Aalborg Universitet

AALBORG UNIVERSITY

Programming Concepts in Playful Programming Products

Allsopp, Benjamin Brink; Ejsing-Duun, Stine

Published in:
Proceedings of the 10th European Conference on Games Based Learning

Creative Commons License
Unspecified

Publication date:
2016

Document Version
Version created as part of publication process; publisher's layout; not normally made publicly available

Link to publication from Aalborg University

Citation for published version (APA):
Allsopp, B. B., & Ejsing-Duun, S. (2016). Programming Concepts in Playful Programming Products. In *Proceedings of the 10th European Conference on Games Based Learning: ECGBL 2016 Proceeding* (pp. 1-10). University of West Scotland.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal -

Take down policy
If you believe that this document breaches copyright please contact us at vbn@aub.aau.dk providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from vbn.aau.dk on: May 02, 2026

Programming Concepts in Playful Programming Products

Benjamin Brink Allsopp[1, 3], Stine Ejsing-Duun[2, 3]
[1]Department of Learning and Philosophy, Aalborg University, Copenhagen, Denmark
[2]Department of Communication and Psychology, Aalborg University, Copenhagen, Denmark
[3]Research Lab: It and Learning Design (ILD)
ben@learning.aau.dk
sed@hum.aau.dk

Abstract: There is wide recognition that teaching children to program is immensely important. A new digital divide is potentially defined in terms of competencies to create programs rather than access to computers. In this context perseverance and motivation needed to acquire programming skills are gatekeepers as are appropriate learning materials. Thus a new category of software products that attempts to turn learning to programme into play is of special interest. These playful programming products are the focus of our research program trying to bring perspective to their potential. A mid level goal of this program is to comprehensively compare different products against a stable list of skills and understandings supported by the category of playful programming software. This paper aims to initiate systematic development of this list of skills and takes inspiration from Brennan & Resnick (2012) where a model of computational thinking is held up against features of the visual programming language Scratch. However, we refine this model by assessing another playful programming product against it. The changing model is represented in a series of concept specialisation maps to support greater overview and transparency. A final relatively stable map is discussed with respect to the larger research projects goals.

Keywords: Playful programming, Programming concepts, Concept specialisation maps.

1. Introduction

Learning to program and computational thinking is receiving growing interest in education research (e.g. Guzdial & Soloway, 2008; Resnick et al, 2009; Repenning, Webb & Ioannidou, 2010; Brennan & Resnick, 2012). This interest is justifiable in terms of addressing a new digital divide between the programmer and the programmed (Rushkoff, 2010), or between the producer and the consumer (Gold, 2014) with potentially greater social, cultural, economic and political significance than the divide defined by access to media and computers (Selwyn, 2004). Unlike the previous digital divide, the new divide is not addressable by decreasing production costs of devices or more equitable distribution of devices. Programming requires complex abstractions, overview and mathematical insight (Misfeldt & Ejsing-Duun, 2015). Learning to program is a demanding activity that requires perseverance and motivation. Robins, Rountree & Rountree (2003) review extensive literature on teaching children and adults to program, we focus exclusively on playful programming.

1.1 Playful Programming

In recent years a number of computer games, robotic toys, programming apps and other software products (and even board games) have been released that attempt to turn learning to program into play. These playful programming products include Scratch, Lightbot, Hopscotch, Code Monkey Island, Kodable, Robot Turtles, Code Combat, Cork the Volcano, Codemancer, Machineers, Bee-Bot, CodeSpells, LEGO Mindstorms, LEGO Bits and Bricks, LEGO WeDo, Human Resource Machine and more. Products that claim to shortcut the path to programming are interesting, but we lack clarity on what they can do. They differ in their underlying didactic and pedagogical strategies and in the aspects of computational thinking they aim to teach, but no detailed overview of their differences exists. For the educator interested in choosing a product, this lack of overview is challenging. There are so many products that it is not convenient to try them all, and the developers own descriptions of the products are often hyperbolic and hard to compare. Furthermore, a clearer understanding of the underlying programming concepts and their relations can improve the design of playful programming products in the future.

1.2 Goals and scoping

This paper is part of *Playful Programming* - a larger project with the goal of bringing together different stakeholders (developers, educators, parents, learners and researchers) with a common vocabulary for

describing, developing, teaching with and comparing products. Central to this attempt is the examination of existing products to determine their features. In broad strokes we can imagine distinguishing products based on how they relate to the categories: *who*, *why*, *how* and *what*. The *who* covers who can benefit from using the product, the *why* covers their motivation (inside and outside the game), the *how* covers the didactic approaches used to facilitate learning, and the *what* covers what aspects of computer programming can be learnt using the product. In this paper we focus on what can be learned using different products. We imagine comparing a large number of products against a list of skills and understandings, but for now we want to achieve a greater level of impartiality and detail with respect to creating this list. Specifically, we are concerned that lists of skills and understandings can be compiled in many different ways to favour specific products. Skills and understandings can be highly ambiguous and overlap in confusing ways.

1.2.1 This paper's goal: a list of programming concepts

The goal of this paper is to produce a relatively stable candidate list of programming concepts exercised in existing products, which can in later work be used in a further process assessing a large number of products. Within this wording is embedded a number of decisions. To keep the list relevant for comparing products we want the list to be based on actual products rather than an independent assessment of the actual skills and understandings needed to program. However, this suggests a circular situation where we want to create a list of skills and understandings that can help us assess all products, yet we need to assess all products to ensure that we have a complete list. When we aim for *a relatively stable candidate list* this is because we conceptually differentiate between a later process of assessing the majority of products while occasionally updating the list and the current process of developing most of the list by examining two products: Scratch and Lightbot.

Other decisions in scoping this paper involve working with a reduced breadth of supported skills and understandings. Firstly, we will not concern ourselves with skills and understandings that may be relevant for being a good programmer, but also relate to many other creative processes. Secondly, we will only concern ourselves with skills and understandings that are obvious from observing the product directly. For now we want to focus on what the products offer and thus for now avoid observing user engaged in the tool. Thirdly, this initial mapping of skills does not include levels of learning, where the same issue must for example be considered in terms of whether the user can for example use it, understand it or define it, which will be relevant at a later point. Together these decisions have focused this work on what we call "programming concepts" and correspond to Brennan & Resnick's (2012) description of computational concepts, which are: "[…] the concepts designers engage with as they program" and "[…] common in many programming languages".

Finally, to achieve less ambiguity and untangle some of the overlaps between programming concepts we will adopt a new approach to representing the different concepts as they relate to each other in concept specialisation maps. Concept specialisation maps will be introduced in the research approach section below.

2. Our Approach

In brief, our approach involves starting with a provisional map of concepts based on the computational concepts described by Brennan and Resnick (2012), examining a product to identify how these categories suffice to describe the computational concepts that the product addresses. When a programming concept or area of concepts are not satisfactorily captured in the map, it is modified before further exploring deficiencies of the map. The approach resembles other processes of research and design inquiry, but the specific methodological foundation of this work is inspired by Lakatos' (1976) philosophical work on mathematical proofs. Here science is championed by inviting falsification through facilitating greater social scrutiny. The research/epistemological value of this work is the high level of transparency in developing a model. We aim to achieve this level of transparency through the use of concept specialization maps to present iteration of our model.

2.1 Concept specialisation maps (CSM)

Concept specialisation maps (CSMs) are a form of diagram introduced (or perhaps reintroduced) here to overview how concepts are related and to help make subtle distinctions between different concepts spatially apparent. They are shown throughout this paper (Figures 1, 3, 5, 6, 7 & 8) and superficially resemble, but are distinct from maps showing learning prerequisites (Nagarjun, 2009). Both are directed acyclic graphs based on

nodes and arcs between these. But where arcs in learning prerequisites maps show how learning one thing is dependent on first learning another thing, arcs is CSMs show how one concept is a special type of another concept. Thus a programming concept labelled “Procedures with flow control” has a specialisation sub concept labelled “Procedures with loops” because loops are one kind of flow control. Unlike Generalizes/Specialises arcs in Unified Modelling Language (UML) use case diagrams and class diagrams (Rumbaugh, Jacobson & Booch 2005), the specialisation arcs in CSMs are drawn without arrowheads because the direction of specialization is always apparent from the graph’s layout. There are potentially other uses for CSMs and especially interactive/dynamic CSMs in the presentation of programming concepts to wider audiences, but for now (in our research process) CSM is exclusively considered an epistemological tool towards identifying a list of programming concepts that will later be used in a matrix to compare many different products.

2.1.1 Our process

The plan for developing a relatively stable candidate list of programming concepts is to start with Brennan & Resnick’s (2012) peer-reviewed model of computational concepts, which is revised and mapped out in a CSM. We then systematically work through a product asking what programming concepts do they really exercise (as opposed to what they only purport to exercise). Depending on the clarity of the observations we either update the map or consider potential updates that need further confirmation. A benefit of CSMs is that they allow us to not just add new concepts, but also to visualise concepts that partially overlap. These unclear distinctions may be untangled by recognising that they are both subtypes of a common parent concept. This involves identifying potential parents in the other displayed concepts or including new concepts as parents in the map where they relate to other concepts. Each time the map is updated it is necessary to consider if the product supported concepts accommodated by the previous version of the map are still accommodated in the revised maps, and adapt accordingly. It is the opening of some of these critical reflections to peer scrutiny that constitutes the scientific foundation for this paper, and it is CSMs that allow us to consider more complex models of programming concepts than would be suited for description in prose text alone.

3. Our Initial Map

Our goal is to develop a CSM by testing it with a product that it was not initially designed for, but to do this we need an initial map. This section first introduces Brennan and Resnick’s existing model of computational thinking, which they developed in relation to the Scratch programming environment. At first we modify it to fit the scoping of this article and display it as a CSM. After this we briefly examine Scratch ourselves and modify the CSM more substantially. This is partly to exploit CSMs ability to show connections between concepts.

3.1 The model of computational thinking

Brennan and Resnick identify three main types of computational thinking covering: “[…] computational concepts (the concepts designers engage with as they program, such as iteration, parallelism, etc.), computational practices (the practices designers develop as they engage with the concepts, such as debugging projects or remixing others’ work), and computational perspectives (the perspectives designers form about the world around them and about themselves)” (Brennan & Resnick 2012, p. 1). While all of these are interesting for the playful programming project we are here interested in their computational concepts shown as a CSM in Figure 1. Brennan & Resnick provide short descriptions of each specialisation, which they elaborate through examples of Scratch programs using the concepts.

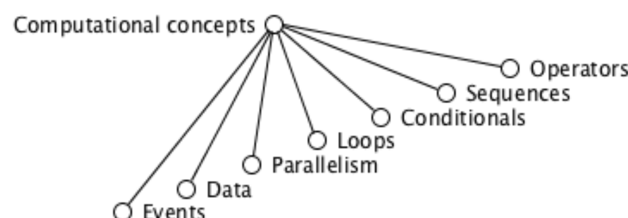

Figure 1. A CSM showing Brennan and Resnick’s specialisation of *computational concepts*

3.2 Scratch

Scratch is the earliest playful programming product we have considered. It is an interactive environment developed by the Lifelong Kindergarten research group at the MIT Media Lab. It allows young people to create their own programs in the form of interactive stories, games, and simulations (Resnick et al., 2009). The central programming activity involves snapping blocks representing instructions together into scripts to determine the behaviour of entities (sprites) in the program (stage). Available blocks are organised in a number of groupings including: Motion, Looks, Sound, Pen, Data, Events, Control, Sensing and Operators as shown in Figure 2. The metaphor used for Scratch is theatre, thus the program is happening on a stage, the programs are scripts, and sprites can be dressed with costumes (appearances) etc.

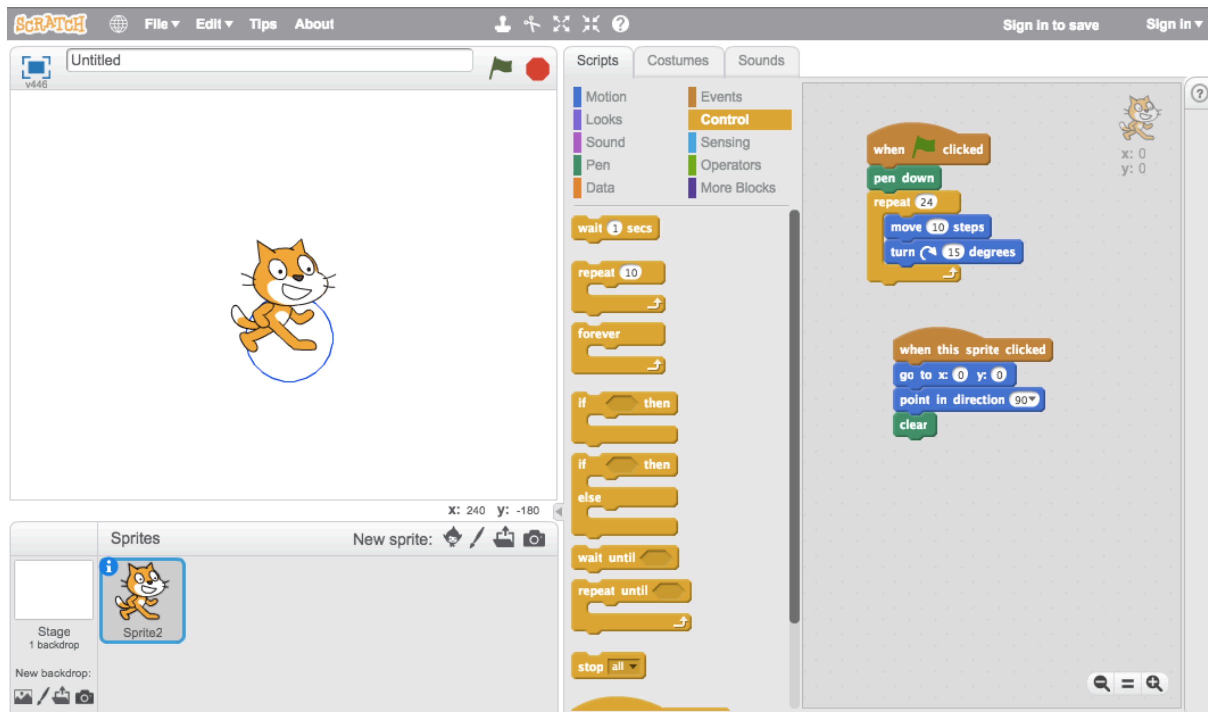

Figure 2. The Scratch programming environment

3.2.1 Our initial adaptions of the model.

Examining Scratch, the CSM from Brennan & Resnick model seems to match well. It was easy to confirm that there were blocks, or ways of using blocks for all of the computational concepts. However there were a number of details that warranted inclusion in the CSM. We did the following:

1. Changed the name of *computational concepts* to *programming concepts*. "Computation" is a broad term that covers non-computer related skills (e.g. calculating) as well as computer efficiency concerns not relevant in the context of playful programming that we focus on.
2. Changed the name of *data* to *variables* and added two specialisations. Here we found the term "data" too broad; Scratch only supports creating simple variables and arrays under the data grouping. We drew both as specialisation of the *variables* concept.
3. Added *encapsulation*. Brennan & Resnick (2012. p. 9) discuss abstracting and modularization as a computational practice and characterise it as "building something large by putting together collections of smaller parts". It seems that this practices must be supported in terms of a programming concepts. Encapsulation allows you to abstract away the complexity of some code and make it available in other code by calling its name.
4. Added *objects* both as a specialisation of encapsulation and of variables. Scratch supports sprites that are objects in that they encapsulate functionality and other variables and can be considered a special type of variable.
5. Replaced a generic *parallelism* concept with two specialisations of the *objects* concept: *objects with parallelism across objects* and *objects with parallelism within objects*. Brennan and Resnick (2012, p.

4) identify parallelism as a core computational concept, however they only describe parallelism in the context of objects: “Scratch supports parallelism across objects. For example, a dance party scene might involve several characters dancing simultaneously, each with a unique sequence of dance instructions. Scratch also supports parallelism within a single object. […] the Scratch cat has been programmed to perform three sets of activities in parallel […]”.

6. Added *arithmetic* and *conditions* as specialisations of the *operator* concept. Both types of blocks can be seen under the operator grouping in Scratch and operate on values to produce new ones. Arithmetic operators produce numbers and conditions produce *true* or *false*.

All of these changes to the CSM are emphasised in Figure 3.

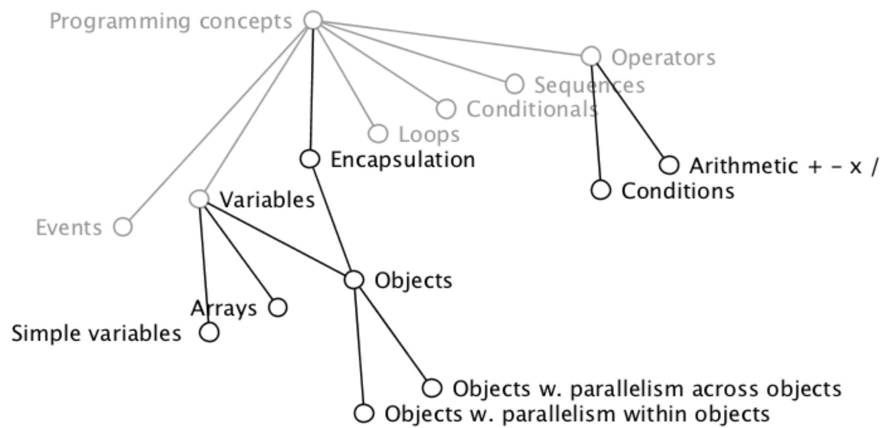

Figure 3. Multiple initial changes to the CSM

4. Observations and Analysis

In this section we will describe a number of considerations leading to changes in our map of supported programming concepts. These considerations arose while trying to use the initial CSM to evaluate another product than it was developed around. We choose Lightbot because it seemed to represent a different category of playful programming products than Scratch. We will discuss this distinction further in the discussion, but while Scratch can be thought of as a visuospatial authoring tool, Lightbot seems more like a traditional navigation game. In choosing a product form a different category, we hoped to further challenge the Scratch bias of the initial CSM and consider more changes. Our considerations are grouped according to the two product versions where they became apparent: Lightbot Web and Lightbot 9+. Lightbot Web is shown in Figure 4, but both have the same basic appearance.

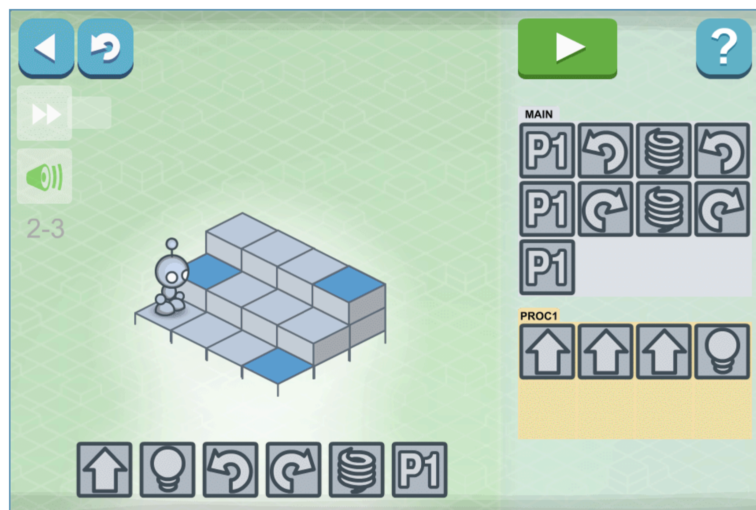

Figure 4. A level 2 screen from Lightbot Web

4.1 Lightbot Web

Lightbot is promoted as part of the hour of code initiative (https://code.org/learn) and was our first choice as an archetypical example of a playful programming product of the navigation game category. It consists of a series of screens (grouped in levels) that each presents a grid with an obstacle courses for a robot to navigate. The challenge for the user is to give the robot all the necessary instructions in advance. The user drags a number of instruction blocks into an area called *main* as shown in Figure 4. While we were attracted to Lightbot for its minimalist user interface and simple game metaphor, we were intrigued with a claim (Biggs, 2013) that: "Lightbot offers an easy way for kids to learn concepts like loops, if-then statements, and the like without typing or coding." Would we really see looping and conditionals? Lightbot has versions for different age groups and one web version. We played the entire web version while comparing it with and modifying the CSM.

4.1.1 Procedures and procedures calling procedures

When Lightbot Web reaches level 2 the interface shows an additional area for adding instructions. The area labelled "Proc1" (visible in Figure 4.) allows the user to create a procedure comprising multiple instructions that are called from the main instruction area with a block labelled "P1". Our initial CSMs did not include procedures because Brennan and Resnick's model did not include them, but when now considering this programming concept it is obvious that Scratch also exercises it every time a user creates a script. We added *procedures* to the CSM to accommodate this feature of Scratch, however with the Proc1 field Lightbot is exercising something more than Scratch does; it is allowing one procedure to call another with the P1 block. To accommodate this in the CSM we added a specialisation of the *procedures* concept called "procedures calling procedures" emphasised in Figure 5. This concept is reinforced later in Lightbot when an additional area labelled "Proc2" is shown and can be called from the Proc1 area with the block labelled "P2". At this point it also became apparent that procedures calling procedures is also a special type of our encapsulation concept because the code of the second procedure is run simply by calling its name. An arc representing this relationship is also emphasised in Figure 5.

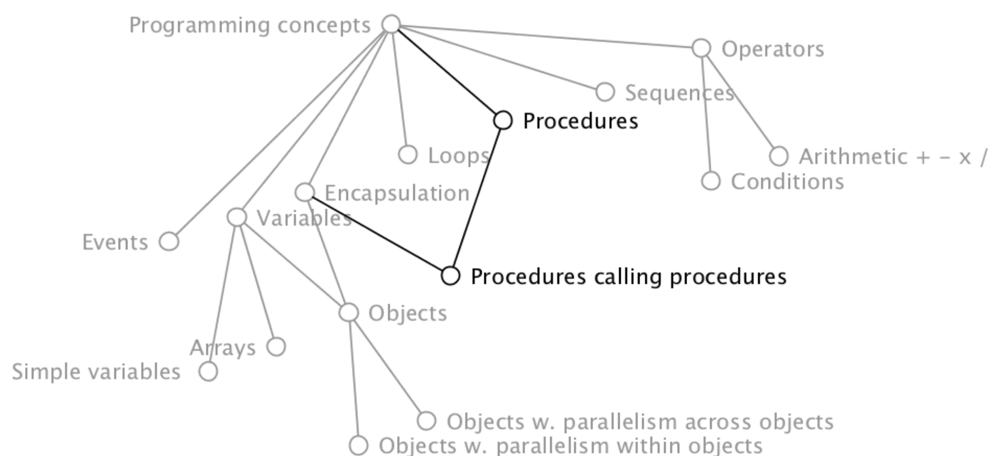

Figure 5. *Procedures calling procedures* as a specialisation of both *procedures* and *encapsulation*

4.1.2 Grouping other concepts as specialisations under the general procedures concept

The updated map (Figure 5 above) represents procedures on the same specialisation level as sequences and loops (directly under programming concepts). This was not accurate as we considered sequences a special simple type of procedures where execution sequence is explicitly specified in a sequence of instructions. This is accommodated in the CSM by showing the *sequences* concept as a specialisation of the *procedures* concept as emphasised in Figure 6. Similarly it was noticed that loops and conditionals are what allow execution sequences to be specified non-sequentially; they allow execution to jump in the instructions. Together they allow an alternative type of procedure to sequences. This is accommodated in the CSM by adding a *procedures*

with flow control concept as a specialisation of *procedures*. The concepts *loops* and *conditionals* could then be repurposed as the concepts *procedures with conditionals* and *procedures with loops* that both can be drawn as specialisations of *procedures with flow control* also emphasised in Figure 6.

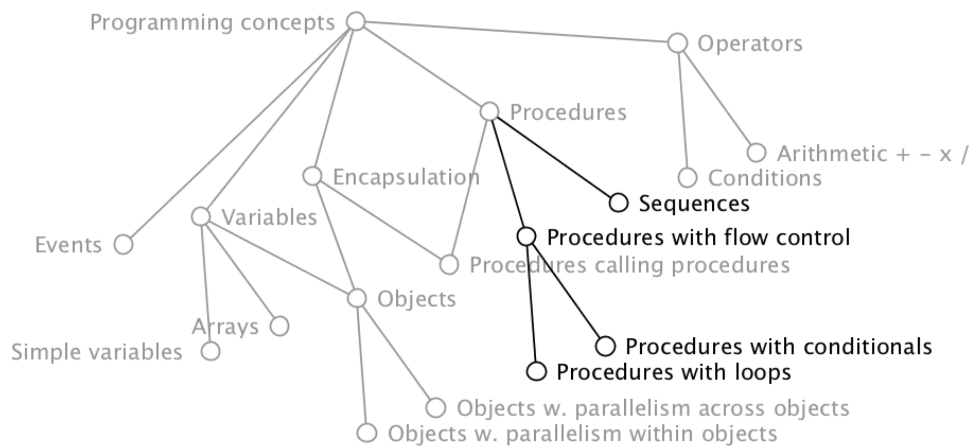

Figure 6. Other specialisations of *procedures*

4.1.3 A special type of looping

Looping is not obviously supported in Lightbot Web. There is no icon labelled "loop" or resembling a looping structure. However at level 3 we see that a procedure can end by calling itself and thereby recursively repeat itself. This provides an equivalent result to some more explicit loop structures. Should we simply mark Lightbot Web as exercising loops? In that case we would draw an arc from *procedures calling procedures* to *procedures with loops* to indicate the latter is a special type of *procedures calling procedures*. An argument against this is that it would brush over important distinctions: A playful programmer that perfects the use of self-calling processes will not necessarily recognize a more explicit loop structure and visa versa. Also a procedure calling itself is more like a looping procedure than a loop in a procedure; they do not share the same parent. Finally, we anticipate that there will be types of programs that will specifically require loops within procedures. We decided to make the distinctions between two ways of looping visible in the CSM by adding a *procedures looping by calling themselves* concept as a specialisation of the *procedures calling procedures* concept as emphasized in Figure 7.

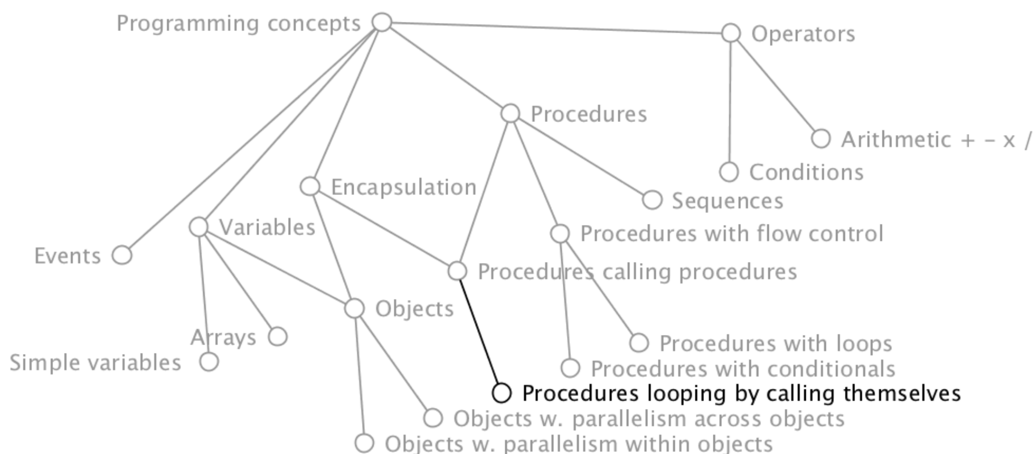

Figure 7. *Procedures looping by calling themselves* as specialisation of *procedures calling themselves*

4.2 Lightbot 9+

After completing all the mazes available on Lightbot Web we had not seen anything suggesting the support of if-then statements as promised by Biggs (2013). We decided to examine the interface of Lightbot's most

advanced version for ages 9+ available on iPhone/iPad, Android, Windows Phone, Mac and Kindle to identify any advanced features not seen in the Web version. We noticed one of relevance to our question.

4.2.1 Different types of conditionals and conditions

Lightbot 9+ displays some grid squares of the obstacle course in a bright colour. Examining these, we saw that a new feature allowed the user to *paint* individual instruction blocks in the procedure to only work when the robot is standing on a square of the same colour. If the robot is standing on a square of a different colour the instruction block is simply skipped. This is undeniably an implementation of procedures with conditionals, but it seems like a crude or simplistic implementation. Because we do not support degrees of implementation in the CSM, we needed to find conceptual differences between these two implementations. We did this in two parts (see Figure 8 below):

1. We considered that the Lightbot feature did not allow the specification of alternative instructions if the condition is not met (the execution simply moves to the next instruction without the colour). In contrast to this Scratch supports if-else structures where alternative instructions are given. This distinction is supported in the CSM by including two specialisations under the *procedures with conditionals* concept: *procedures with if-then conditionals* and *procedures with if-else conditionals.*
2. We recognised that the feature was also limited in the sophistication of the conditions that could be specified. This is not part of a flow control concept but to do with types of conditions operators the environment supports. Conditions operators are the expressions that can be evaluated as either true or false. While Scratch allows three different condition operators ("=", ">" and "<"), Lightbot only supports checking if the values of two properties are equal ("="). In this case Lightbot checks if the colour property of the robots position is equal to the colour property of the instruction. This distinction is supported in the CSM by including two specialisations under the *conditions* concept: *the greater than, less than (> <)* concept and *the equals (=)* concept.

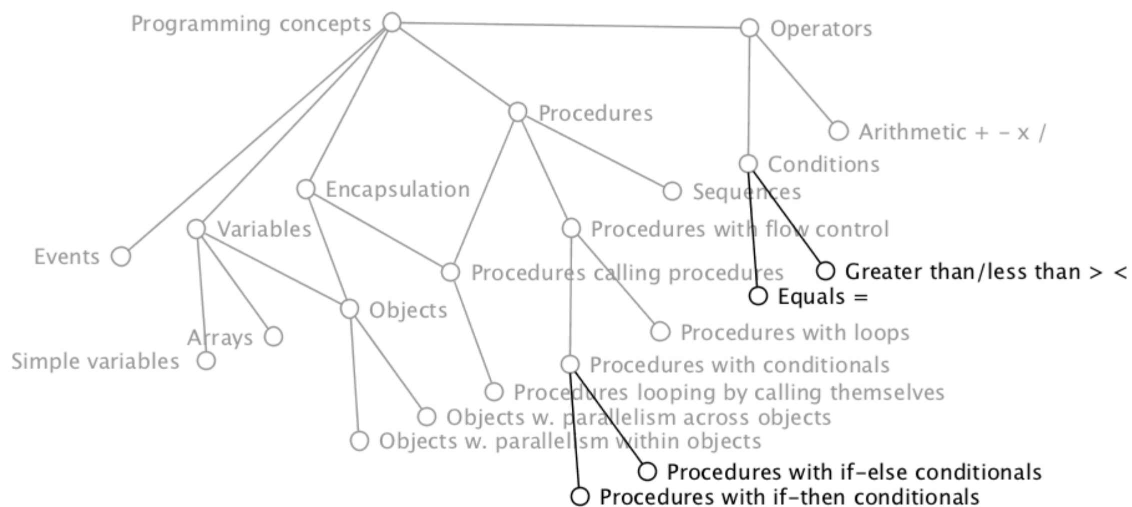

Figure 8. Concepts specialising *procedures with conditionals* and *conditions*

5. Discussion

In the above we have developed a map distinguishing various playful programming concepts. To assess the effectiveness of our CSM based approach to identifying programing concepts, and how it supports the larger *playful programing* project, we will ask and address three questions:

Has using CSM maps helped understand the two products? This can be considered in several ways. They have helped us in identifying relevant programming concepts. We believe that every distinction shown in the maps could in principle be made with words alone, and we have tried making our textual descriptions self-contained, however it seems that CSMs have helped this process. They have helped us to form a clearer overview and identify finer distinctions between programming concepts. However, they have only partially helped us identify the breadth of what the two products do. We have not used them in exploring the *who*, *why*, and *how*

of the products. We have neither used them to identify all of *what* they exercise, for example considering Brennan & Resnick's practices and perspectives, or different levels of learning. They have helped identify multiple concepts that could otherwise be thought of as one and we have certainly added to Brennan and Resnick's computational concepts, but the maps do not show that we have covered the products fully. For this we must rely on the many hours we have spent exploring the products.

Will our model allow us to compare many products? With distinctions between different types of looping, conditionals and conditions we will be able to show how products that only partially support these concepts differ. A convenient way to compare many different products is in a matrix listing programming concepts on the horizontal and product names on the vertical edges. The beginning of such a matrix is shown in Figure 9. Any number of products can be added at the right and the list of concepts on the left is a direct transformation of the current CSM. The list is indented to preserve some insight into how concepts specialise each other, but it could be shown without these indents. Notice that a product is considered to exercise a general concept if it supports one of its specialisations. When exploring more products we will need to make adaptions to the list (via the CSM), however we hope that the list is relatively stable.

	Scratch	LightBot Web	LightBot 9+
Programming concepts	✓	✓	✓
Operators	✓		✓
(+ - x :) Arithmetic	✓		
Conditions	✓		✓
Greater than/less than > <	✓		
Equals =	✓		✓
Procedures	✓	✓	✓
Sequences	✓	✓	✓
Procedures w. flow control	✓		✓
Procedures with conditionals	✓		✓
Procedures with If conditionals	✓		✓
Procedures with If-else conditionals	✓		
Procedures with loops	✓		
Procedures calling procedures		✓	✓
Procedures looping by calling themselves		✓	✓
Encapsulation	✓	✓	✓
see *Procedures calling procedures* under *Procedures*			
Objects	✓		
Objects with parallelism across objects	✓		
Objects with parallelism within objects	✓		
Variables	✓		
see *Objects* under *Encapsulation*			
Simple variables	✓		
Arrays	✓		
Events	✓		

Figure 9. Three products/versions compared against a list showing the current concepts in the CSM.

How stable is the list? We have only examined two products thoroughly, but there is some reason to believe the list will remain relatively stable. We have superficially examined several other products. This has given us many small observations; For example, the product Hopscotch is similar to Scratch, but differs in that it supports *procedures calling procedures*. However a broader pattern also emerges. Playful programming products seem to be dominated by three categories. Two of these have already been identified: navigation games (LightBot, Bee-Bot, LEGO Bits and Bricks and Robo Rally) and visuospatial authoring tools (Scratch, Hopscotch and LEGO Mindstorms). A third category consists of environments supporting coding in text based programming languages (Swifty and Codecademy). As the third category supports traditional full-featured programming languages it becomes less interesting in the context of comparing playful programming concepts. On one level the relative stability of our concept list is suggested by the fact that we have explored

one product from each of the two relevant categories. This is suggested further on a deeper level when we begin to see that products within our categories seem to share the same programming concepts. General concepts like *conditionals* and *loops* can be ascribed to both categories, but specialised concepts like *greater than/less than (> <)*, *procedures with if-else conditionals* and *procedures with loops* are exclusively seen in visuospatial authoring tools. It seems that the programming concepts exercised in an environment are somewhat restrained by their category.

Finally it is worth restating the purpose of this paper in terms of the larger goal of building a vocabulary for describing, developing, teaching and comparing products with the ultimate goal to support the adoption and design of playful programming products in the future.

6. Conclusion

This article has taken inspiration from a model of computational thinking already used to evaluate the Scratch programming environment. It introduces a diagramming technique to support greater overview of the relationships between concepts (than normally achieved in prose text). Using this technique, we create an expanded map of programming concepts while re-examining Scratch. This initial map is applied to an assessment of a different playful programming product: Lightbot. Through a number of iterations the map is updated when differences in the products are identified. Eventually the map accommodates both products allowing subtle distinctions in how general concepts like loops and conditionals are supported. Once clarity is achieved, the map is easily flattened into a list (a candidate list of programming concepts) that can be used to evaluate other playful programming products. Further adaptations to the list are expected when the list is used to evaluate a larger number of products, but it is considered relatively stable.

References

Biggs, John. "Light-Bot Teaches Computer Science With A Cute Little Robot And Some Symbol-Based Programming". *TechCrunch*. N.p., 2016. Web. 4 May 2016.

Brennan, K. and Resnick, M., 2012, April. New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada*.

Gold, J., 2014. *Screen-Smart parenting: How to find balance and benefit in your child's use of social media, apps, and digital devices*. Guilford Publications.

Guzdial, M. and Soloway, E., 2002. Teaching the Nintendo generation to program. *Communications of the ACM*, *45*(4), pp.17-21.

Lakatos, I., 1976. *Proofs and Refutations*. Cambridge University Press.

Misfeldt, M. and Ejsing-Duun, S., 2015. Learning mathematics through programming: An instrumental approach to potentials and pitfalls. In *9th Congress of European Research in Mathematics Education*.

Nagarjuna, G., 2009 "Collaborative Creation of Teaching Learning Sequences and an Atlas of Knowledge", *Mathematics Teaching-Research Journal Online, vol. 3, no. 3*, p. 23.

Repenning, A., Webb, D. and Ioannidou, A., 2010, March. Scalable game design and the development of a checklist for getting computational thinking into public schools. In *Proceedings of the 41st ACM technical symposium on Computer science education* (pp. 265-269). ACM.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B. and Kafai, Y., 2009. Scratch: programming for all. *Communications of the ACM*, *52*(11), pp.60-67.

Robins, R., Rountree, J. and Rountree N., 2003. "Learning and Teaching Programming: A Review and Discussion". *Computer Science Education, vol. 13, no 2*, Routledge.

Rumbaugh, J., Jacobson, I. and Booch, G., 2004. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education.

Rushkoff, D., 2010. *Program or be programmed: Ten commands for a digital age*. Or Books.