1248	14636
11	118	1281	1101	16092
12	121	1394	985	17108
13	143	1478	879	18450
14	158	1577	765	19548
15	168	1673	659	20786
16	193	1741	566	22002
17	224	1756	520	23629
18	219	1790	491	24887
19	247	1792	461	26267
20	268	1796	436	27174



16.5 Tasks

1. Develop a finite automaton as a predicate for detection
   a: **correct license plates** from three different cities.
   b: **correct IBAN numbers**. You can limit your search to a few banks.
   c: **passwords** of sufficient complexity. Define beforehand what "sufficiently complex" means.

2. Improve **hyphenation** by taking into account
   a: double consonants.
   b: typical prefixes.

3. Develop and test a **coupled Turing machine**,
   a: that copies one group of ones over another (*K2*).
   b: which pushes one group of ones to the left to another until the groups are separated only by a zero.
   c: which multiplies two natural numbers with each other.
   d: which writes a 1 after two groups of ones, if they are the same length, otherwise a zero.
   e: that subtracts two natural numbers - if that's possible. If she doesn't, she'll go crazy: she'll run away to the right.

4. a: Replace the trade of all partial automata with the neighbors "per round" by a randomly controlled process in which machines trade with neighboring (with any) partners.
   b: Replace the Von Neumann neighborhood with a **Moore neighborhood**.
   c: The machine can easily be converted to an **Ising model** by considering the machines as **spin grids**. Per round, the majority of the neighboring spins tilt the spin in the middle in their direction. There are various magnetized areas.

5. a: Find out about Stephen Wolfram's **linear cellular automata**.
   b: Implement the model.

17 Projects

17.1 LOGO for the Poor

Level: *from middle school* Materials: *LOGO for the poor*

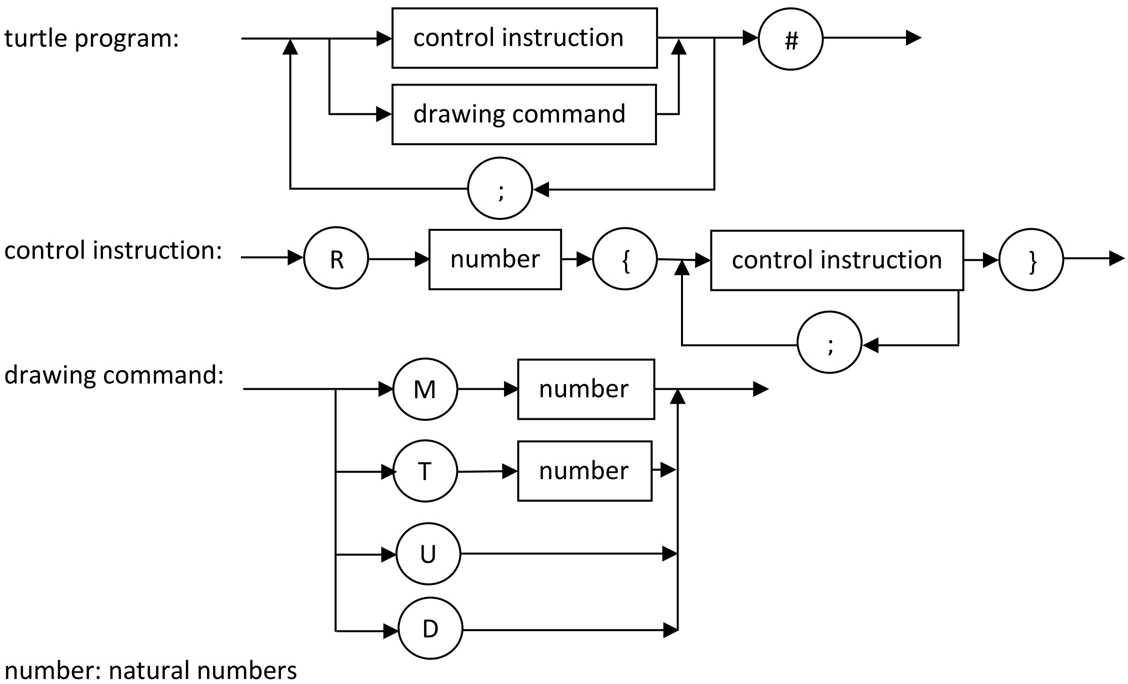
We want to develop a small programming language that we can use to write programs for a turtle - that is, for every *Snap!* sprite. The project should show how a text-based language works and how the error messages are generated. We reduce the problem a little by allowing one-letter commands only. If we look at the possibilities of the pen used in *Snap!* and select some of them, we get a possible command set (very small here):

Mn moves the turtle by the distance of length *n* in the current direction
Tn rotates the turtle on the spot by *n* degrees
U lifts the pin
D lowers the pin

We add a control structure to these four commands, here: a loop - and the minimal version of a programming language is ready.

Rn{ drawing commands }

We cast this rough sketch in the form of syntax diagrams: A turtle program consists of a sequence of commands separated by semicolons. The program ends with a double cross sign.

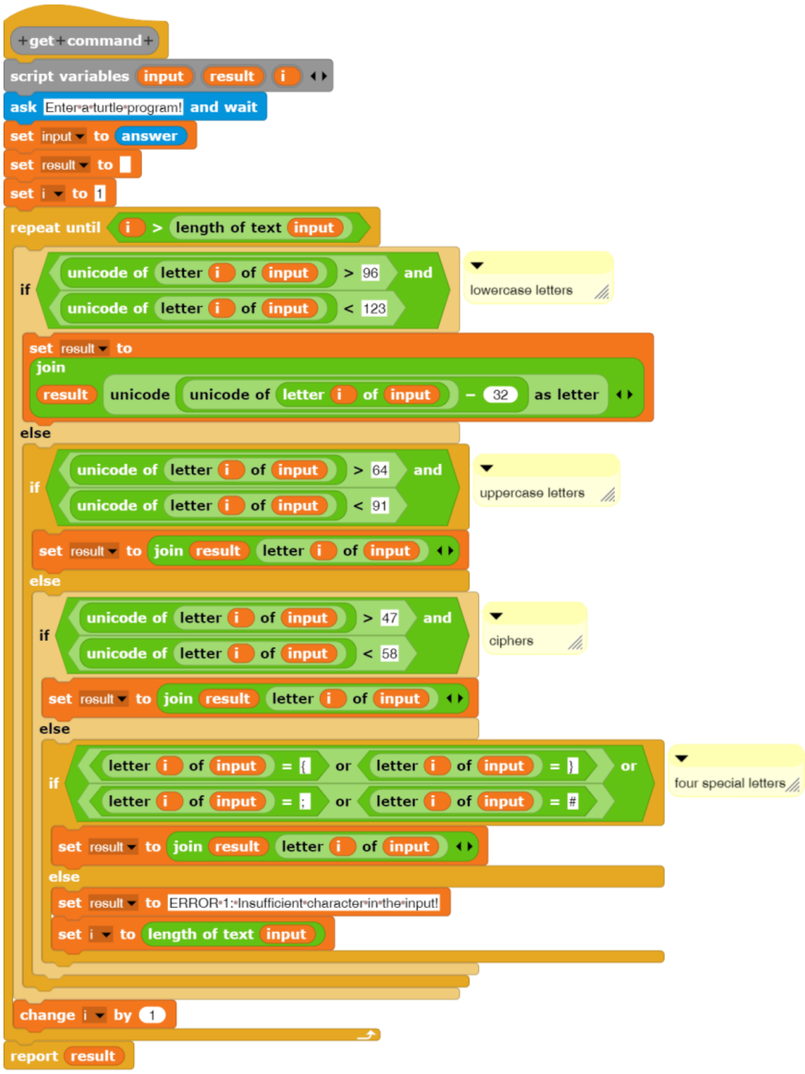


number: natural numbers

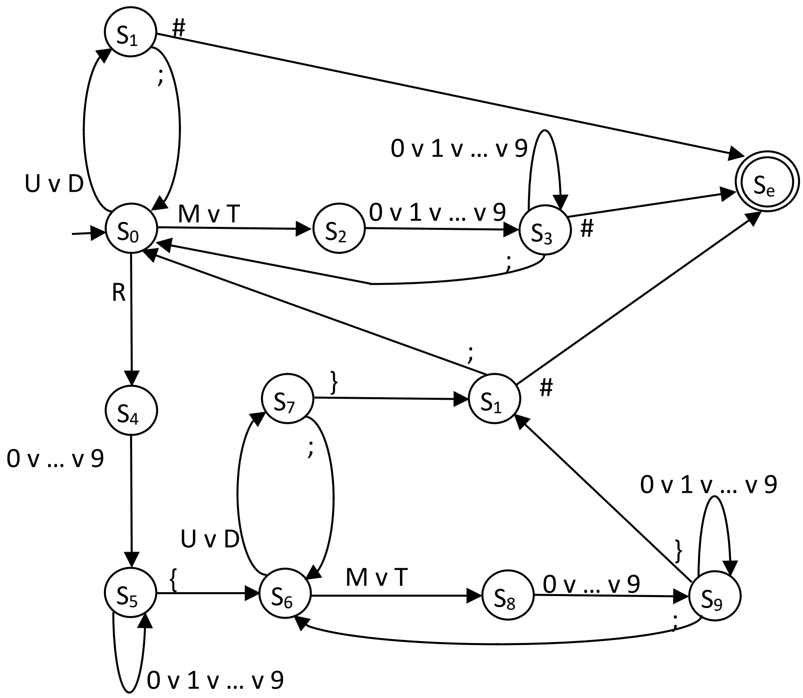
Programs are e.g.: *D;R4{M100;T90};U#*
M100;T90;M100;T90;M100;T90;M100;T90#
D;R180{M200;T183};R360{M1;T1}#

We assume that superfluous characters such as spaces are removed from the program first. We can achieve this, for example, by converting entered lowercase letters into uppercase letters and allowing digits and the four special characters "*;*", "*#*", "*{*" and "*}*". All other characters lead to the error message "*ERROR 1: Wrong character in the input!*".

So, we write a simple input method with character control.



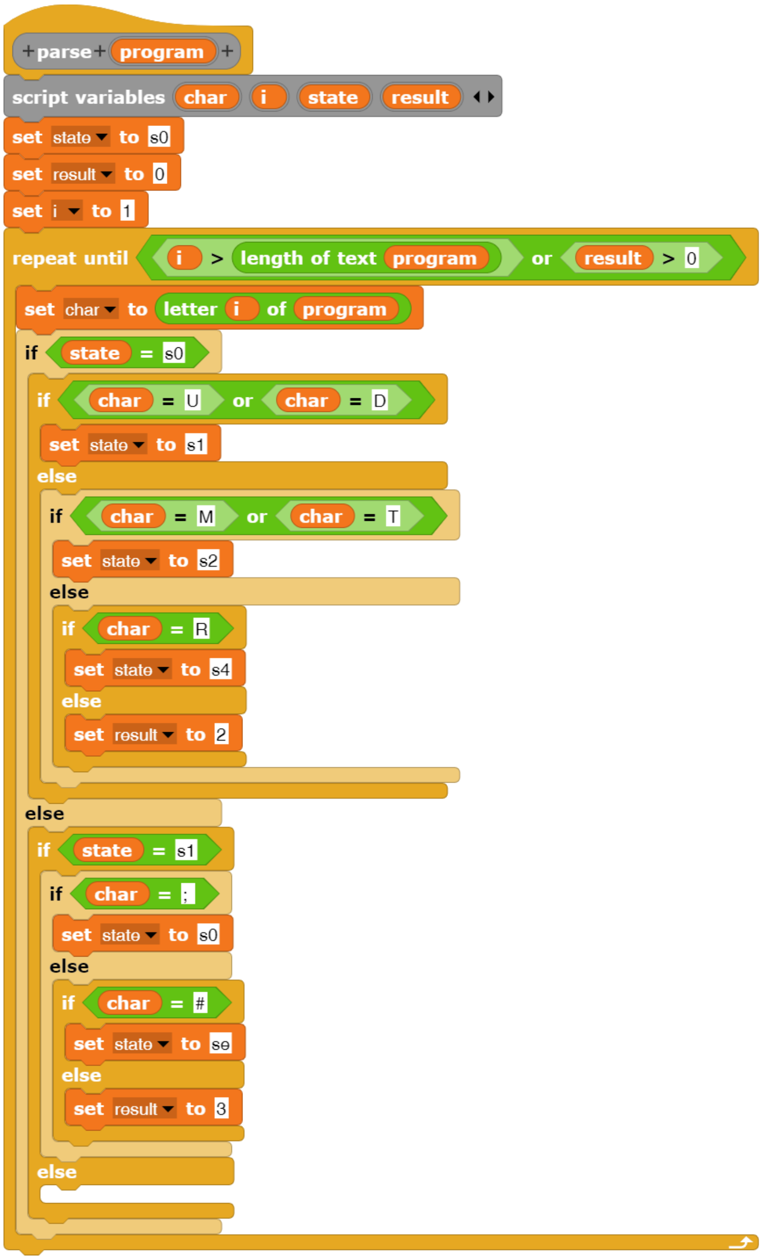
The input must be checked to see whether it represents a permitted LOGO program - it is "parsed". In this case we can develop the parser as a finite automaton[75]. The unspecified transitions lead to an error state.



[75] Why is that, by the way?

In the individual states we can decide which signs lead to subsequent states and which do not. This allows us to indicate which characters were actually expected in the event of incorrect entries. If we number these error messages of the parser in the order of their occurrence, we get the adjacent table. If we also evaluate the position of the character in the command where the error occurred, then we can even display the error.

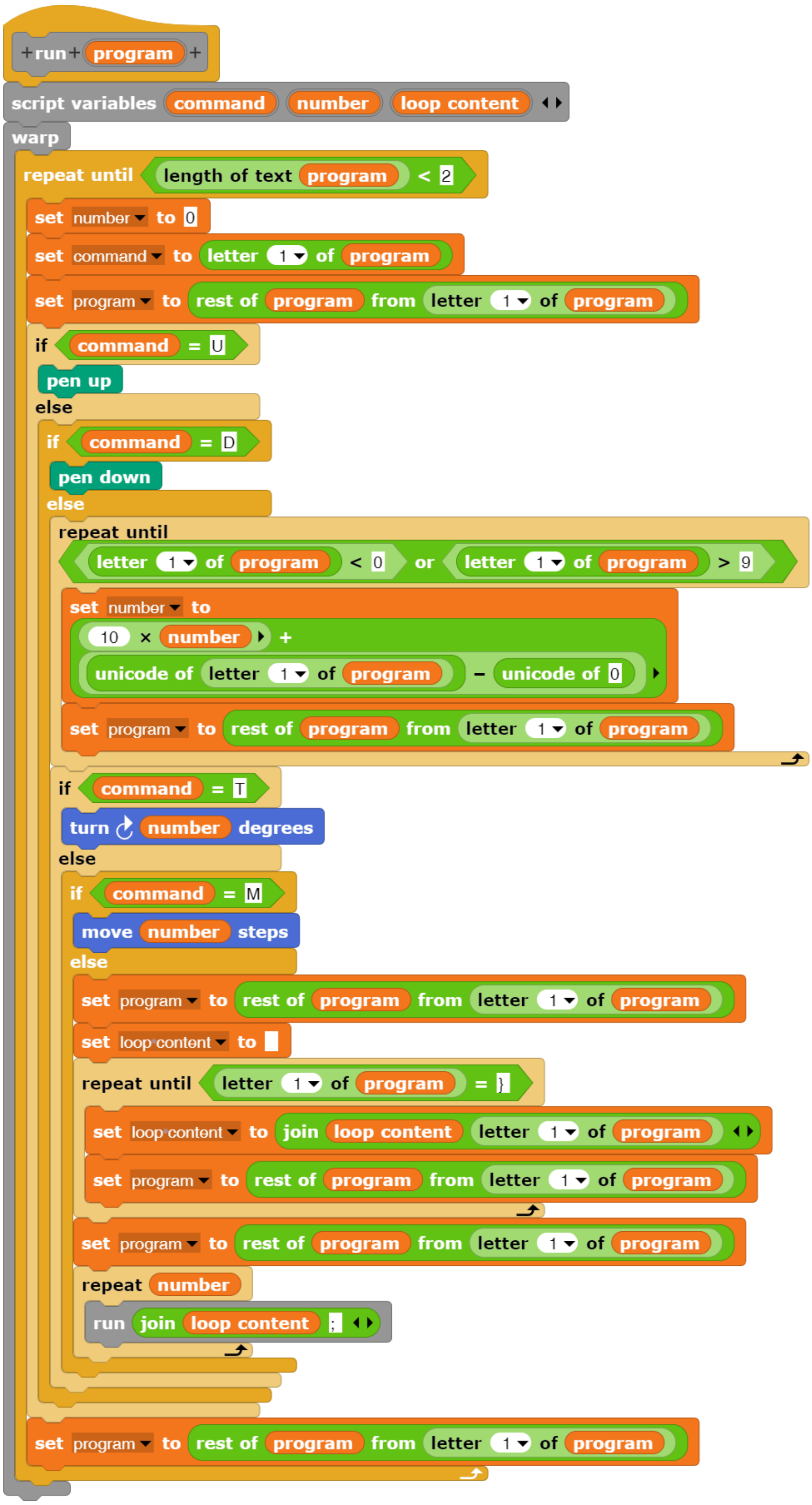
state	possible error message
S_0, S_6	2: unknown command
S_1, S_{10}	3: <;> or <#> expected
S_2, S_4, S_8	4: number expected
S_3	5: number, <;> or <#> expected
S_5	6: number or <{> expected
S_7	7: <;> or <}> expected
S_9	8: Zahl, <;> or <}> expected
	9: unexpected end of input



The translation of the parser consists only of a very long copy of the state graph - of nested alternatives. We only show the first part.

The parser *parse <program>* is guided through the state diagram by the character string of the program. If there is no permissible transition in a state, it reports the corresponding error by the value of the "result" variable. Correct programs have the value *0* as a result.

The interpreter *run <program>* can assume that the entered program is error-free - after all it was parsed. Therefore, it can take the first character of the program one after the other - this is the next command - and delete this character. Depending on the command, it executes this and searches for the required parameters, e.g. the angle of rotation. All processed characters are deleted. This ends when the program consists only of the last character – the "#".



The program is processed character by character, the processed characters are deleted. We use the function *rest of* from out string library.

PenUp command (U)

PenDown command (D)

gather number

Turn command (T)

Move command (M)

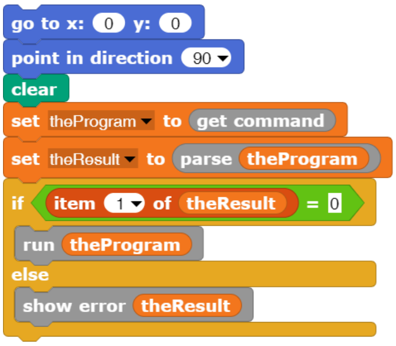
run the loop (R)

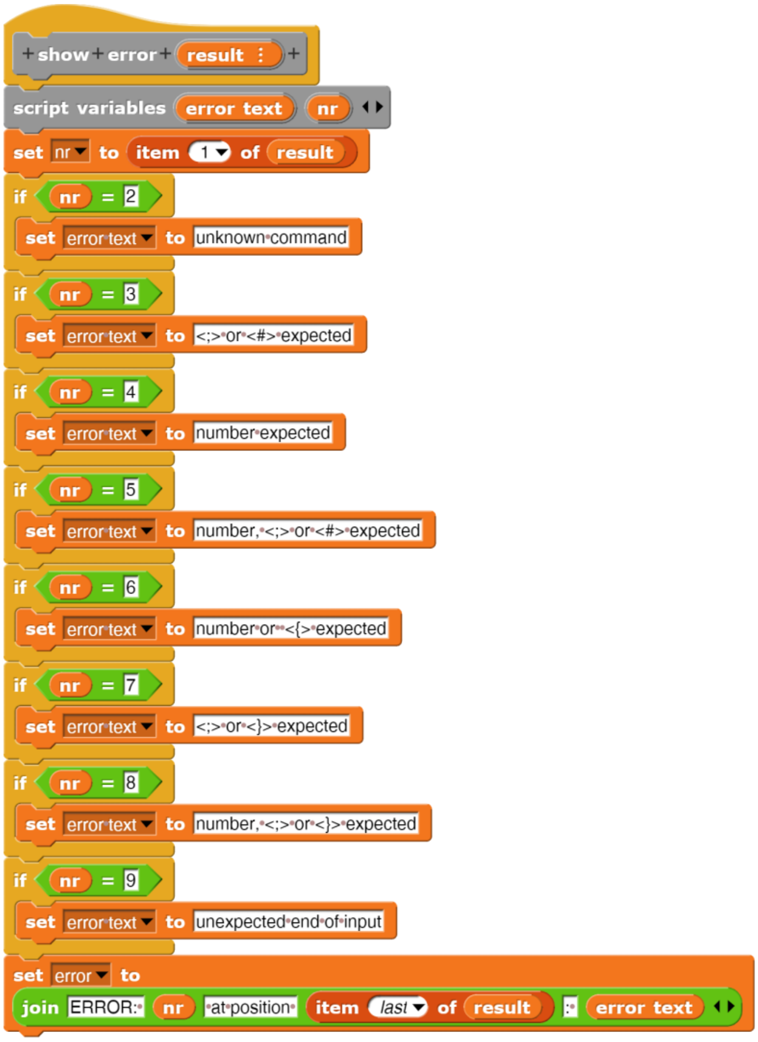
search for loop contents until the next "}"...

... and execute as often as the number indicates. Append a ";" to the loop contents.

If we output the error messages in plain text, then our programming language will slowly become usable.

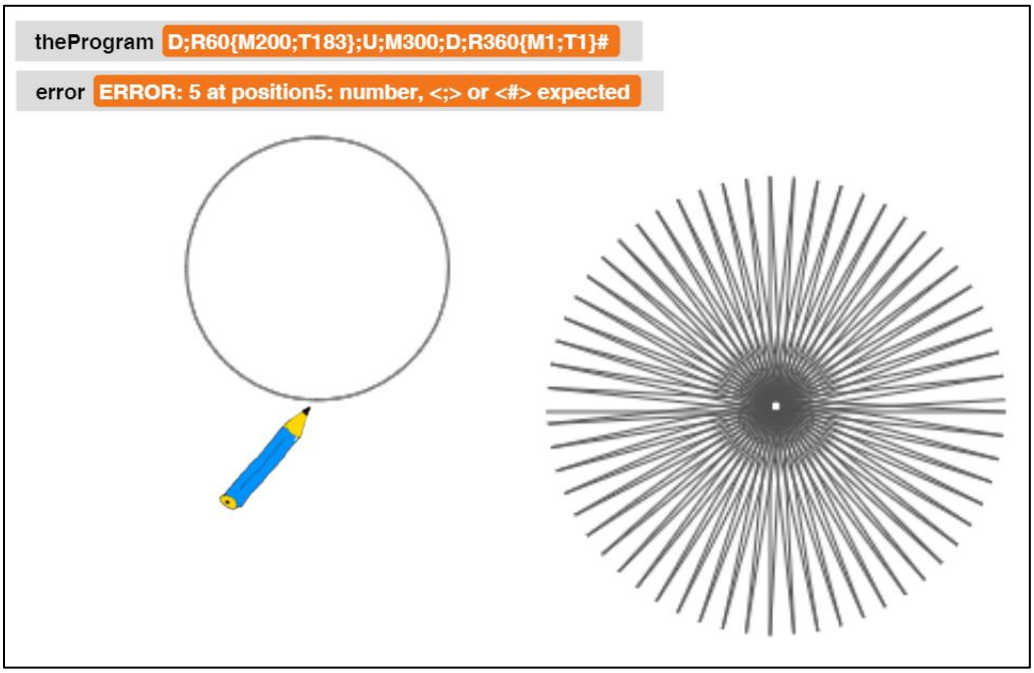
We can evaluate programs with a short script.





We should realize that the definition of this language is purely arbitrary. The body of the loop could also be enclosed with square brackets, with percent signs or smileys instead of curly brackets, and the fact that statements are separated by semicolons, but not terminated, also arises only from the current whim. A program is syntactically "correct", if it corresponds to the language definition, and this again corresponds to the conceptions of the language developers. It does not follow from generally valid rules.

It is also possible to learn from the error messages. They indicate where an error is noticed, not where it was made. The indicated error position can therefore lie far behind the actual location of the error.



Actually, it is a bit strange to develop a very primitive text-based language in a graphical programming language. However, experience shows that learners usually combine the work of computer scientists with the development of cryptic texts - i.e. they sometimes want to program "really". We can accommodate this wish if we use such a mini-language in a standard field of computer science, in this case automata theory. Since we develop it ourselves, we promote understanding for the processing of texts, which takes place on many levels in IT systems. In addition, we have found a highly differentiating topic suitable for division of work and challenging activities, which quickly leads to presentable results.

Tasks

1. **Expand** the language **LOGO** by
   a: a Home (H) command that sends the turtle to the center of the screen.
   b: a Clear command (C) that clears the screen.
   c: a Color<n> (Fn) command that allows you to select a pen color.
   d: a command TurnTo<angle> (Nn), which rotates the Turtle to a certain angle.
   e: a command MoveTo<x><Y> (Vx,y), which sends the turtle to a certain point.

2. Develop a **scanner** that allows you to enter the turtle commands in long form, for example, to write *Turn 90* instead of *T90*. The scanner should recognize these commands and output them again in short form.

3. Introduce an **alternative**: Depending on the color of the pixel at the location of the turtle, it should be possible to execute different command sequences. Reduce the syntax appropriately and implement the command.

4. Two types of **loops** are to be introduced in this way: The turtle should execute drawing commands as long as (*WHILE*) or until (*DO*) the turtle is above pixels of a specified color. Allow position-dependent predicates as well.

17.2 SnapMinder by Jens Mönig[76]

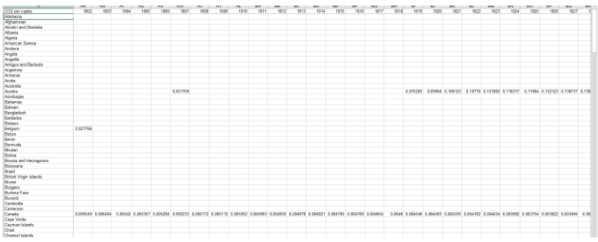
Level: *high school* Materials: *SnapMinder*

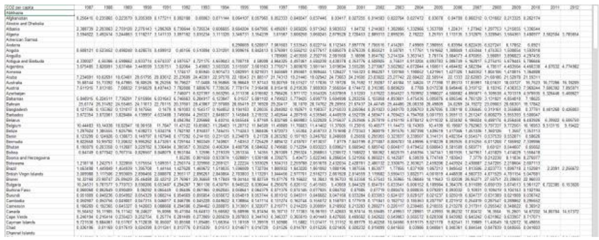
The program is based on data from the *Gapminder Foundation*[77], which provides tools for visualizing statistical data on the Internet. One of these shows the development of the countries in the recent past, whereby life expectancy is represented above income and the size of the "bubbles" corresponds to the total population of the country in one year. If you move the slider, you can impressively follow the temporal development of the countries in this coordinate system. For me, the program is a wonderful example of how visualization can be used to identify anomalies in data *("Why does a country suddenly drop down?" "Why does a country move in circles?", ...)*, the causes of which can then be explored.

The data used - and many others - can be found in tabular form at https://www.gapminder.org/data/.

Importing Table Data

To import the required data, we load the file into a spreadsheet program and immediately save it again as a tab-delimited text file. Let us take CO2 emissions per person from 1751 to 2012[78] as an example. For the first years we find only a few values, but then it gets dense.



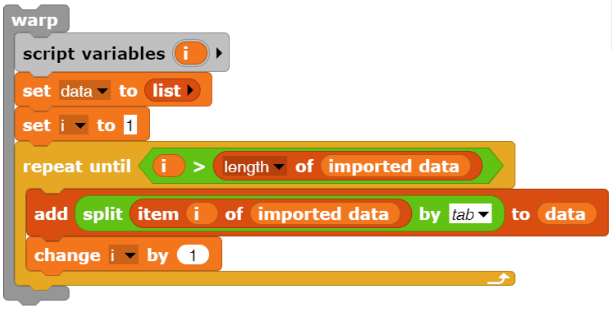


We read the generated text file into a variable via its context menu (*import...*). To do this, it must be displayed in the work area. We get a very long string of characters.

[76] With permission of the author, available at snap.berkeley.edu/run#present:Username=jens&Project-Name=SnapMinder
[77] https://www.gapminder.org/
[78] CDIAC: Carbon Dioxide Information Analysis Center

We turn them into a list: 

Each line again contains a character string with the data for each country, whereby the data are separated by tabs. Therefore, we "hack" the list line by line in the same way, but with a different separator, and add the sublists to a new list variable called *data*.



This provides the necessary data for editing in *Snap!*.

The SnapMinder Data

The program contains the required data as described above in the variables *income data*, *life data* und *population data*. It prepares them for further use with the help of higher order list operations[79]. As an example, we show the population:

Convert the *population data* into a list (here in "one step") and throw out those "without interesting content".

The operations are very compact due to their nesting. If you take them apart, however, they are easy to understand. As an example, we take the first nested block. It can be read "from behind" as ...

transform data of the existing countries into a table as discussed above,

discard unusable data ("... no numbers") and

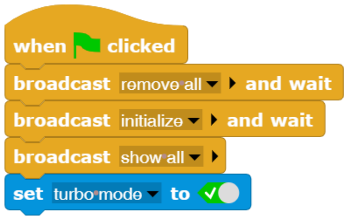
assign the result to the *population* variable.

population

194	A	B	C	D	E	F
1	Total population	1800	1810	1820	1830	1840
2	Afghanistan	3280000	3280000	3323519	3448982	3625022
3	Albania	410445	423591	438671	457234	478227
4	Algeria	2503218	2595056	2713079	2880355	3082721
5	Andorra	2654	2654	2700	2835	3026
6	Angola	1567028	1567028	1597530	1686390	1813100
7	Antigua and Barbuda	37000	37000	37000	37000	37000
8	Argentina	534000	534000	570719	686703	873747
9	Armenia	413326	413326	423527	453507	496835
10	Aruba	19286	19286	19555	20332	21423
11	Australia	351014	342440	334002	348143	434095
12	Austria	3205587	3286650	3391206	3538286	3728381

[79] Jens Mönig uses a little trick: If you move the block of a list operation over the *join* block from the string operations, which is displayed "empty" (join), i.e. without input parameters, then it turns into the *join input list*-Block (join input list: all but first of), which converts the list into a simple string. The function can also be easily written by the user.

The program starts with three messages that cause old country sprites to delete themselves and initialize the other objects, especially the data lists. For the data this causes:



Process the data as described above. First the income, ...

... then life expectancy, ...

... and extract the countries from it.

Assign the income to the countries.

Same for life expectancy.

Write back the population data from the auxiliary variable.

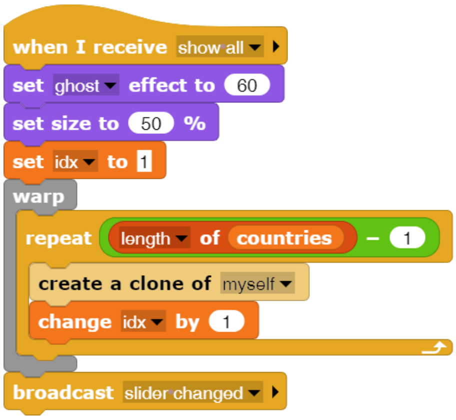
Set some variable values.

Extract the years.

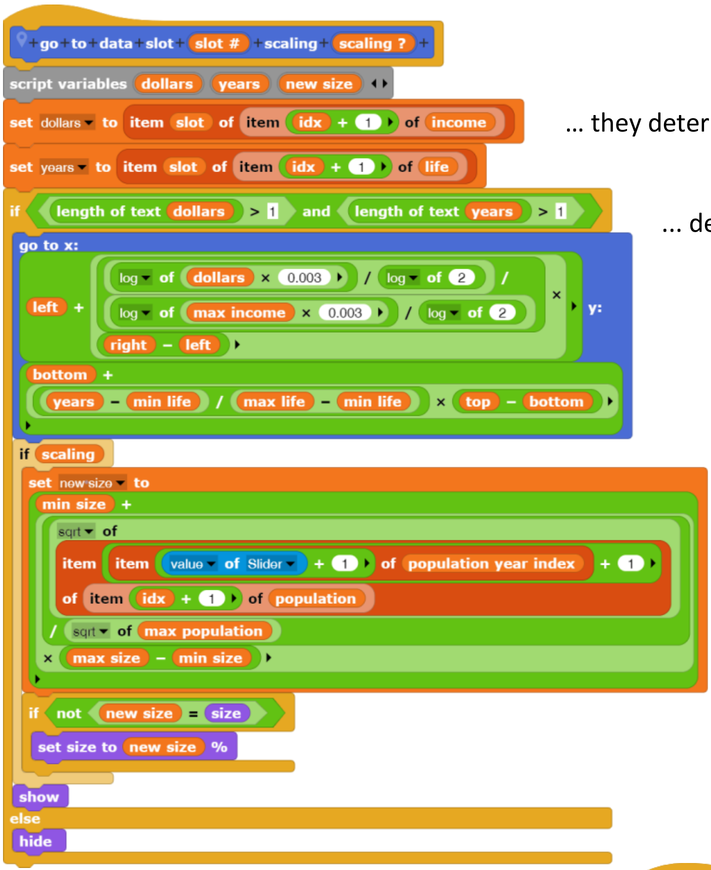
Create a list of years as an index.

The SnapMinder Countries

At the start of the program as many *country* clones, represented by a semitransparent rectangle, are created as *countries* are included in the country list. Each clone has its own index *idx*.



The main function of the countries is to position themselves in the coordinate system of average income and life expectancy in relation to the year under consideration. For this ...

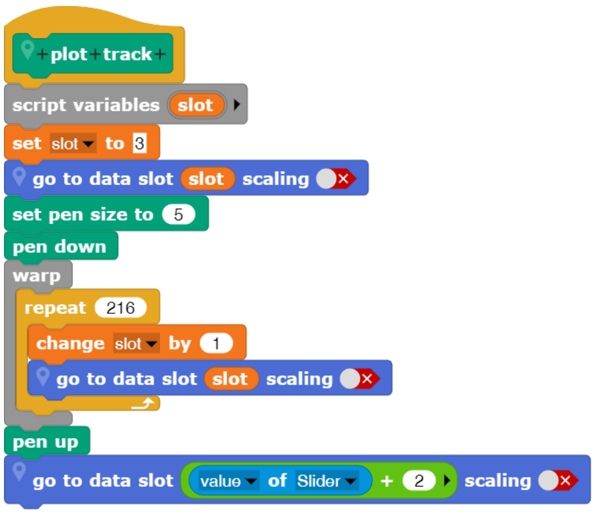


... they determine these data for their country, ...

... determine the position ...

... and their size, which is given by the population of the country in the year under consideration.

This block is called, among other things, when a plot of the country, i.e. the movement in the coordinate system with the year as parameter, is generated.



Use SnapMinder

The presentation is impressive because, on the one hand, the countries move from bottom left to top right in the course of time, i.e. they develop positively. But if you take a closer look at some countries, this development is by no means continuous: there are abrupt downward swings, backward movements, circles, periodic movements, ... The program gives rise to research into the causes of these developments, and there are a few surprises! We show plots of some countries, then you should do research! 😉

USA

Germany

China

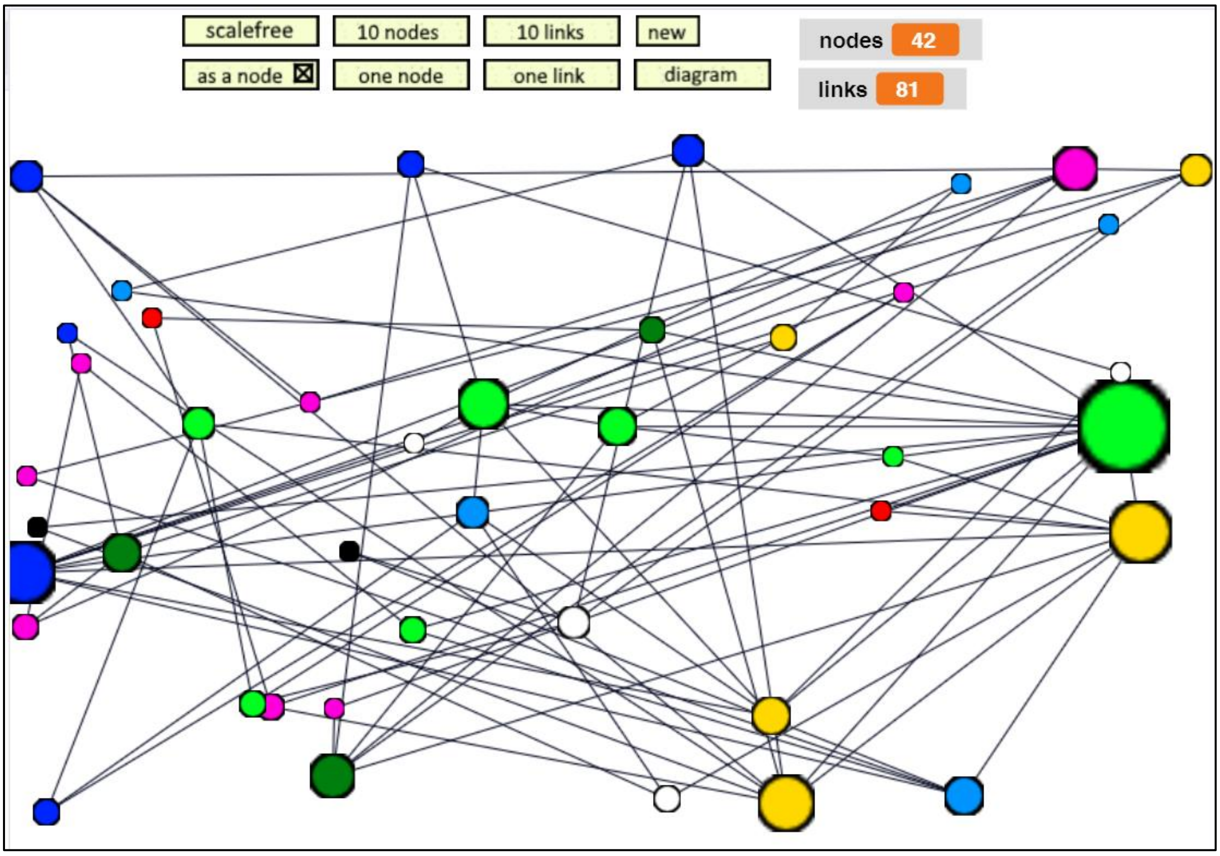
India

Norway

Somalia

17.3 Connectivity: The World is Small[80]

Level: *high school* Materials: *Connectivity*



The handling of networks is often reduced to protocols and other technical details. But you can also ask other questions, e.g. about the connection of networks.

- If we have *n* nodes, how many links do we need for the network to be largely connected?
- Or vice versa: How many and which nodes must be destroyed for a network to break up into its subnets?
- Or: What is the mean distance, counted in links, between the nodes of a network?

Nodes and links can be very different in nature. It can be e.g.

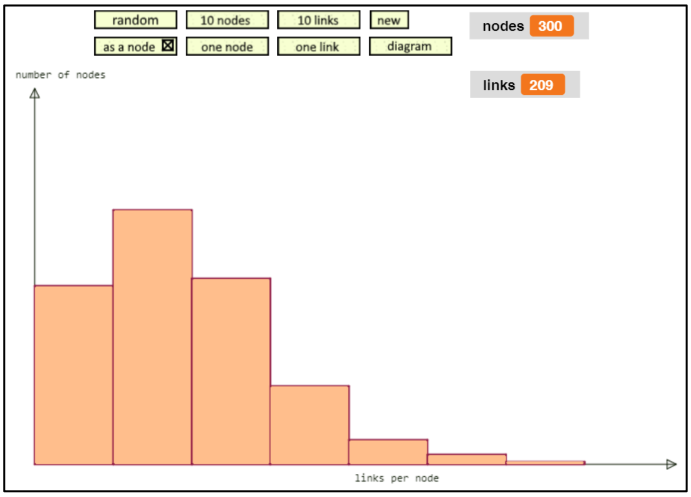
- technical links between computer systems,
- customer/supplier relations in the economy,
- the logical connections via linked websites,
- social relations between persons or groups of persons
- hydrogen bonds in organic compounds,
- neuronal networks
- or infection chains.

[80] based on E. Modrow: Informatik mit Delphi – Band 2, emu-online, 2003

Random Networks

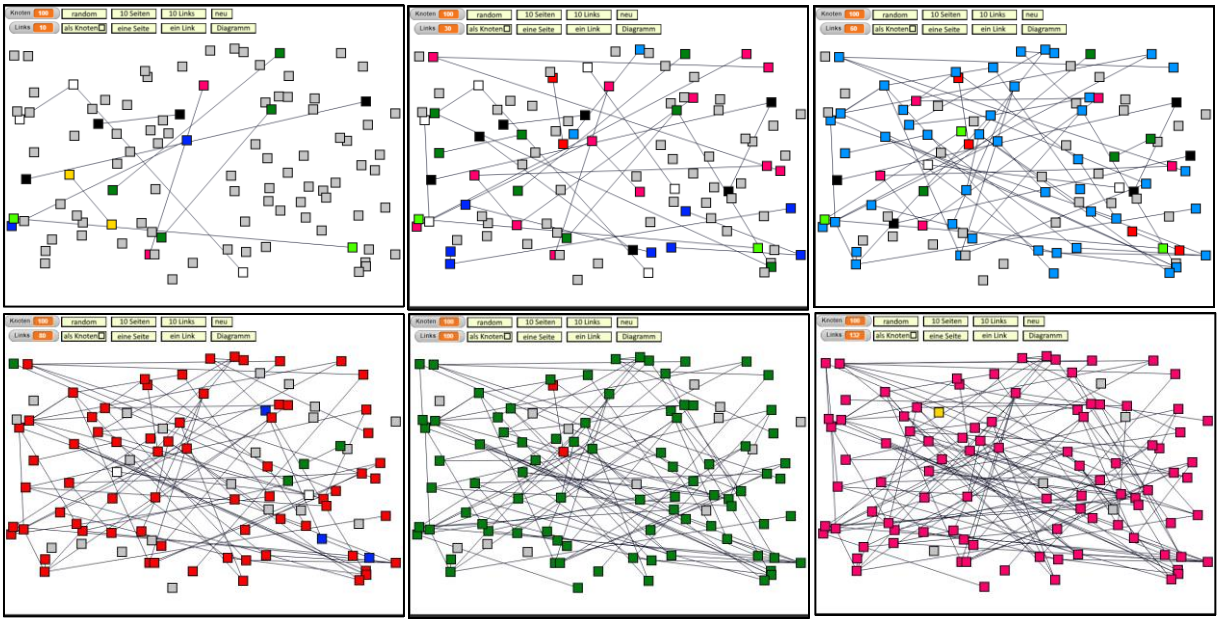
The starting point for such questions were *Random Networks*. They are created when we build *N* network nodes (or pages, ...) that we subsequently link to each other. Let us take the Internet as an example. If there are *N* pages with on average *k* links per page, then with *n* mouse-clicks k^n pages are accessible. We can reach virtually any page if it is: $k^n = N \rightarrow n = \log N / \log k$. With *5 billion* pages and $k = 7$, $n = 11.5$, i.e.: with about 12 mouse clicks on average, you can visit any page of this network. Similar considerations and practical studies have been carried out on social relations, etc. They can be found under the name *Small World Phenomenon*[81].

If you display the distribution of links per page, you get a *Poisson distribution* for Random Networks.



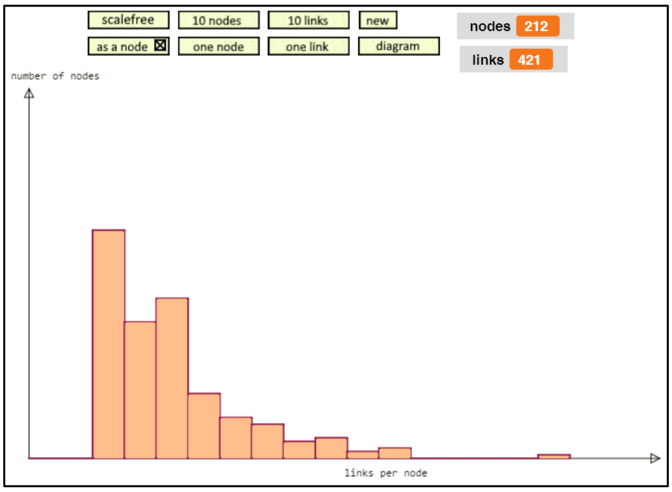
It is somewhat more difficult to decide whether a network is (largely) coherent, i.e. whether all nodes are connected to each other. We can answer this question by coloring: start with one node and color all the nodes that can be reached by it in the same color, then a coherent network shows a kind of phase transition: almost suddenly all nodes take on the same color.

You can see that the network - except for a few slips - is coherent if the number of links roughly corresponds to the number of nodes. Further links do little to change.



[81] https://de.wikipedia.org/wiki/Kleine-Welt-Ph%C3%A4nomen

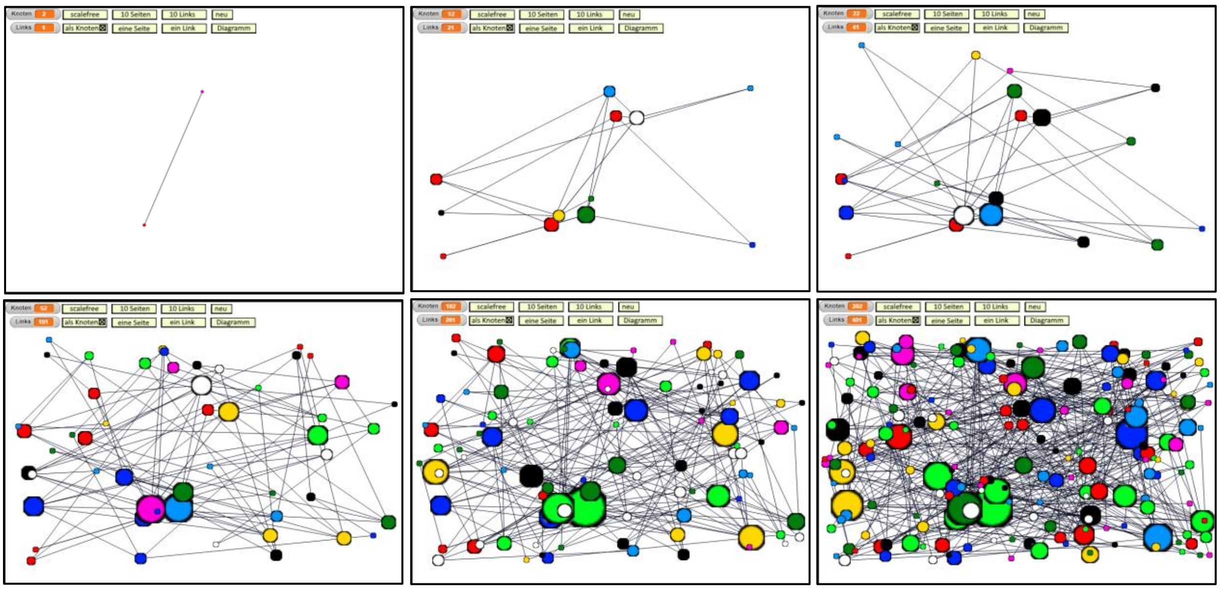
Scalefree Networks



Albert-László Barabási[82] showed in 2002 that growing networks like the Internet have a different distribution of links per node than Random Networks. It can be described by a *Pareto distribution*. Brief descriptions can be found in

http://barabasi.com/f/623.pdf bzw. *http://barabasi.com/f/624.pdf.*

A *Scalefree network* can be created by alternately adding nodes and links where the new nodes have two links to existing nodes. The older nodes are more likely to be linked than the younger nodes. Because the network is always coherent, there is no need to color contiguous nodes. But we want to make the size of the nodes dependent on the number of their links.



Scalefree networks are the same on all scales, i.e. numerous nodes with few connections are connected to a few nodes with many connections, so-called *hubs*. The connections between nodes normally run from the start node to the next hub, then via a few more hubs to the target node. Hubs can be, for example, people with many contacts (teachers, representatives, ...), central computers or distribution centers in merchandise management.

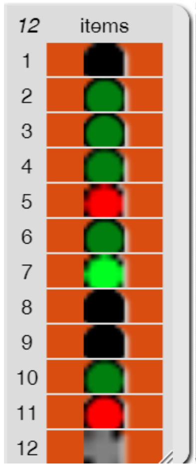
Scalefree Networks are extremely robust against technical faults. For example, if a network connection happens to fail, it probably does not affect a hub, and if it does, other hubs will compensate this. However, they are also extremely susceptible to targeted interference. If only a few hubs in this network type are destroyed, the network disintegrates into its individual parts.

The topic is suitable as an introduction to discussions about vaccination protection, preventing the spread of diseases, influencing political opinion-forming, optimizing the flow of goods, ...

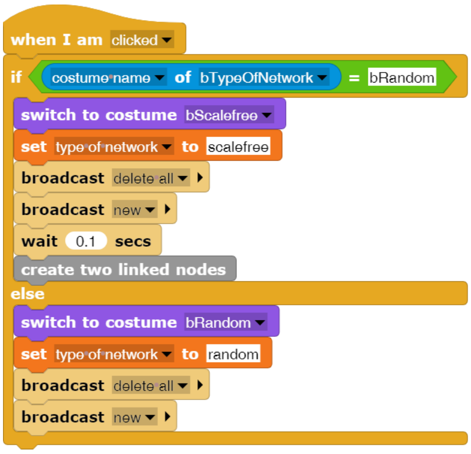
[82] A.Barabási: Linked: the new science of networks, Perseus Publishing 2002

The Implementation

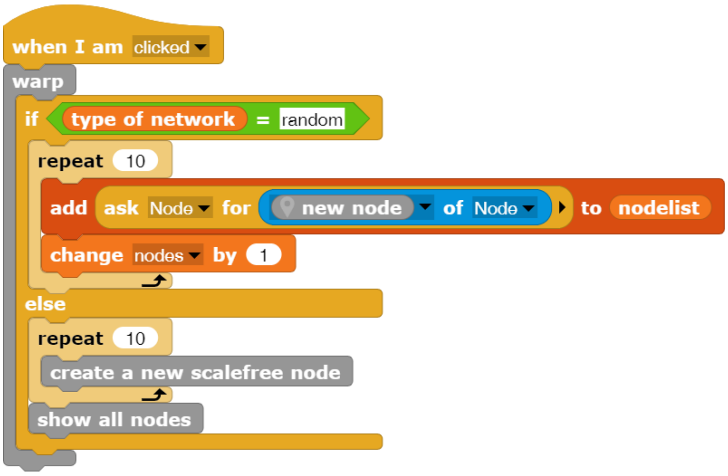
We want to create a fairly simple model as a tool for researching network properties. It is essentially based on a *node* from which clones are generated and two lists, of which the *node list* contains the nodes already generated and the *link list* consists of sub-lists with the numbers of the two end nodes of the links. With their help, methods can be implemented largely independently of each other. They are used by the operating elements shown. The controls depend on the selected net type (*random/scalefree*) and the display of the nodes (*rectangular/round with different sizes*).



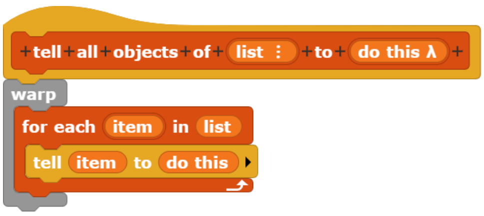
13	A	B
1	9	7
2	4	6
3	8	3
4	9	11
5	1	3
6	3	9
7	6	3
8	10	4
9	6	2
10	8	2
11	10	2
12	1	8
13	5	11



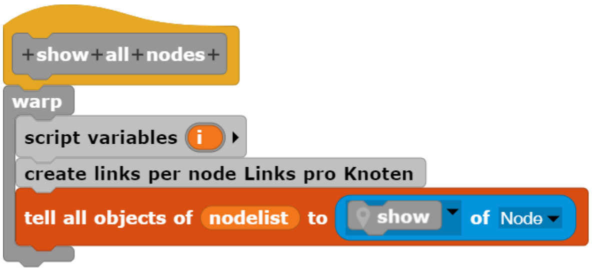
Buttons for switching between net types or for creating 10 nodes react to mouse clicks:



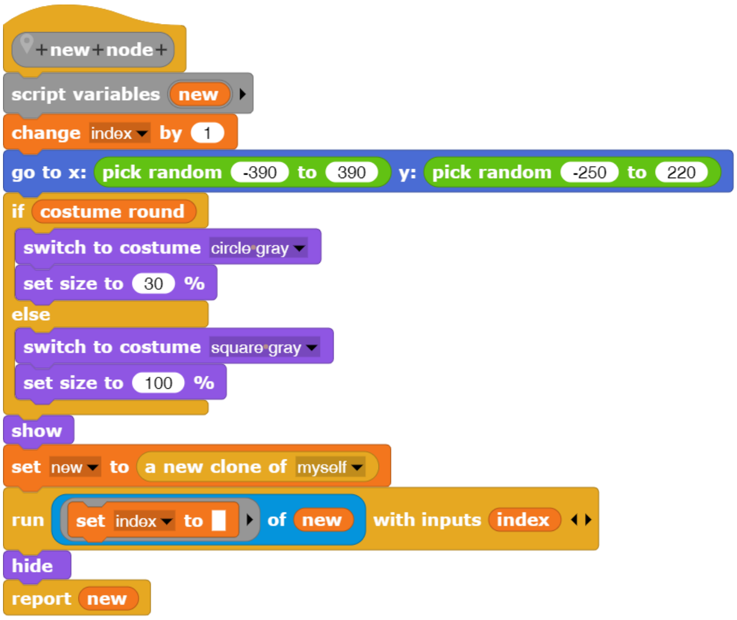
Since we often have to iterate over such node lists, we introduce a new control structure that executes an instruction for all objects in a list:



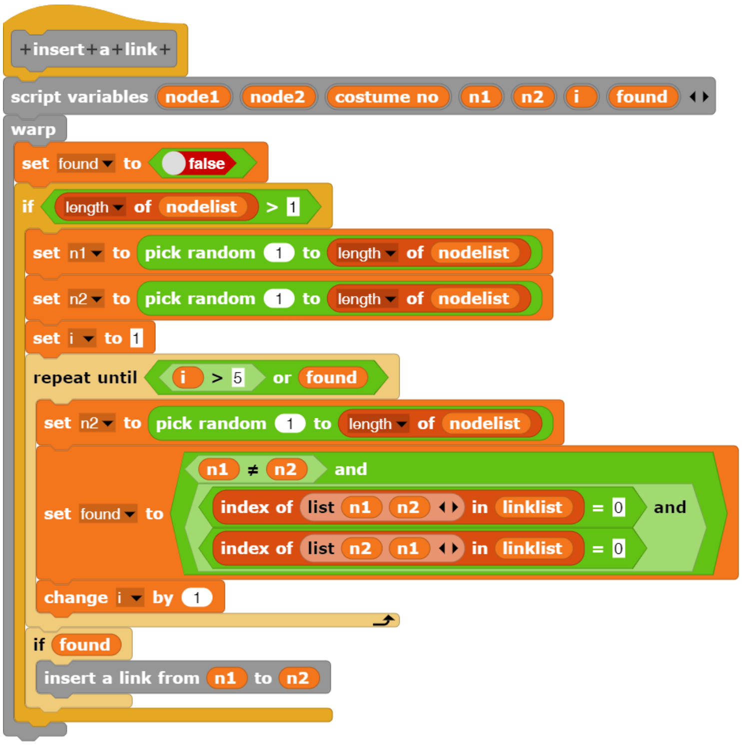
This makes it very easy, for example, to display all nodes:



New nodes are created by cloning the prototype. The prototype can be asked to perform this action.



A new link is inserted into the network by trying to find two nodes that are not yet connected. The link list must then be searched to see if the link already exists. If not, the search returns *0*. This allows the ends of the link to be determined. Since the resulting nets are quickly becoming large, the search for them does not take too long.



Once you know which nodes are to be connected from a link, ...

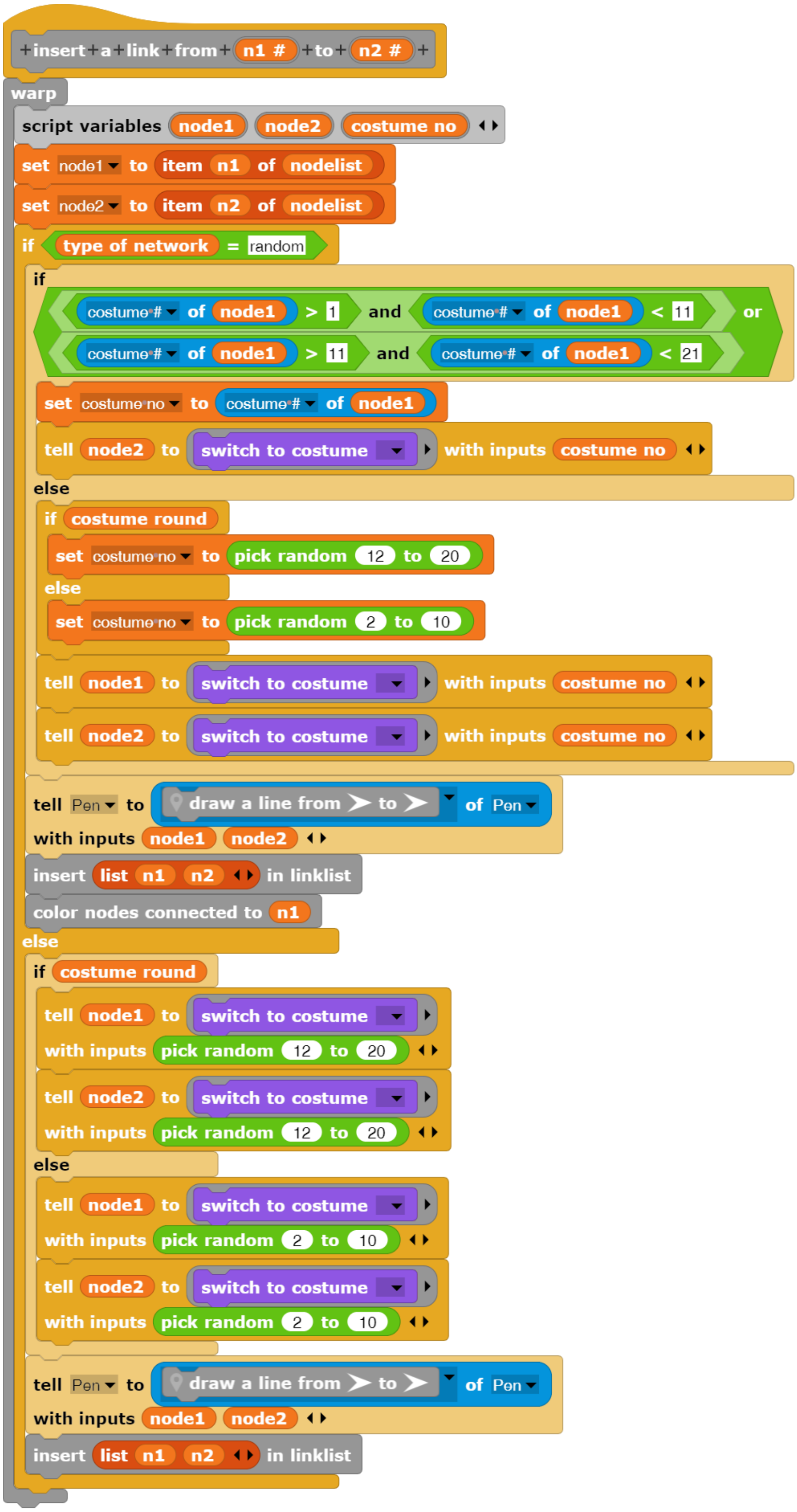
... the affected nodes are searched for, ...

... the costume according to the net type is selected, and the nodes are asked to change to it.

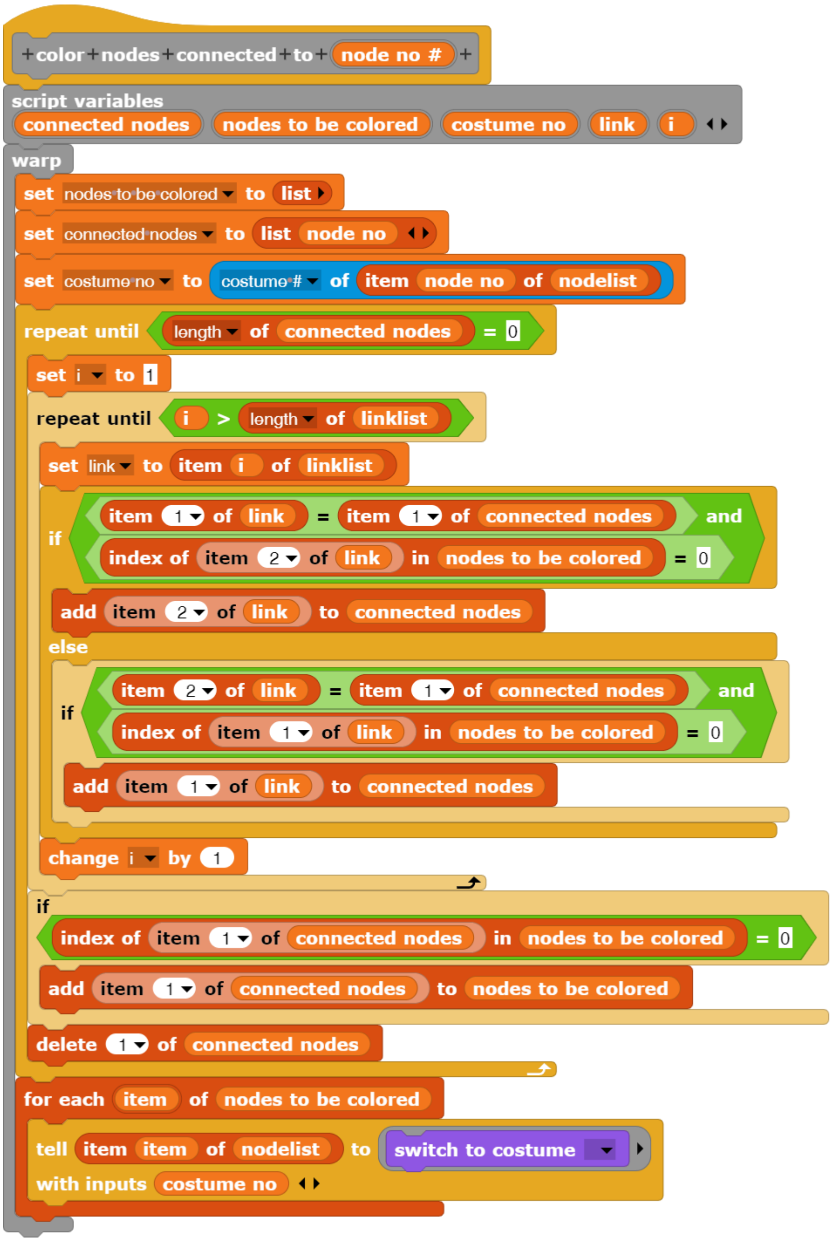
The pen is asked to draw a line between the nodes.

Finally, the new link is entered in the link list and the related nodes are colored in the same way.

With Scalefree Networks it is a bit easier because the costumes are chosen randomly.



The most complex part is the coloring of the connected subnets. We work with two lists, from which the *connected nodes* get all nodes that can be reached from the starting node. The *nodes to be colored* contain the nodes that have to be colored – sic.



We start with the given node number as the beginning and remember its costume.

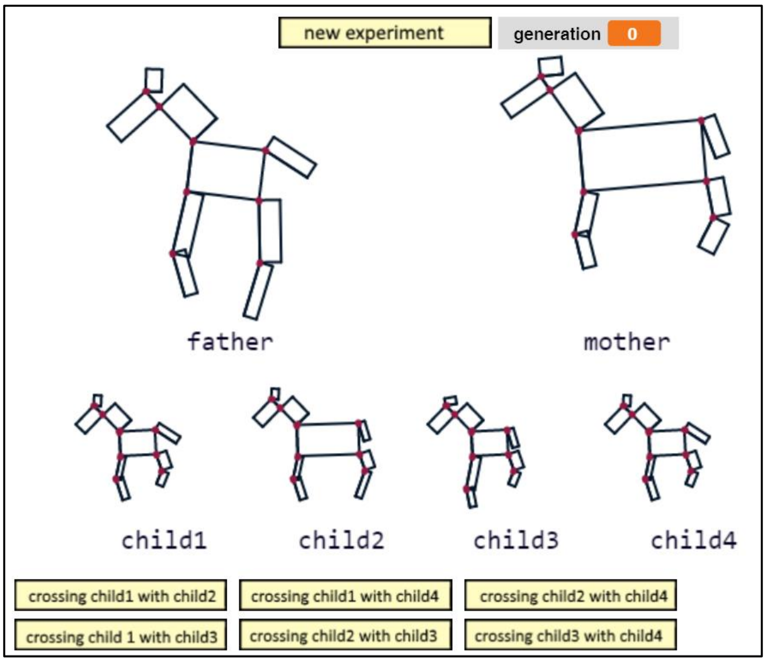
As long as there are still nodes in the list, we examine the link list to see if the first node number of the connected nodes appears in the link either to the left or right. If so, the other node is also connected to the source node and is added to the list if it is not already in the list.

If the first node in the list is not yet contained in the list *nodes to be colored*, it is entered there and removed from the list of *connected nodes*.

Finally, the costumes of all nodes to be colored are set to the same value as the costume number of the initial node.

The controls, the two (and further) net types, the creation, joining and coloring of nodes as well as the diagram creation are based on the sub-lists and can be developed largely independently of each other. The topic is therefore well suited for teaching in different working groups.

17.4 Evolution



Level: *high school*
Materials: *Evolution*

The aim of this small project is to produce a presentable result with the simplest possible methods, which can be used in class if required. The methods, e.g. for the representation of the animals, are partly found by "trial and error", which of course challenges improvements. That's the way it's supposed to be. The starting points of the parts are somewhat highlighted in the pictures.

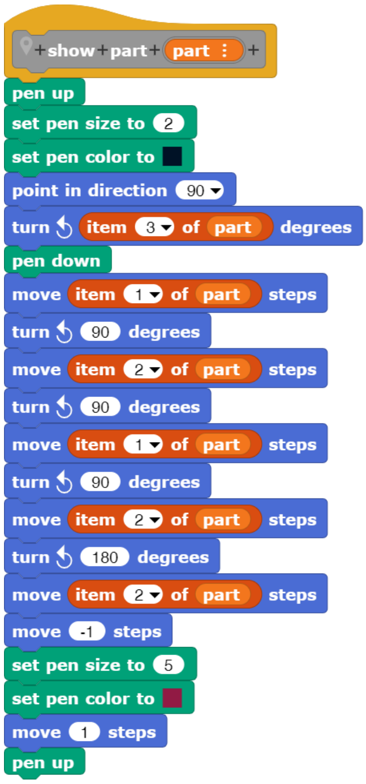
In the project, "animals" are randomly created, each consisting of 9 rectangles of random size, which are rotated to create a kind of horse. With a different composition, other "animals" can be quickly produced. The partial rectangles are always drawn in the same order and orientation, so that you have to try out where to start drawing. Of course, this problem can be solved more elegantly with some mathematics, and if parameters can be used to influence how a rectangle is drawn, then it can be done more beautifully - in a different way. But it can also be done quite simply.

After the production of two animals, four offspring are created and shown slightly smaller below. From these you can choose two and appoint them as new parents. If you repeat this, you can "breed out" certain characteristics, e.g. small heads or short legs. At each crossing, the characteristics are changed at random. If a part becomes too small, it falls away. So, you can breed something like seals or ostriches out of the initial horses.

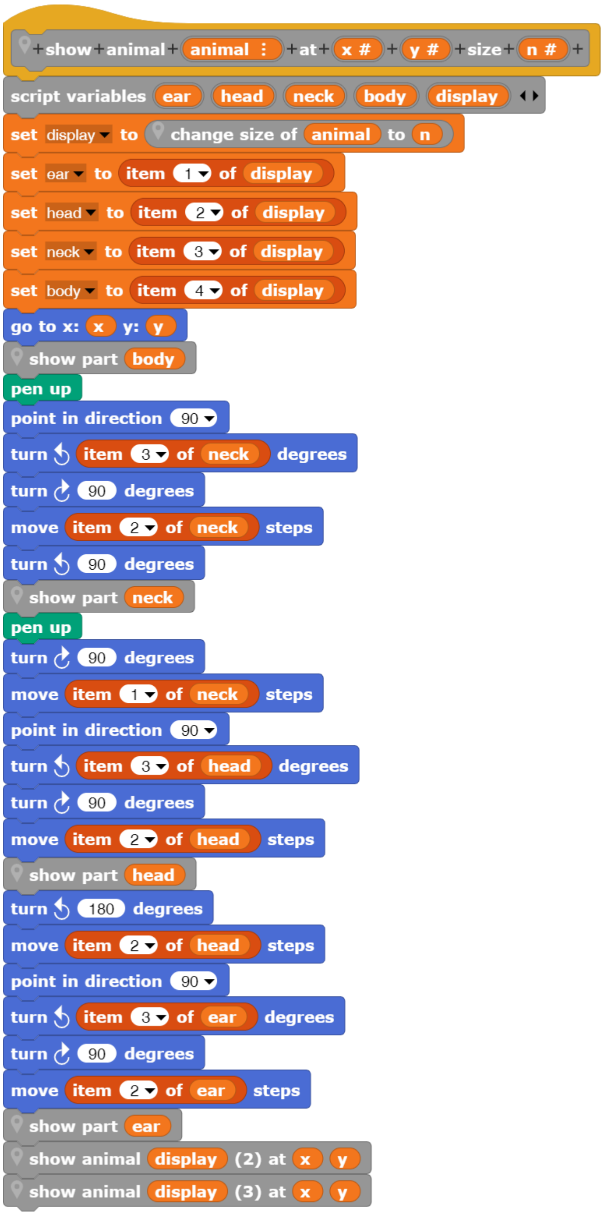
It makes sense to create new parts by mutations or to change the starting point of the parts, i.e. to let them "migrate". To do this, the data structures must be changed, for example by recording the coordinates of the approach points and adjusting the methods accordingly.

New animals can be created from the object *Animal*, which has a local method for this. In it, the parts of the animal are generated as lists of "reasonably usable" random numbers. They are then combined to form the complete list.

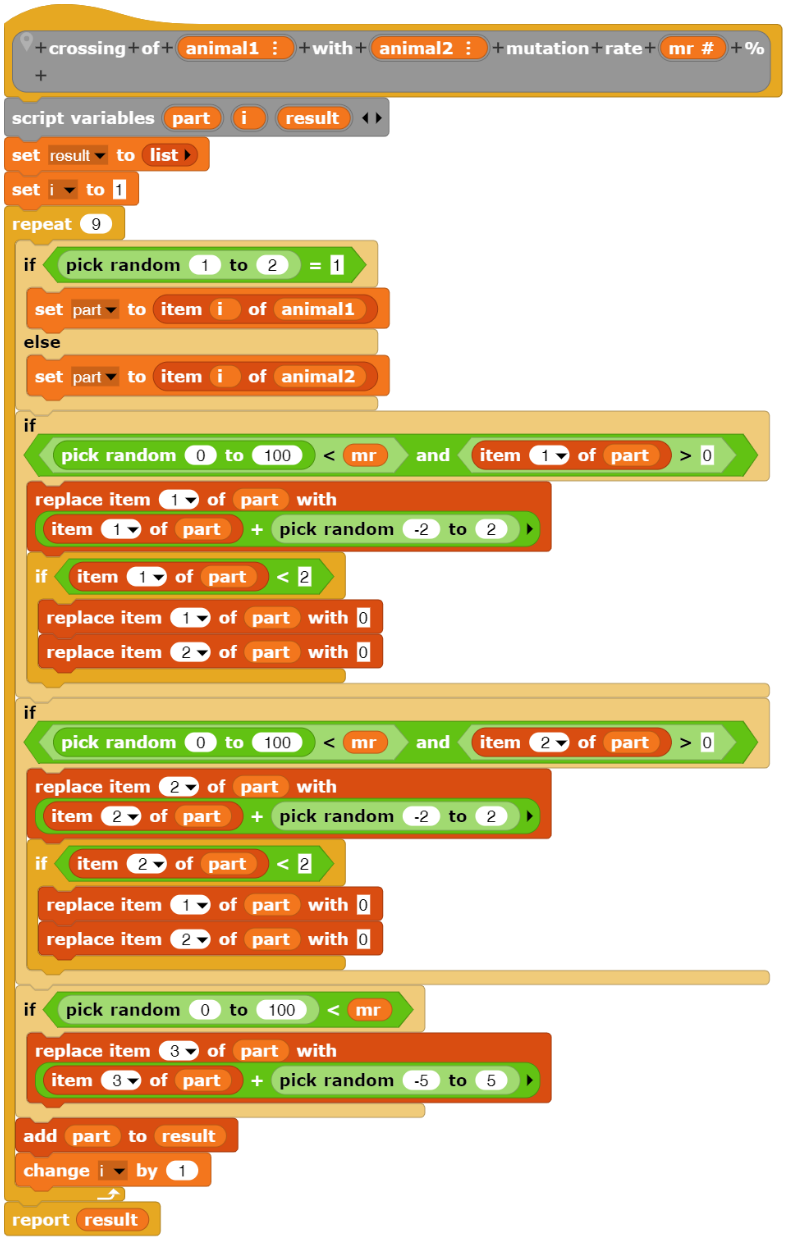
The parts of the animals are always drawn with the same method *show part*. The pen moves to the horizontal position and rotates to the angle passed as the third element in the list, then draws a rectangle with the lengths passed as the first and second element. In addition, the starting point is emphasized somewhat.



The method *show animal* first changes the size of the animal as indicated. Then the parts are drawn at the "tried out" points. Only the first part of it is shown.



Two animals are "crossed" by randomly assembling the parts of one or the other animal into a new one. During each of these processes the dimensions are changed randomly - depending on the mutation rate *mr*.



Select from which animal a part will be taken.

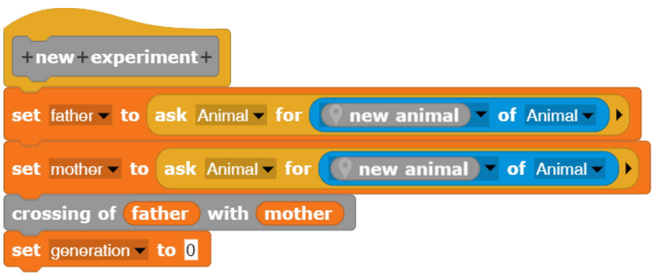
Change the width of the part randomly.

Too small parts are removed.

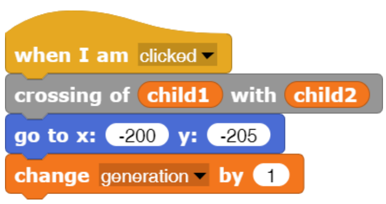
The same, for the height.

Add part to new animal.

Return result.

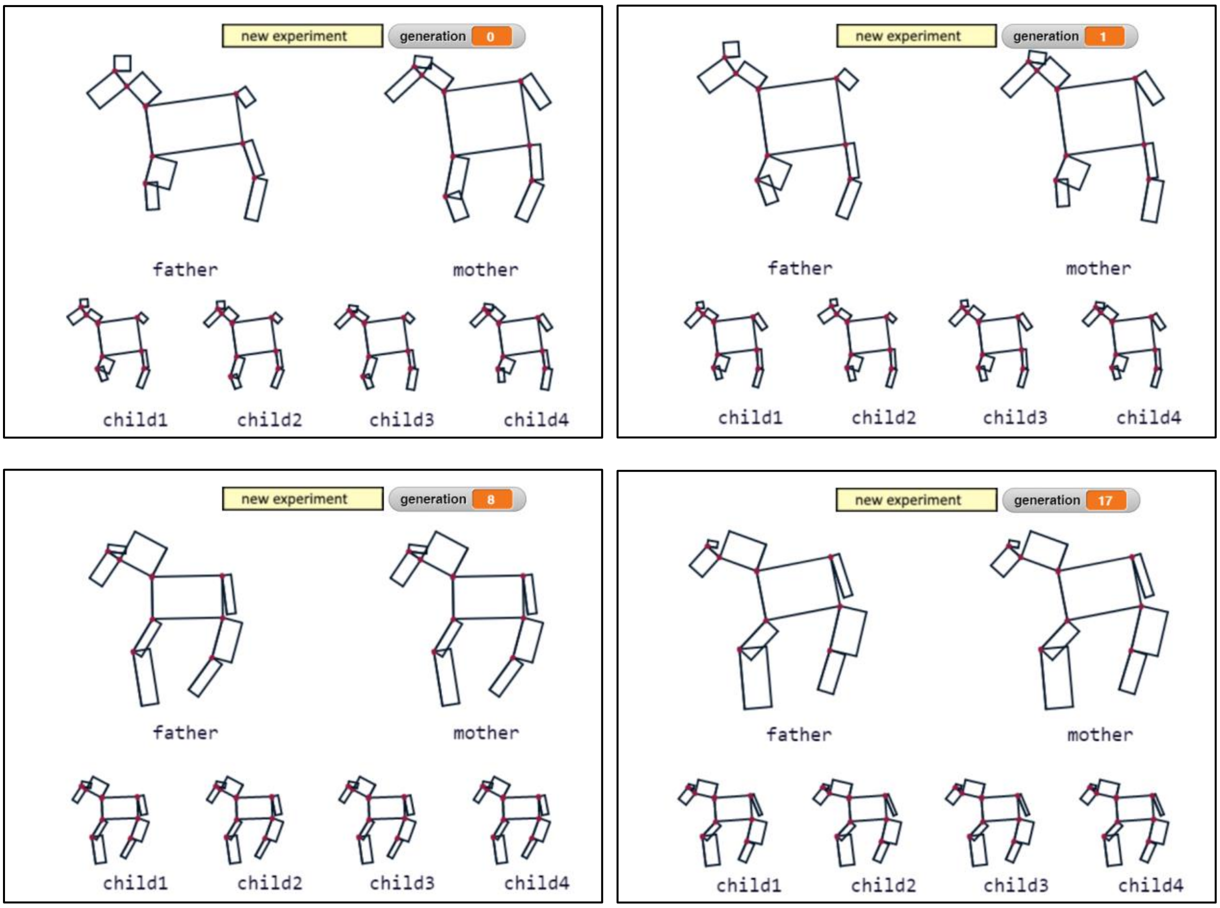


A *new experiment* is started by asking the *Animal* object to create two new animals as *father* and *mother*. They'll be crossed.



This is done accordingly with the children.

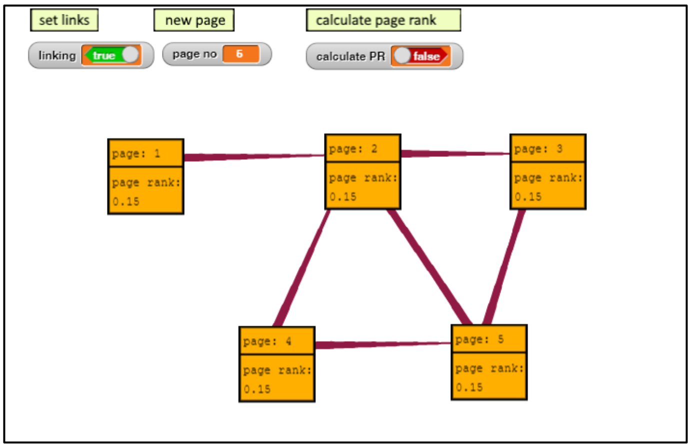
Let us try to breed "jumping ponies" with short tails. First, we create the parents and select candidates for ponies from the offspring.



Well - evolution is just unfathomable! 😉

17.5 Rate Websites: PageRank[83]

Level: *high school*
Materials: *Page rank*



If you know the addresses of websites, you can reach them directly via the net. But what happens when we search for pages with specific content? For this purpose, of course we use the search engines, which propose us to certain keywords network addresses from their tables of contents. These directories can be created by web crawlers automatically visiting as many accessible websites as possible, jumping from link to link, and adding the keywords found there to the table of contents of the search engine. This usually results in extremely extensive address collections for the same keyword.

Since users of search engines cannot handle large unordered address collections, the pages found for a keyword must be sorted according to their importance. Users then usually use relatively few addresses that appear first. The links below are hardly noticed. So at least the commercially operating providers on the net must be interested in appearing as high up as possible in the lists created by search engines in order to be found by potential customers at all. They use all tricks to achieve this.

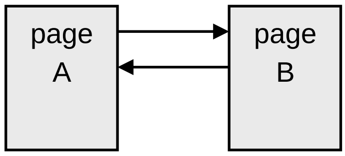
So far, nothing has been said about the meaning of a page's information for the keyword. Just showing up doesn't mean much. For example, if a page contains the text "*Nothing is written here about Goettingen*", it will still be included in the table of contents relating to the keyword "*Goettingen*". So, we need other evaluation criteria. In the simplest case, the authors of a web page enter keywords in the meta tags for the content of the page: *<meta name = "keywords" content = " Snap,School,Computer Science">*

However, this possibility is often abused by using frequently used keywords - which do not affect the page content at all - to direct potential "victims" to the site. Not very helpful is the idea to count how often the keyword appears on the page. In this case, web pages sometimes contain certain keywords "invisible", e.g. by writing the keyword very often in white on a white background. Of course, you can also have people rate websites and enter them in the search directories. But this is a very expensive and relatively slow way to create directories, and of course such an evaluation is subjective. It is also often difficult to evaluate pages with special content - e.g. from archaeology. In the worst case, the "value" of a page does not result from its content, but from the amount paid for the evaluation.

Another way to use the expertise of web authors for the evaluation of web pages on the one hand and to automate the evaluation process on the other hand is realized in the so-called *PageRank* procedure. Unlike the meta tags that evaluate your own website, links from one website to other websites are seen as a knowledge-based vote by which authors indicate that other websites contain interesting content. If someone refers to a page with physical content, the author will most likely understand something about the content. Moreover, since it is usually not known which other websites refer to their own, web authors can only manipulate this procedure with difficulty.

[83] based on E. Modrow: Technische Informatik mit Delphi, emu-online, 2004

The PageRank method does not evaluate all links equally. It determines a rank (the Page-Rank) for each known website, which describes the "weight" of this page. This rank is divided during the "vote" by links to all references leading away from the page. If a web page contains only one outbound link, then this receives the entire weight of the page, if it contains two, the weight is halved, and so on. (If the page does not contain an outgoing link, it will not take part in the vote. In the PageRank calculation, it returns the value 0.) The rank of a website increases if as many high ranked pages as possible refer to it and if these pages contain as few links as possible.



As a first example, let's choose two pages that mutually refer to each other. To calculate the PageRank of page *A* - *PR(A)* - we need the PageRank *PR(B)* of page *B*, because a link from *B* leads to page *A*. The calculation of *PR(B)*, however, again includes *PR(A)*. So, we need an old value of *PR(A)* to determine the new one. Since this argumentation can be continued, a method must be developed to reduce the influence of the old values on the calculation of the new rank, so that a stable result is obtained in the course of the calculations. This is achieved by multiplying the contribution of the incoming links by a factor *d* which is less than *1*. Since this is included in every calculation, the "very old" PageRanks are multiplied by d^n, a number that is increasingly approaching zero. For example, you select the value *0.85* for *d*. If we designate the times at which the PageRank was calculated in the past as t_1, t_2, t_3, …, whereby a larger index should mean an earlier time, then for both our web pages we get:

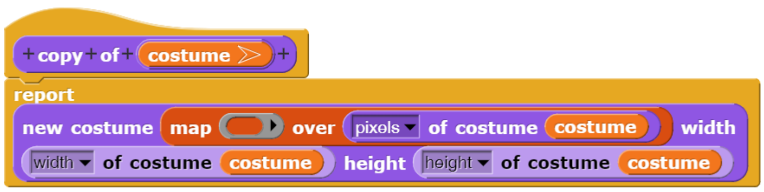
$$PR_{t_1}(A) = \ldots + 0{,}85 \cdot PR_{t_2}(B) = \ldots + 0{,}85 \cdot (\ldots + 0{,}85 \cdot PR_{t_3}(A)) = \ldots + 0{,}85 \cdot \ldots + 0{,}85^2 \cdot PR_{t_3}(A) = \ldots$$

If page *B* had more than one outbound link, we would have to divide its rank in the calculation by the number of links - *C(B)*. We must proceed accordingly with the other sites that have links to page *A*. If we call these *n* web pages T_1, T_2, …, T_n and replace the three dots in the above relationship with *(1-d)*, then we get the original formula that was initially given by Google for the page rank calculation:

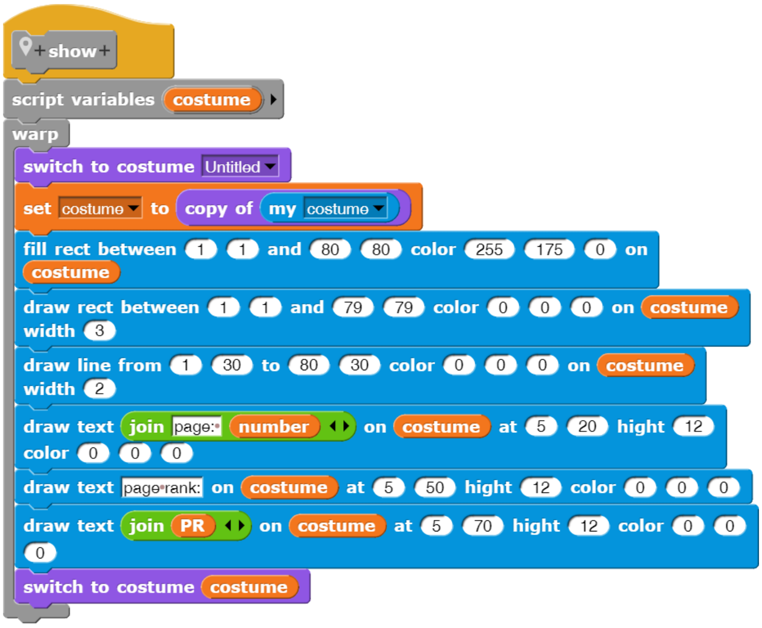
$$PR(A) = (1-d) + d \cdot \left(\frac{PR(T_1)}{C(T_1)} + \frac{PR(T_2)}{C(T_2)} + \ldots + \frac{PR(T_n)}{C(T_n)}\right),\ d = 0{,}85$$

The rank of a website is at least *0.15*. But what influence do the other terms have? We want to clarify the question with a simulation program in which symbolic web pages can be created and linked. The PageRanks can be calculated in a "website" created in this way.

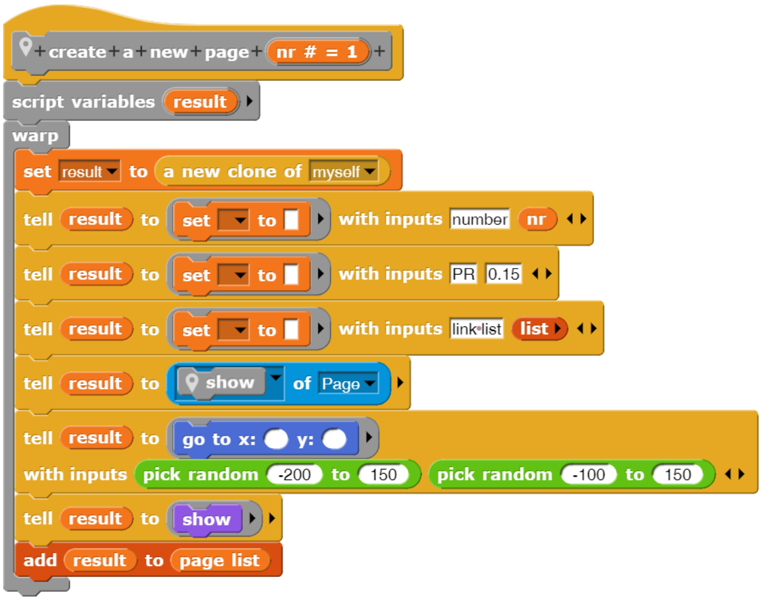
In our program, in addition to the buttons shown, which serve to control the functionality, we need the prototype of a "*Page*", which (here) should be a website, as well as a global list of all generated pages. Each page contains a *link list* with the numbers of the linked pages, a *number*, a PageRank *PR* and a help variable *PRnew*, in which the newly calculated PageRank is added up. Pages should be able to display themselves on the screen. Since this changes the costume, we better operate on a copy of the current version. A corresponding block can be written quickly.



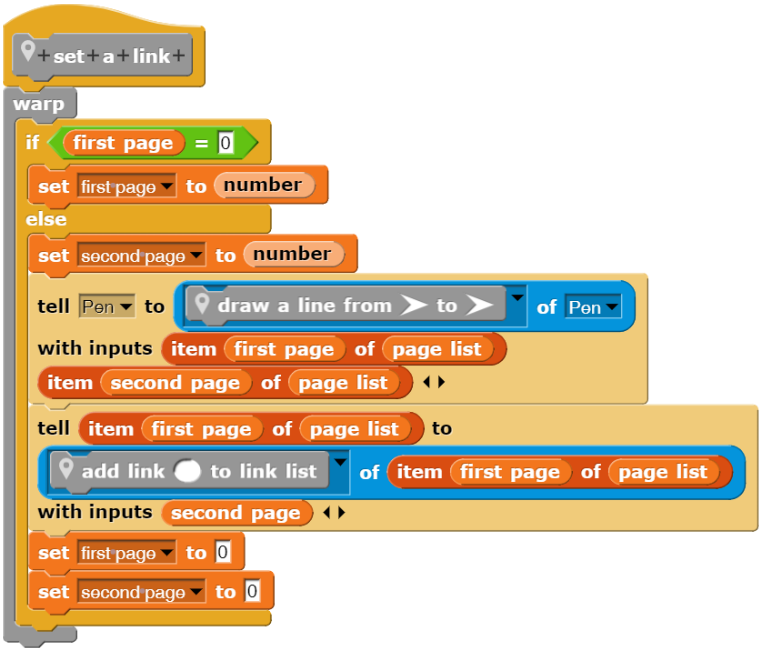
For the display of text and lines on the sprite we again use the appropriate graphics library.



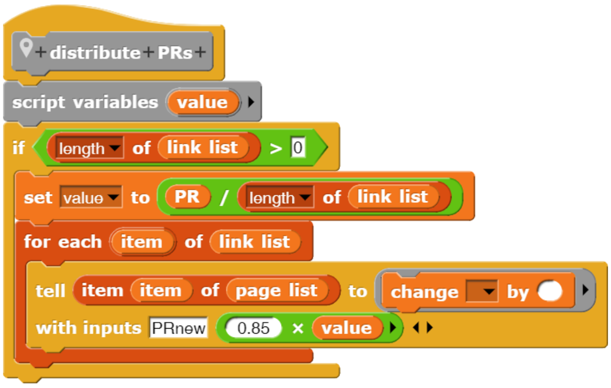
The most important task of the prototype is to create clones of itself. We save such a clone in a script variable *result* and ask it to perform the operations that produce the desired result through a sequence of commands. The generated page is added to the *page list*.

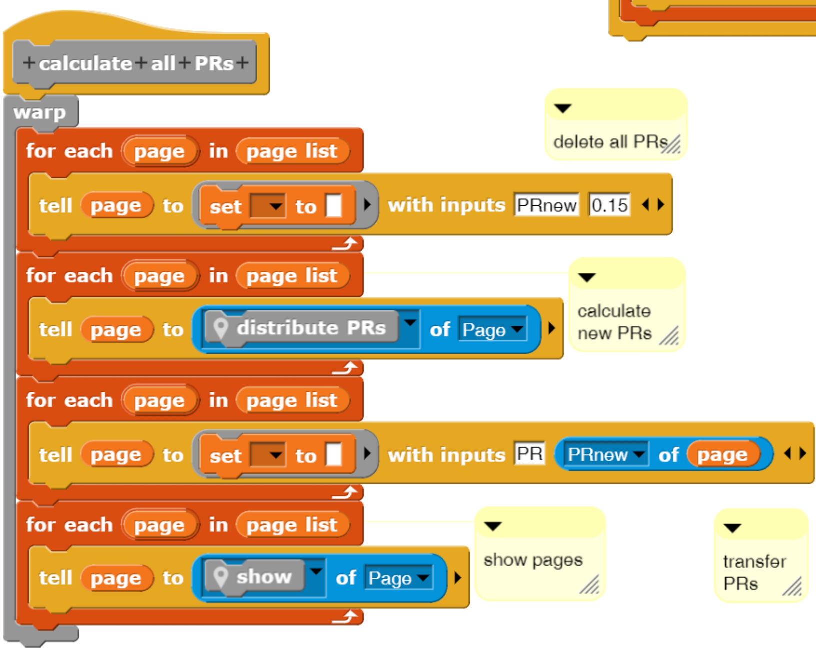


In the corresponding mode, pages are connected by clicking on two pages in succession. The numbers of the affected pages are stored in two global variables. Then the first one can be asked to "link" to the second one. The *Pen* draws a line between the sides that decreases in thickness, a kind of arrow. (Mutually connected sides thus maintain a connection almost the same thickness.) The second page is inserted into the link list of the first page.



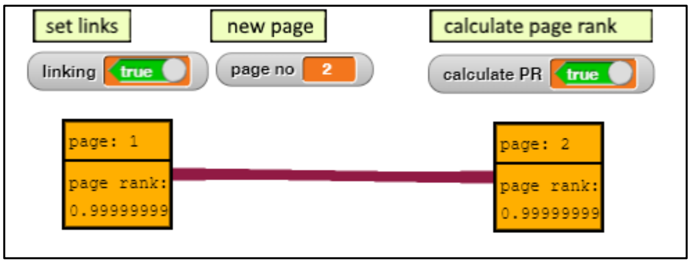
When recalculating the PageRanks, each page must distribute its current *value* to all connected pages. The page calculates this value and asks all pages of the link list to increase their auxiliary value *PRnew* accordingly.



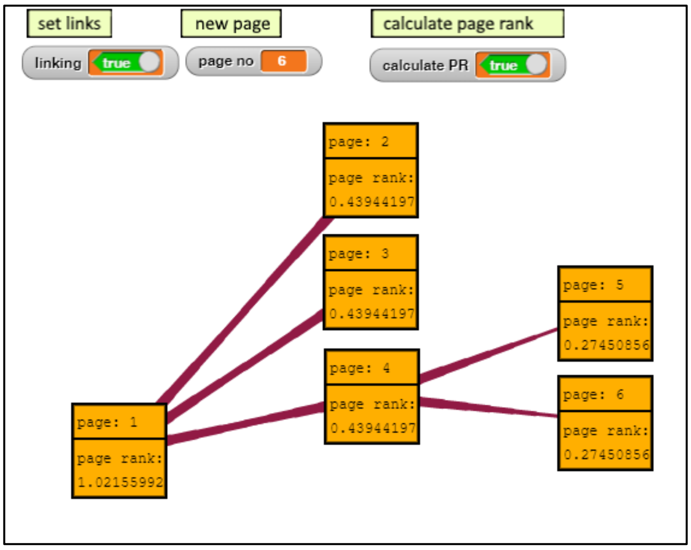


You can use these auxiliary methods to calculate the pageranks. First of all, all auxiliary variables of the involved pages are set to zero. Then all pages distribute their values to the connected other pages. When this is done, the auxiliary variables are copied into the PR variables and the pages are redrawn with the new values.

We now want to use our simulation program. We create two websites, link them and calculate the PageRanks. You can see that the values converge towards *1* (independent of the initial PageRank, by the way). This is of course no surprise, because this is exactly what we intended to achieve with the introduction of the "damping factor" of *0.85*.

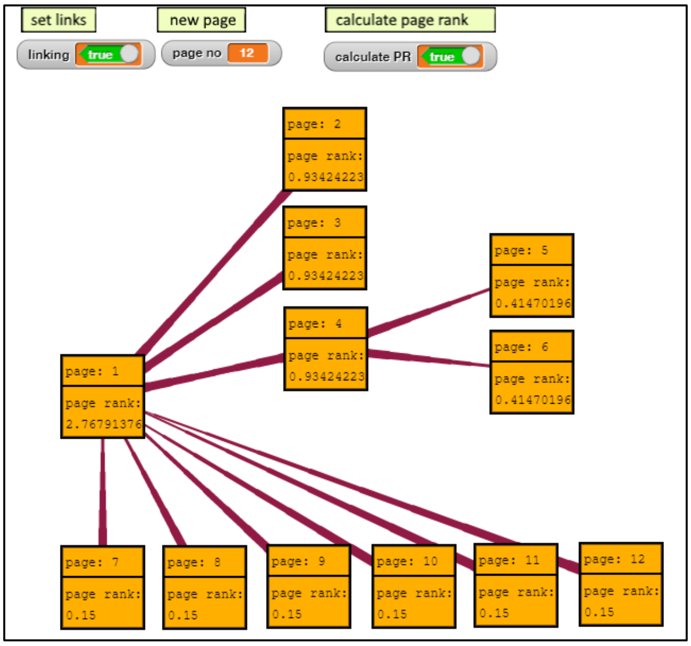


As next example we choose the structure of a typical homepage with a tree structure, which starts from an index page and branches to subdirectories.



We now assume that there are additional external sites that link to our homepage.

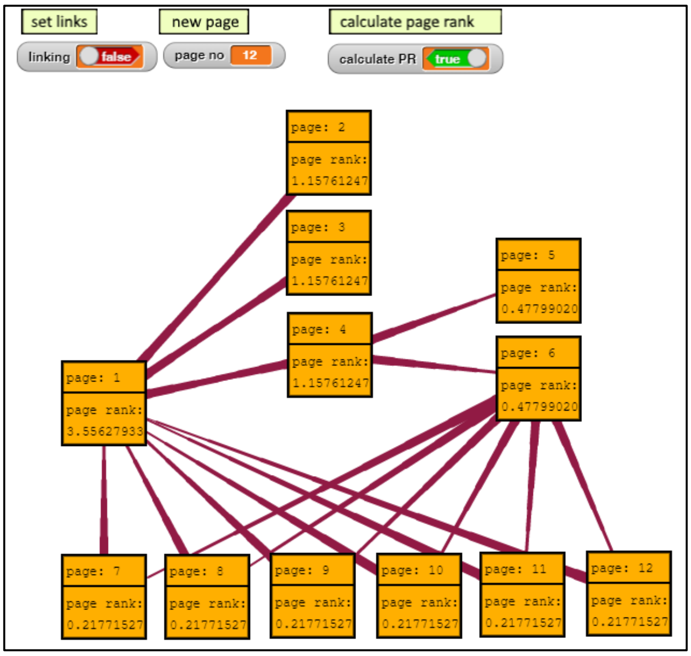
The PageRank of the homepage increases considerably, also the weight of the internal pages increases.



Finally, we want to assume that the external pages are again referenced in a link list of the homepage.

The rank of the homepage continues to rise. One can see how the importance of the pages is growing in a network of pages that mutually refer to one another in order to express their "respect" for one another.

The PageRank procedure is a technical process that can also be transferred to other, e.g. social systems.[84] However, it quickly leads to socio-political questions, because the focus is not on the content of the pages, but on their structure and functionality.



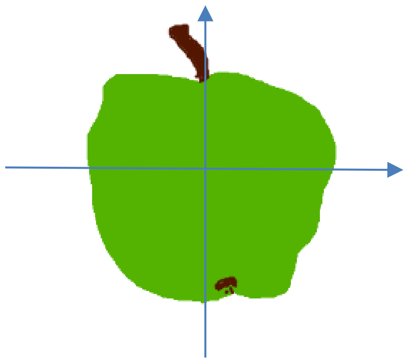
1. If the result of the PageRank calculation is decisive for the "visibility" of the pages[85], why are commercially oriented private companies allowed to decide on this visibility?
2. The intelligence of the system results from the expertise of those who have consciously set links in very different areas. Isn't the result actually a public good that should be available to everyone without some profit (and power) from it?
3. If only the PageRank would be decisive, the search results would always have to be arranged in the same order. Obviously, this is not the case: the results differ depending on the person who is looking for. They are filtered according to their interests assumed by the search engine. In extreme cases, you only get the results that you want to see - or that someone thinks you want to see - or that someone thinks you should see. The political consequences (keyword: "echo chambers") are currently under discussion

[84] There it even comes from: https://de.wikipedia.org/wiki/PageRank
[85] What only appears at the back is practically non-existent on the net.

17.6 The Smart Scale

Level: *high school* Materials: *Smart scale*

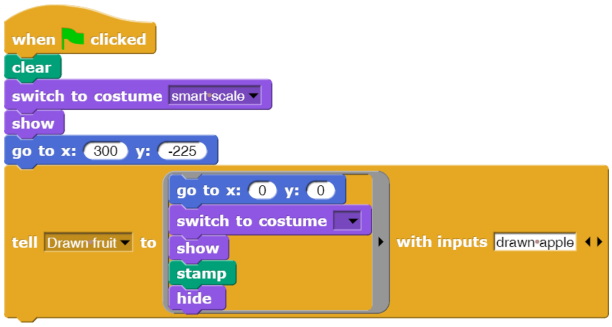
A sensation is looming in the supermarket: the fruit department has ordered an "intelligent" scale with a camera that is supposed to recognize and weigh fruit at the same time. Unfortunately, only the camera is included, the fruit recognition has to be implemented by yourself. The fruit department gets help from the scanner cashiers, because they have already done something similar earlier in this book.



First, we try to find some criteria to distinguish fruits. We draw an apple, an orange, an apricot, and a banana. The differences are obvious:

- apple and orange are round, the banana is long
- orange, apricot and banana are orange-yellow, the apple is (in this case) green
- the apricot is small, the others are bigger

But what do "round", "long", "yellow" and "green", "big" mean???



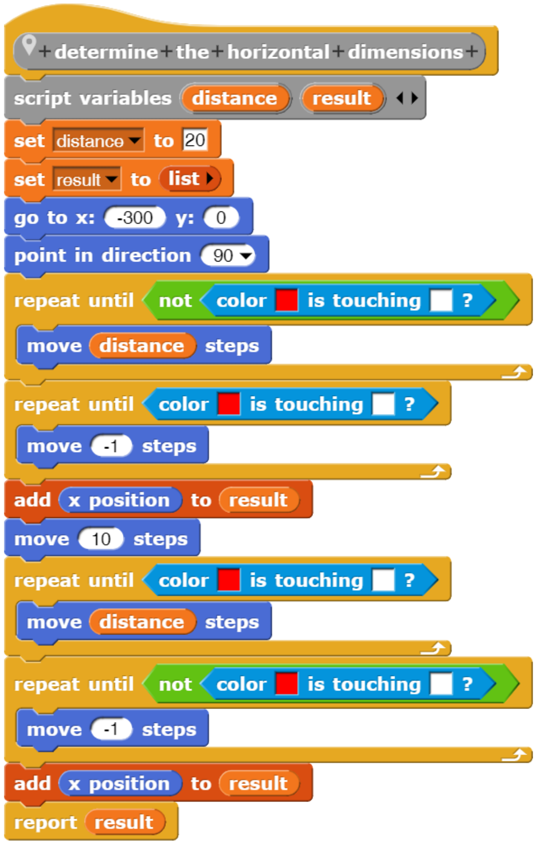
We know it, but the computer doesn't. We have to teach him.

We change the stage size to 800 x 600 pixels and bring the object with the costumes of the self-drawn fruit (*Drawn fruit*) to the center of the stage. There we ask it to assume the "*drawn apple*" costume and stamp it on the stage. After that it should hide.

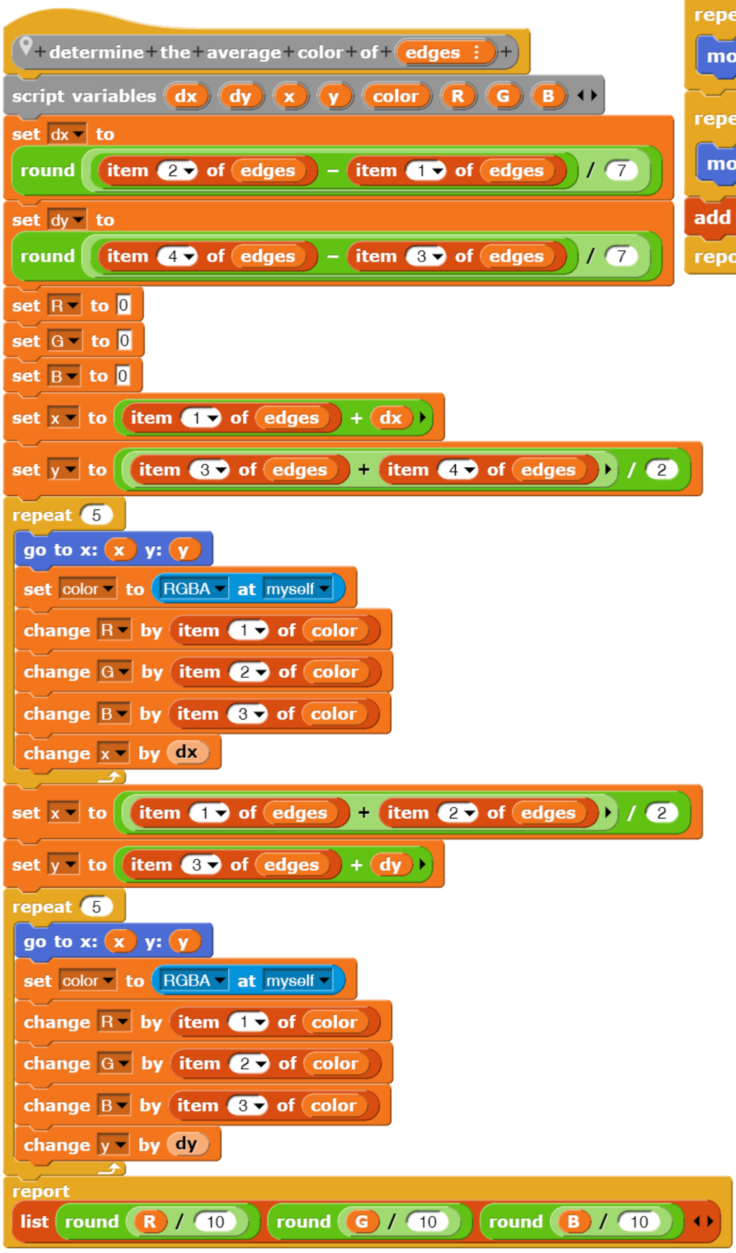
We then instruct a *Laser* to determine the properties of the currently visible fruit. For this purpose, it is to run across the image from left to right and from bottom to top, similar to the barcode scanner. In doing so, it measures the size of the object on these routes and calculates the ratio of the results. "*Round*" objects should have a ratio close to *1*, "*long*" objects a small value. For "*oval*" objects, we should actually use multiple measurement directions. But for now, "*oval*" means for us "*not round and not long*".

So that the measurement does not take too long, the laser makes larger steps until it hits the fruit. It then takes small steps back to the edge of the fruit and remembers its x-position. It does the same at the opposite edge.

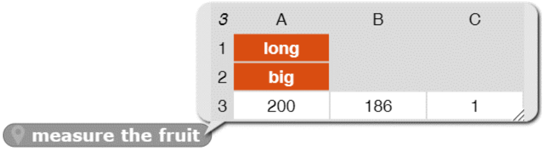
The *determine horizontal dimensions* - block of the *Laser* provides a list with two values: left and right border. Correspondingly, the *determine vertical dimensions* - block lower and upper limit of the object. With these results we can decide whether an object is round, long or oval. And we know its size.



The color of the object is still missing. We already know the limits within which the fruit is located on the stage. We pass this to a block *determine the average color of .....* In this block, the laser is sent to 5 points on the middle horizontal, determining the RGB values each time. The same is done on the mean vertical. After that we determine the average values of the color channels.



With these methods, the *Laser* can determine the characteristic properties of a fruit.

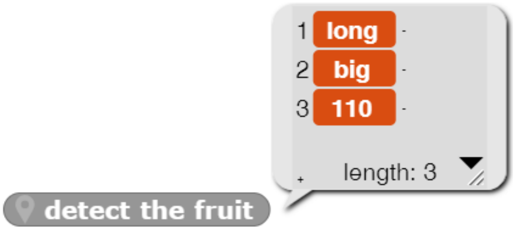


Normal fruits have different colors. But our RGB values can display 256 * 256 * 256 colors, so 16,777,218. That's a little too many. We need a method to reduce the number of colors.

We try this: for each RGB channel we decide whether the color value is "*high* or "*low*". If it is high, we set it to 255, otherwise to 0, so we only get two possible values for each channel, so 2 * 2 * 2 = 8 possible colors. With this procedure we try out whether we can see anything useful at all - or not.

It's looking good, isn't it?

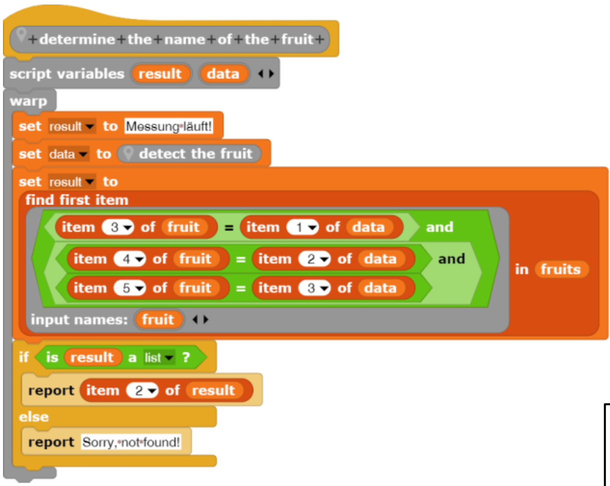
So, we can equip the smart *Fruit scale* with a method that asks the *Laser* to determine the fruit data.



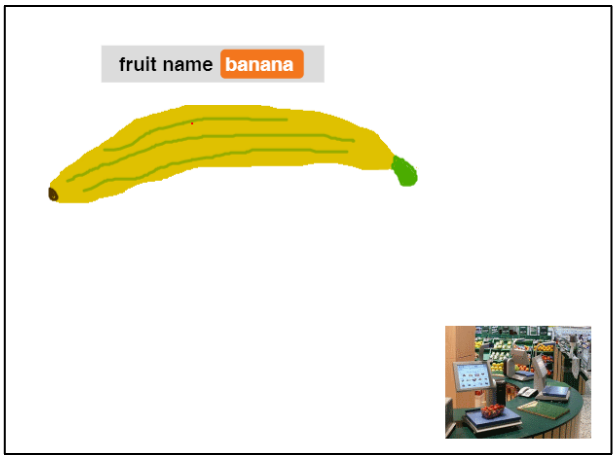
fruits

16	A	B	C	D	E
1	100	apple red	round	big	100
2	101	apple green	round	big	010
3	102	tomato	round	middle	100
4	103	orange	round	big	110
5	104	apricot	oval	middle	110
6	105	banana	long	big	110
7	106	cherry	round	small	100

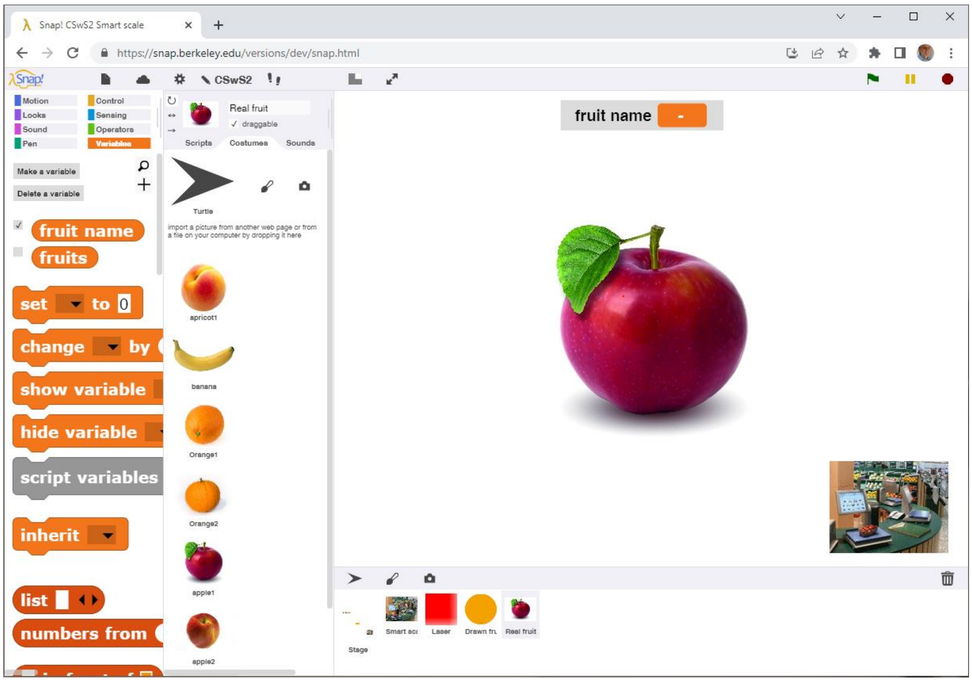
And we can use this result to compare the data with those of the stored *fruits*. These are to be present in a variable *fruits*, in which the *article number*, the *designation,* and the typical *fruit data* are stored.



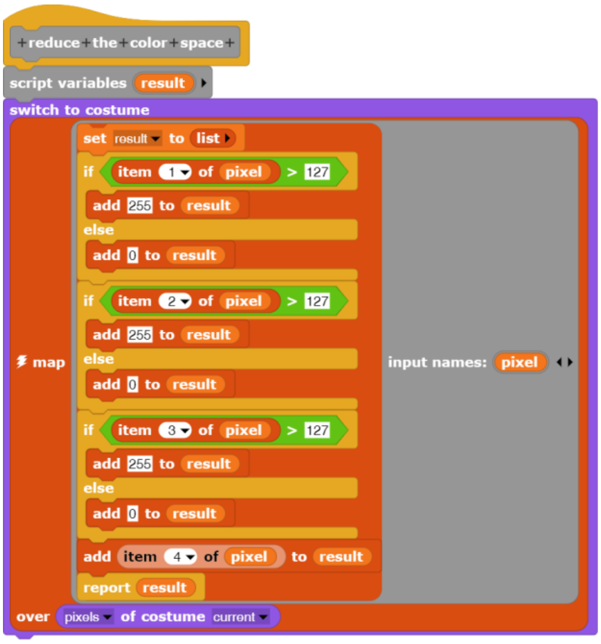
It's working!



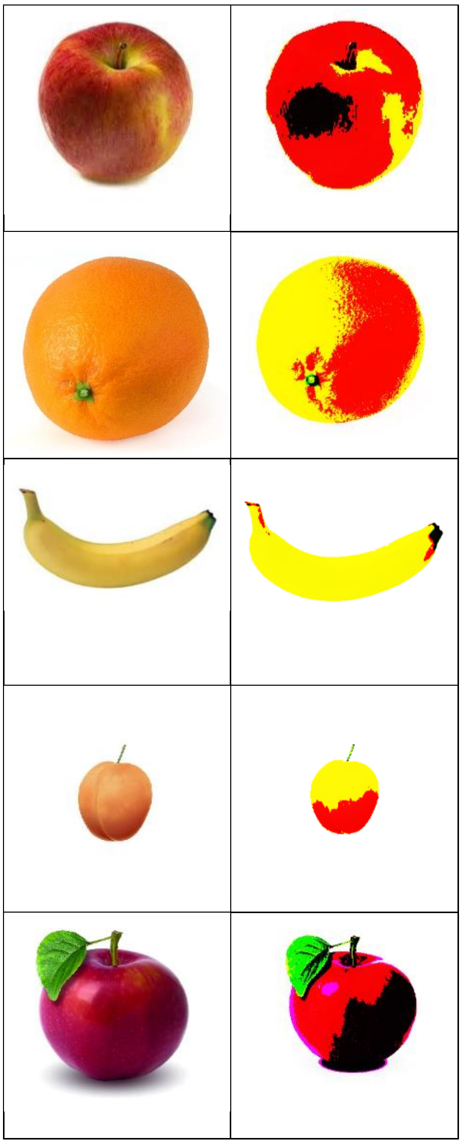
After these successes the crew of the fruit scale becomes courageous and tries to analyze real fruit pictures.



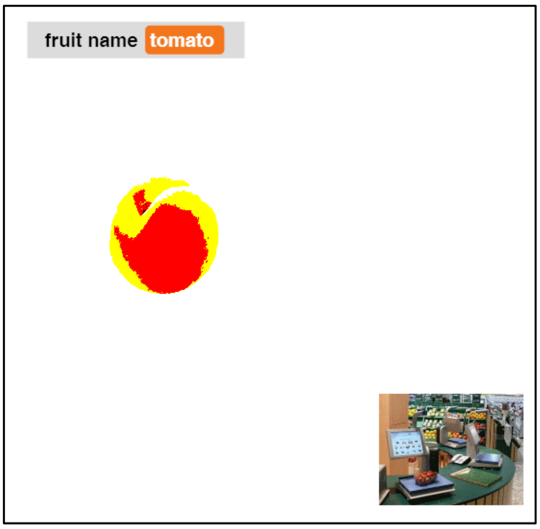
Their color spaces should also be reduced, similar to the drawn fruits. Then we get again a color reduced image on the stage.



We reduce the number of colors as described ...



... and stamp the image on the stage. After that we call the previously developed fruit determination again.



Okay, we should work on the entries of the fruit table as well. 😉

Now you have the full toolbox together for optical fruit determination:

1. Take a picture of a fruit and choose it as the costume of a sprite. You can take pictures with your smartphone or laptop camera. The background should be white.
2. Reduce the color space of the image.
3. Measure size and shape of the fruit.
4. Measure the mean color of the fruit and reduce it as well.
5. Calculate the color code of the fruit.

The obtained data *shape*, *size* and *color code* can be used as columns of a database table. We will have three different values each for size and shape as well as 8 possible color codes. This allows us to distinguish 3 * 3 * 8 = 72 fruits. Try a "real" intelligent fruit scale in a department store - we're not that bad. 😉

Tasks

1. Create a **database** table for fruits of the following type:

pnr	fruit	shape	size	color code
123	red apple	round	big	100
223	cherry	round	small	100
456	banana	long	big	110
…	…	…	…	…

2. Add the table to your database.

3. Write an **evaluation method** so that it provides the name and price of the fruit. To do this, use database commands.

4. The **color reduction process** is very coarse. Come up with a better way.

5. Our fruit recognition process only works well if the fruit is placed in the center of the stage and aligned horizontally. If we fit a sprite with a fruit picture as a costume, we can **center and align the Sprite** in the middle before we print the costume. Implement the procedure.

6. If we use a more detailed **color code**, we can distinguish more fruits. Would that be progress in any situation?

7. It could be that the background of the fruit is not white. Can you help?

17.7 License Plate Recognition

Level: *high school* Materials: *License plate recognition*

The success with the smart scale goes through the supermarket like a wildfire. It also reaches the security department. Among other things, it is responsible for the parking garage. To simplify the payment of parking fees, the department installs automatic license plate recognition. Registered customers with a customer card and automatic billing no longer have to stop in front of the parking garage barrier - at least that's the hope.

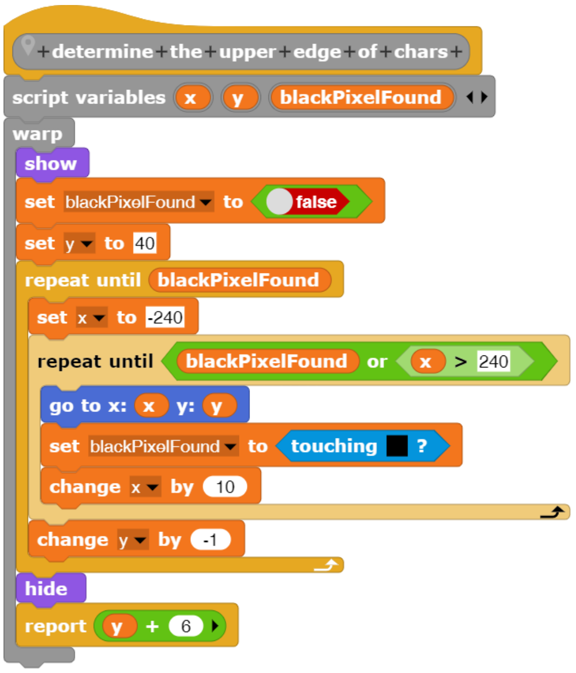
Car license plates contain special character sets that facilitate character recognition by computers. In Europe they have a black border - and that is good for us. So, let's try to determine the numbers on the plate. (We leave the other characters to you.) Fortunately, we have already realized almost all tools for our project. All you must do is ask the people at the smart fruit scale!

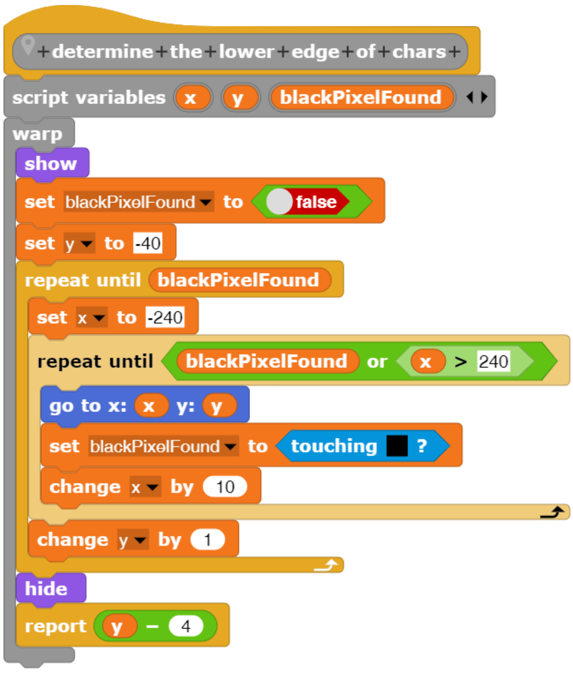
We are trying to develop an extremely simple method of license plate recognition. The result is very sensitive to changes in position and size of the license plates. But these disadvantages can be easily corrected by using a detailed measurement method. Take a look at the exercises!

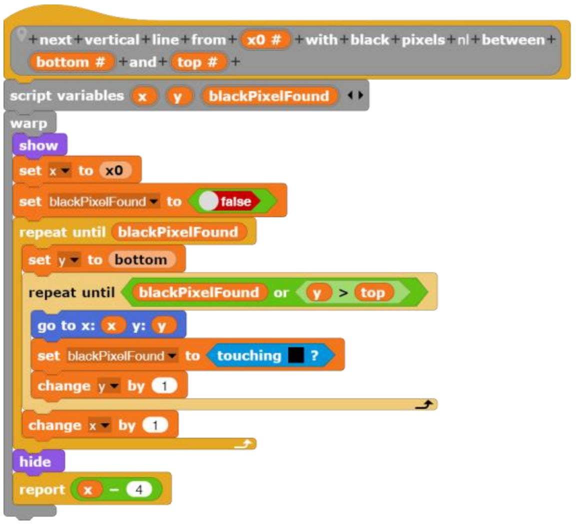
OCR (*Optical Character Recognition*) uses complex methods, often with neural networks, to recognize characters. Here we are inventing a simpler procedure that is similar to that of the smart scale. Because all our marks on the license plate are the same width, we can easily identify them once we have found the boundaries of the license plate. With the intelligent scale you can see how this happens. We continue to use their laser.

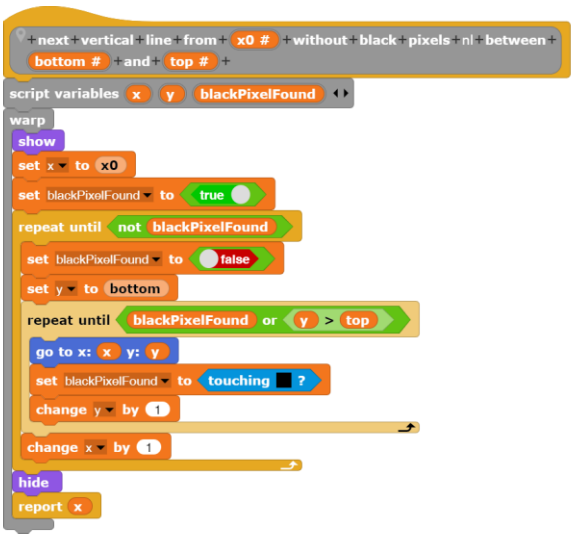
We can quickly generate license plates using various generators on the Internet. We save them as costumes of a sprite *License plate*. After clicking on the green flag, we stamp the costume onto the stage - as with the intelligent scale. The relevant area with the digits is then located between $-240<x<240$ and $-40<y<40$.

We start by searching the top and bottom of the license plate for lines that do not contain black pixels. Their positions indicate the upper and lower edge of the relevant characters. Then we search from left to right for vertical lines with black pixels. When we find the first one, we also have the beginning of the first character. Then we search for the first vertical line without black pixels. Their x-position is the end of the first character. We have a "window" with the first sign in it. The next line with black pixels gives the width of the gap between the characters.

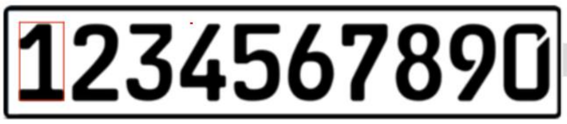


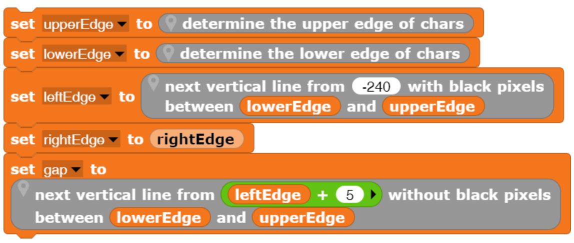






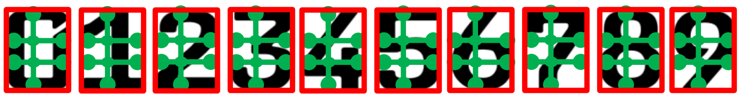
With the help of these blocks, we determine the boundaries of the first digit and the distance to the second. Then we draw the circumscribing rectangle in red.

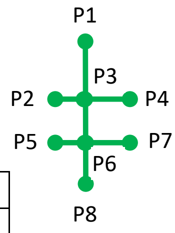




We can now mentally move this window over all characters of the license plate and try to recognize the characters within the field.

The number recognition itself is still missing. As a starting point we take the characters with the rectangle around. We imagine a "sensor field" consisting of three crossing lines. We measure the colors at the round points. We number the points as shown and look at the results in tabular form. (gray fields: result difficult to predict).



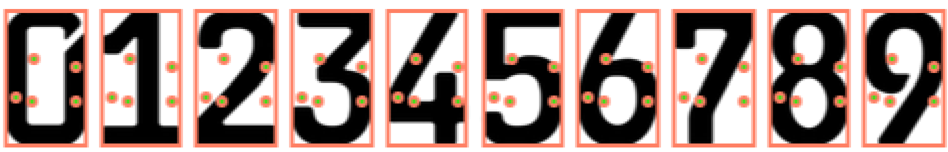


char	P1	P2	P3	P4	P5	P6	P7	P8	code
0									00100100
1									01111110
2									01101010
3									01011100 01111100
4									11010001
5									00001100
6									0100100
7									01111010
8									00010100 01010100
9									00101100 00101110

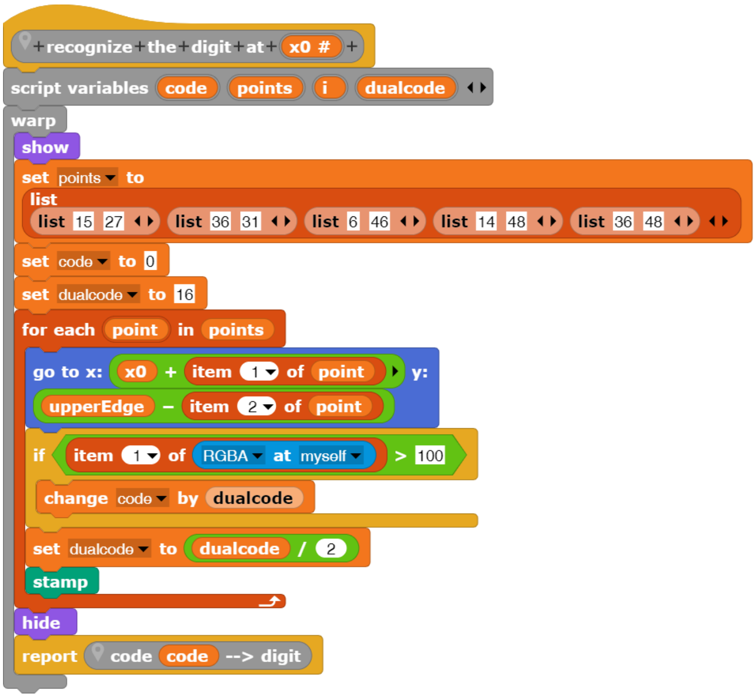
Errors may occur with characters *3*, *8* and *9* if the points are not very well adjusted. But that doesn't matter, because if we move the sensors *P2*, *P3* and *P7* a little bit so that they provide clear values, we can even do without the sensors *P1*, *P2* and *P8* (e.g.) and still have a usable code.

char	P1	P2	P3	P4	P5	P6	P7	P8	code	value
0									10010	18
1									11111	31
2									10101	21
3									11110	30
4									01000	8
5									10110	22
6									00010	2
7									11101	29
8									01010	10
9									10111	23

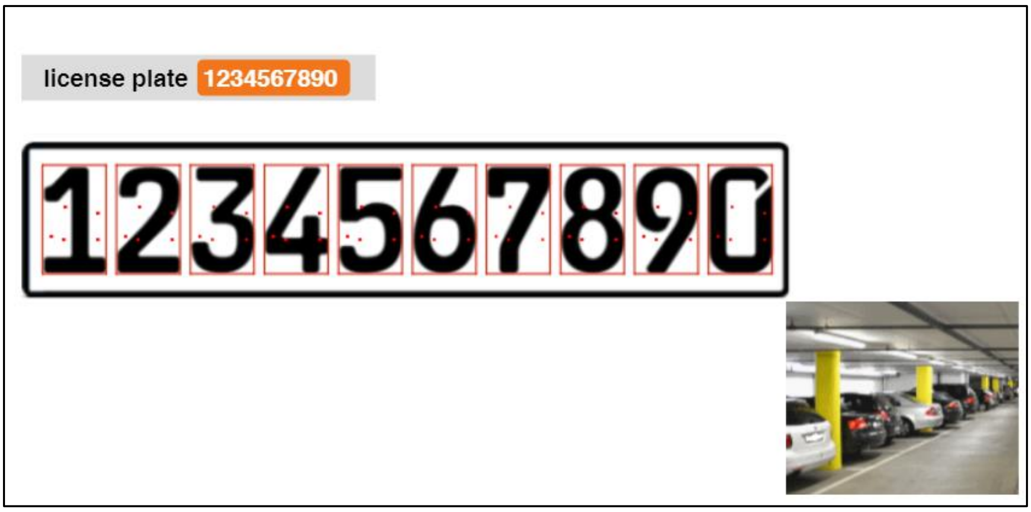
A possible layout for the remaining sensors would be:



We choose a license plate with all ten characters. The sensors are placed in suitable places (here: (14|24), ...) and stored in a list to read the colors in the character window at the positions and to form a code number from the colors interpreted as a dual code. When we're done, we transform the code into the right character.



Now the security department can ask the laser from their office in the car park which car has just arrived:



The result is particularly impressive for the advertising department, which immediately sees completely new applications for the process. Everyone's very proud of the security!

Tasks

1. Character recognition in the examples is very simple, but very sensitive to changes in the size and position of the license plate. Use **more sensors** to detect the characters more reliably.

2. **Extend character recognition** to the entire character set for vehicle license plates.

3. Character recognition programs can **learn**. If the script does not find any recognizable patterns, it should display its result and ask for the correct character. Save the patterns and the corresponding characters in a database table. Use queries to identify unknown patterns.

4. If you want to read **dirty license plates**, you won't find any sharp character boundaries. As a result, some sensors will produce errors. Improve the results in such cases by determining the "next correct code" of an incorrect code.

5. The recognition of dirty plates can be improved by converting the color image to a pure black-and-white image and **closing the gaps** caused by the dirt. Find out about suitable procedures for this purpose and implement one of them.

6. The security department needs a **database** of license plates and vehicle owners and their status (customer, company member, unwanted person, external parker, etc.). Can you help?

7. The license plate recognition turns out to be a great success for the security department. All its members are very proud of it and the other members of the company admire the "sheriffs". The **advertising department** now wants to use the data from the license plate table to honor customers as VIP customers who are frequently and for a long time present in the supermarket. These have special parking spaces near the elevator. Write a query to find VIP customers.

8. After some time, the **VIP parking** lots are occupied by pensioners and unemployed. Therefore, the advertising department extends the criteria for VIP customers by a minimum of turnover with their purchases. Because almost all customers use credit cards for payment, this is no problem. Improve VIP customer query accordingly.

9. The advertising department finds that it would be helpful to know not only a customer's turnover but also what they have bought. If it knows the interests of customers, it can provide them with **special offers** and special prices. Determine the additional tables required for this and their columns in the database. Write suitable queries.

10. The advertising department wants to know whether its advertising activities are successful. Do they **reach customers**? Try to answer these questions based on the stored data.

How To …

Index