Teaching the Art of Computer Programming at a Distance by Generating Dialogues using Deep Neural Networks

Teaching the Art of Computer Programming at a Distance by Generating Dialogues using Deep Neural Networks

Yijun Yu[1], Xiaozhu Wang[2], Anton Dil[1], Irum Rauf[1]
[1]The Open University, UK
[2]The Open University of China

Abstract While teaching the art of Computer Programming, students with visual impairments (VI) are disadvantaged, because speech is their preferred modality. Existing accessibility assistants can only read out predefined texts sequentially, word-for-word, sentence-for-sentence, whilst the presentations of programming concepts could be conveyed in a more structured way. Earlier we have shown that deep neural networks such as Tree-Based Convolutional Neural Networks (TBCNN) and Gated Graph Neural Networks (GGNN) can be used to classify algorithms across different programming languages with over 90% accuracy. Furthermore, TBCNN or GGNN have been shown useful for generating natural and conversational dialogues from natural language texts. In this paper, we propose a novel pedagogy called “Programming Assistant”, by creating a personal tutor that can respond to voice commands, which trigger an explanation of programming concepts, hands-free. We generate dialogues using DNNs, which substitute code with the names of algorithms characterising the programs, and we read aloud descriptions of the code. Furthermore, the application of the dialogue generation can be embodied into an Alexa Skill, which turns them into fully natural voices, forming the basis of a smart assistant to handle a large number of formative questions in teaching the Art of Computer Programming at a distance.
Key Words: Transformative Online Pedagogies, Deep Neural Networks, Algorithm Classification, Chat Bots, Alexa Skill, Programming Assistant

1. Introduction

Teaching programming to novices is a recognised problem in computer science education, and authors such as Windslow (1996), Robins et al. (2003), and Haiduc et al. (2010) have shown that automated summarisation of code is a promising direction. What’s common in these pedagogical approaches (Schulte, Clear, Taherkhani, Busjahn, & Paterson, 2010) is the assumption that automated teaching tools are an auxiliary means to the face to face teaching at traditional offline Universities.

Studies about how people read programs reveal a number of layers to understanding code: for example what each statement means, how control passes from one part of code to another, or what algorithm has been employed (Douce, 2008). The ‘obvious answer’ of how to read code – reading from the top of the page downwards – may not be the best one. We may perhaps attempt to build up a picture of what code does by reading documentation to form a first impression and then work our way down to see how the end effect is achieved. Of course, the documentation may be wrong, or our interpretation of the lower-level code may be faulty. We may also understand code in terms of higher level structures such as methods or classes and how they relate to each other to solve a problem – a more integrated approach, as it involves a mixture of intermediate and higher and lower level code analysis. One aspect of this integration knowledge is the ability to recognise design patterns or common sub-problems and their solutions. Unlike English texts, which can be read from start to finish through speech synthesis (Zen, Senior, & Schuster, 2013), the understanding of programming concepts requires frequent navigations back and forth, up and down, in two dimensions. However, traditional accessibility helpers, such as Emacspeak (Raman, 1996), read out the texts sequentially; whilst the presentations of programs are hierarchical in nature:

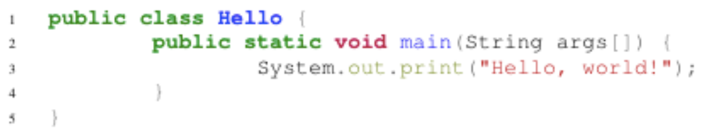
Neverthless, we need to understand the code, which is the only reliable documentation of what it does (Kernighan and Plauger, 1978). Others have argued that the external context of code – e.g. its inputs – are also required to understand a program (Brooks, 1987).

Furnas (1999) points out, in an earlier age of small digital displays, the issues of understanding large structures when viewed through a small window. He proposed a ‘fisheye’ strategy to balance local detail and global context. We suggest that it would be possible to develop a similar approach to program comprehension using audio descriptions, beginning with a high-level description of what code does, and then proceeding to lower-level structures, and lines of code, as needed. Often it may be possible or desirable to skip over some levels of detail. Indeed, the high-level view may be all that is needed in some contexts. Other software geared towards helping visually impaired users to understand programs has also used this approach, e.g. JavaSpeak (Smith et al., 2000) supports navigating trees representing a program's structure.

For online education offered by the Open University, the fundamental ideas behind programming languages are taught through distance learning modules such as *M250* (The Open University, UK), with the aim that students gain first-hand support from the very start, and learn more advanced concepts continuously throughout the course of study. An example of this is the unique learning experience of Software Engineering through the distance education programme (Quinn et al., 2006), where, in addition to students learning technical content, regular interactions with tutors are required, e.g. to elicit stakeholder requirements and refine design.

Given the need for scalability in modules with large cohorts, it is reasonable to aim at fully automating some recurring tasks to alleviate the burden on the tutors. One of the major obstacles to achieving this goal is to support those students with visual disabilities, who require sound as an assisting modality to drive adaptive user interface design (Akiki, Bandara, & Yu, 2017, 2016). However, audio delivery has wider application: it is also relevant in Adaptive User Interfaces (AUIs) (Akiki, Bandara, & Yu, 2014), i.e. software systems that can adapt their modality of use (from desktop to laptop or mobile phones, e.g., from visual to audio) as appropriate to the context. This flexibility of presentation mode, and audio presentation of information in general, can benefit all users of such systems, whether visually impaired or not (Hadwen-Bennett, A. et al. 2018).

To illustrate the task at hand, consider the canonical 'Hello World' program students often begin their programming with. Figure 1 provides an example in Java, which consists of only 5 lines of code.



```java
public class Hello {
        public static void main(String args[]) {
                System.out.print("Hello, world!");
        }
}
```

Fig. 1. A Java program to illustrate programming concepts

Through the use of spaces and indentations, the structure of the program will be clear to most visually capable students. At the highest level, it is the specification of a class, which has 'public' visibility to other classes, named 'Hello'. The pair of curly braces '{' and '}' encloses the members (such as methods) of the class, nested in further structures. The method begins with a header, which includes several modifiers: 'public', 'static', 'void' in this case, the name of the method, 'main', and a list of typed parameters. 'String args[]' here indicates that 'args' is an array variable where each element of the array is of a 'String' type. Beneath the method signature, another pair of curly braces encloses the body of the implementation of the method. In this case, the method body consists of a call to a member of the 'System' class. The recipient of the method call in the 'System' class is a variable 'out' of the 'PrintStream' type , and the 'print' method has an argument 'Hello, world!'. When the program is compiled and executed, the string 'Hello, world!' will appear on the console display.

The above description has a narrative that helps a reader to navigate the syntactical elements from top to down. However, since the program has many details, it is rather tedious to talk through everything, just to find out that what the program is actually doing by listening. A summary may be more useful, or the user may wish to drive an interactive description.

With the advent of voice-interaction technology and products such as Alexa Skill Kit (ASK, *https://developer.amazon.com/alexa-skills-kit*), we seek the opportunity to translate sequential narratives into hierarchical ones, driven by the requirements (Lapouchnian, Yu, Liaskos, & Mylopoulos, 2016) of students.

This new proposal aims to focus on any part of their programs, whilst maintaining an overview relevant to the studied concepts.

To implement this proposal, we introduce a deep neural networks (DNN)-based pedagogy called *Programming Assistant* (PA) that can respond to voice commands that trigger an explanation of programming concepts.

Analogous to pointing a mouse to program elements in an integrated development environment (IDE) such as Eclipse (*http://eclipse.org*) or BlueJ (*https://www.bluej.org*), the new hands-free mode of interactions could generate intelligent dialogues that answer students' questions about the

program or a programming concept, meaningfully (Yu, Tun, & Nuseibeh, 2011).
For example, an interaction might be as follows:

Student: *Alexa, open Program Artist on a Hello World program*
Alexa: *Okay, Program Artist is open. What class would you like to examine?*
Student: *Examine the 'Hello' class.*
Alexa: *Okay, I have opened the 'Hello' class.*
Student: *Does the class have method calls?*
Alexa: *In the Hello class, there are method calls to main and to print.*
Student: *What is going to be printed?*
Alexa: *"Hello comma world exclamation mark" will be printed.*

However, instead of asking the previous question, one may ask instead '*Tell me more about the method call to print'* and the answer might be '*The print method is called through a static variable "out" of the "System" class.'*

A further question can be asked about the 'out' variable too, and so on. This scenario indicates the advantage of using Programming Assistant, which does not have to provide every detail of the program, while partial answers will be provided and will be expanded further by answers to follow up questions. In other words, a dialogue rather than a monologue results from the new way of communicating with students.

In the remainder of the paper, Section 2 presents an overview of the approach and deep neural networks, Section 3 compares with related work, Section 4 discusses our initial evaluation and concludes.

2. Our Approach

Figure 2 illustrates an overview our Programming Assistant architecture. First, a program will be parsed into abstract syntax trees (AST), which represent the nested structure of code. The system will translate an initial question with respect to the initial parameter (typically configured as the root node of the AST). Combining the question and the parameters, PA will report a result back to the student. The student can ask follow-on questions using the returned parameters as the new context.

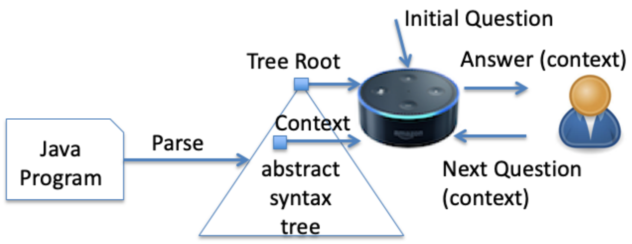


Figure 2: An overview of PA architecture

The parsing to an AST can be done on the server side of the Alexa Skill, while the interactions with the student would alter the parameters depending on the additional questions students asked.

In this paper, we have shown an example dialogue based on the simple program in the last section. Figure 3 lists the AST in terms of XML tree, which is generated from our FAST parser (Yu, 2019) efficiently on the server side.

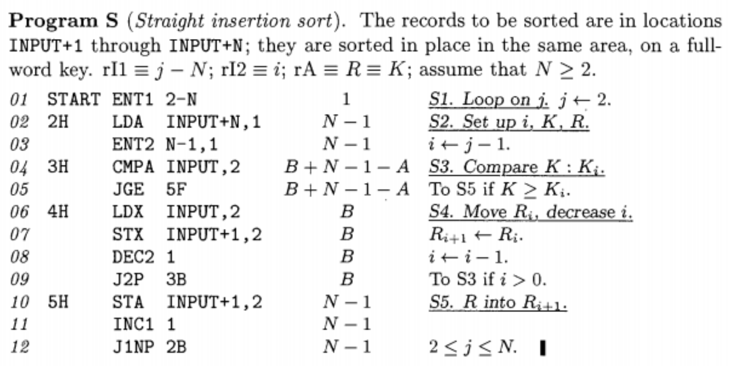
```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?> <unit><class><specifier>public</specifier>
class <name>Hello</name> <block> <function><specifier>public</specifier> <specifier>static</specifier>
<type><name>void </name></type> <name>main</name><parameter_list>
(<parameter><decl><type><name>String</name></type> <name><name>args</name><index>[]</index>
</name></decl></parameter>)</parameter_list> <block> <expr_stmt><expr><call><name><name>
System</name><operator>.</operator><name>out </name><operator>.</operator><name>print
</name></name><argument_list>(<argument> <expr><literal type="string">"Hello, world!"
</literal></expr></argument>)</argument_list> </call></expr>;</expr_stmt> </block></function>
</block></class> </unit>
```

Figure 3: An XML corresponding to the AST of the example program

The rest of the infrastructure follows an AI agent approach, where the front end to the agent is a human-friendly voice interface using a cloud-based Alexa Skill Kit environment, and the back end to the agent is a cross-language machine learning model trained on a large corpora of curated source code.

2.1 Deep Neural Networks (DNN) for Algorithm Teaching Tasks

Knuth’s The Art of Computer Programming is an example of traditional teaching materials for students to learn programming (Knuth, 1968). In his preface, the intention of using assembler language to explain algorithms was to provide an intermediate representation closest to the machine instructions. For example, Figure 4 presents one example of an Insertion Sort, which consists of five statements at a high-level on the right and the corresponding machine instructions to implement them on the left.



Program S (*Straight insertion sort*). The records to be sorted are in locations INPUT+1 through INPUT+N; they are sorted in place in the same area, on a full-word key. rI1 $\equiv j - N$; rI2 $\equiv i$; rA $\equiv R \equiv K$; assume that $N \geq 2$.

```
START ENT1 2-N                1          S1. Loop on j. j ← 2.
2H    LDA  INPUT+N,1      N − 1          S2. Set up i, K, R.
      ENT2 N-1,1          N − 1          i ← j − 1.
3H    CMPA INPUT,2    B + N − 1 − A      S3. Compare K : Ki.
      JGE  5F         B + N − 1 − A      To S5 if K ≥ Ki.
4H    LDX  INPUT,2            B          S4. Move Ri, decrease i.
      STX  INPUT+1,2          B          Ri+1 ← Ri.
      DEC2 1                  B          i ← i − 1.
      J2P  3B                 B          To S3 if i > 0.
5H    STA  INPUT+1,2      N − 1          S5. R into Ri+1.
      INC1 1              N − 1
      J1NP 2B             N − 1          2 ≤ j ≤ N. ▮
```

Figure 4. “Insertion Sort” in the assembler of Art of Computer Programming (p.80)

However, the essence of an algorithm may be presented in different high-level programming languages, such as Java or C#. A question is, could there be a programming language-agnostic way to represent an algorithm?

Recently deep neural networks (DNN) have been proposed to represent source code, such as Tree-Based Convolution Neural Networks (Mou, Li, Zhang, Wang, & Jin, 2016) and Gated Graph Neural Networks (Li, Tarlow, Brockschmidt, & Zemel, 2016). Comparing to traditional representation of code as a bag of words, or a sequence of tokens (n-grams), deep neural networks that take into account structures such as nested syntax trees and/or semantic graphs in the code have been shown to be more effective (Nghi, Yu, & Jiang, 2019). For the algorithm benchmarks in Java and C++, it is shown that over 90% accuracy can be achieved in classifying the algorithms regardless of which programming language was chosen. For example, the DNN model trained to learn from algorithms implemented in Java could still be used when the underlying programming language becomes C++.

As a result, we can query the underlying algorithm classifier using any input program to get 90% accurate answer without any human tutor intervention. In fact, recent progress in ASTNN (Zhang et al., 2019) has shown a 98% accuracy when the DNN are highly tuned for the benchmarks. Furthermore, DNN’s such as TBCNN and GGNN have applications in summarisation of code snippets into natural language utterances (Mou, Meng, et al., 2016, Fernandes, Allamanis, & Brockschmidt, 2019).

As a result, the DNNs can be used for teaching different tasks in programming as long as there is a high accuracy. Of course, even with 90% accuracy, it demands human interaction to explain the the remaining 10%.

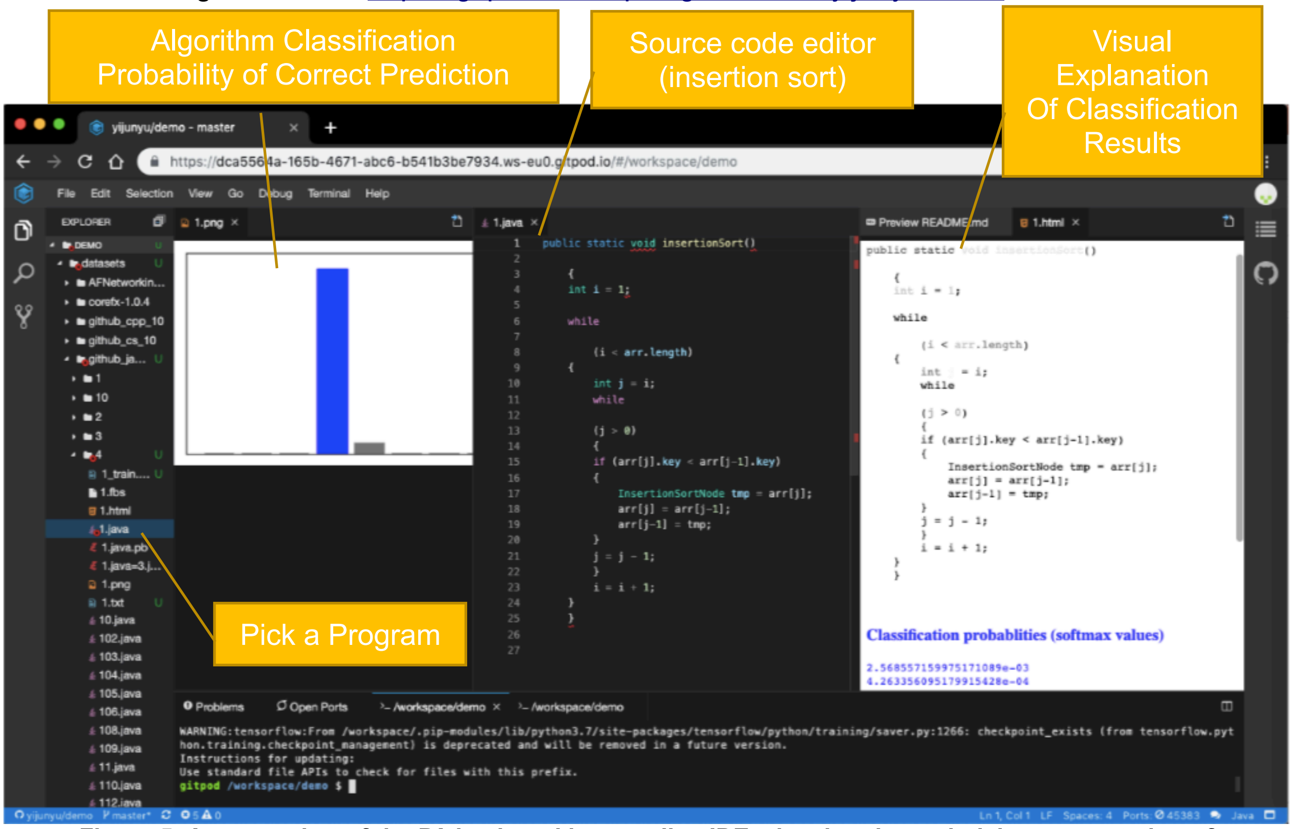
2.2 Alexa Skill

Initially we have concentrated on high-level program classification. We used the open-source project flask (*https://github.com/johnwheeler/flask-ask*) to simplify the development and deployment of server side implementation using its python interface.

On the client side, the ASK must be configured in such a way that many types of questions can be asked (using Alexa’s powerful synthesis model), while we have to define the context variables by recognising the parameters used in the answers to previous questions. Since open-ended questions can be asked, we have some predefined parameters to prompt students when they are stuck, e.g. the initial landing node of the AST in the navigation is chosen as the root node of the AST.

2.3 Online IDE

To be able to demonstrate the PA, we have also implemented a pedagogical online IDE, which does not have a voice interface through Alexa. A screenshot of the IDE is shown in Figure 5, which is the result of clicking at the URL https://gitpod.io/#https://github.com/yijunyu/demo in a Web browser.



```java
public static void insertionSort()

    {
    int i = 1;

    while

        (i < arr.length)
    {
        int j = i;
        while

        (j > 0)
        {
        if (arr[j].key < arr[j-1].key)
        {
            InsertionSortNode tmp = arr[j];
            arr[j] = arr[j-1];
            arr[j-1] = tmp;
        }
        j = j - 1;
        }
        i = i + 1;
    }
    }
```

```
public static void insertionSort()

    {
    int i = 1;

    while

        (i < arr.length)
    {
        int j = i;
        while

        (j > 0)
        {
        if (arr[j].key < arr[j-1].key)
        {
            InsertionSortNode tmp = arr[j];
            arr[j] = arr[j-1];
            arr[j-1] = tmp;
        }
        j = j - 1;
        }
        i = i + 1;
    }
    }
```

Classification probablities (softmax values)

```
2.5685571599751710890e-03
4.2633560951799154280e-04
```

```
WARNING:tensorflow:From /workspace/.pip-modules/lib/python3.7/site-packages/tensorflow/python/training/saver.py:1266: checkpoint_exists (from tensorflow.pyt
hon.training.checkpoint_management) is deprecated and will be removed in a future version.
Instructions for updating:
Use standard file APIs to check for files with this prefix.
gitpod /workspace/demo $
```

Figure 5: A screenshot of the PA back-end in an online IDE, showing the underlying computation of DNN: the code snippet in the centre shows an InsertionSort algorithm implemented in Java; the bar chart on the left shows which one of the algorithm class has the highest probability according to the tool, which is correct in this case; and the grey-scale decorated preview of the code explains the underlying reasoning of the classification.

In this IDE, it is possible to see how the DNN works from end-to-end, and in particular how an input algorithm can be classified into one of several predefined algorithms. The students can also preview the code decorated with colours in grey-scale corresponding to the importance of the tokens assigned by the underlying PA.

3 Related Applications

As we have suggested already, programmers not only have to write code; they have to read it also. We may read code for a variety of reasons: to determine the implementation language, to decide whether it is of good quality, to decide whether it is correct, or to decide what it does from a functional point of view, being just a few examples.

Some studies in program comprehension have considered how novices and experts read code, and it is recognised that the ability to quickly summarise what code does is a mark of a superior programmer (Robins et al., 2003). Studies reviewed by Winslow (1996), for example, have concluded that novices approach programming "line by line" rather than using meaningful program "chunks" or structures. Studies collected in Soloway and Spohrer (1988) outline deficits in novices' understanding of various specific programming language constructs (such as variables, loops, arrays and recursion, etc.) It is said that novices are "very local and concrete in their comprehension of programs" (Robins et al., 2003).

Deimel and Naveda (1990) ask how people read computer programs and how to teach students to read code. They point out that the ability to read code is an often overlooked skill. It provides an opportunity for programmers to share and learn from each other's work, including from code deposited in repositories. This also allows a programmer to learn good style by example. In the current age of computing, we would argue that discovering and reusing code in repositories is likewise an important skill. In this context also, a quick summarization of what code does could be helpful.

Difficulties in reading code share some difficulties with reading in general. For example, we may not know how to pronounce certain symbols, and this reduces our ability to understand them and internalize their meaning. Hearing such symbols spoken aloud mitigates against this issue.

A related issue is whether code itself can be considered to be readable, or is inherently unclear, whether deliberately or through inferior or perhaps excessively optimised coding strategies. This in turn relates to standards for coding style that relate to readability. Algorithms that attempt to verbalise code may provide new insights in this area.

In the context of education, particularly where large groups of students are involved, scalability is important, and a quick summarisation of what a program does is potentially a very useful tool to humans in assessing their students' work. Such software does not relieve us of the burden of carefully checking whether code is correct, but it may help to identify where it is not, or to help markers target their efforts more quickly to where advice is needed. This is similar to the approach adopted in 'code reviews', so also of interest in this context.

4 Conclusions

We have proposed a novel pedagogy to teach the art of programming, i.e., algorithms, at a distance. Using utterances for conveying programming concepts in different programming languages, and implemented as an Alexa skill, the proposed programming assistant becomes a tool to answer queries about what source code means.

In the future, we plan to expand PA to an intelligent chat bot. . *Chatbots are computer programs used to conduct auditory or textual conversations* (Winkler & Söllner, 2018). An intelligent chatbot can facilitate the teaching of a wide range of open-ended programming tasks instead of certain sets of prepared algorithms. In doing that, we can leverage the potential of a chatbot in teaching a variety of students who come from different backgrounds and and have different level of expertise. This could be particularly interesting in the context of teaching secure coding practices. Student can be taught to avoid vulnerabilities in code taking into account the conversational flow between a chatbot and a student, A chatbot can intelligently give answers to students in the context of their security understanding and expertise. Interestingly, studies have also shown that the effectiveness of chat bots increases many-fold if the associated social and cognitive contexts are addressed in their design (Brown & Parnin, 2019). These social and cognitive contexts for our PA include how chat bots fit into the workflow of students' learning environments and programming practices. For example, we might consider various layers of correctness in source code (Dil, 2019) such as compilation errors, style guidelines and the passing of unit tests, and the extent to which the code conforms to them, in formulating feedback.

Acknowledgement
This work is in part supported by the eSTEeM programme and the EPSRC project on Secure software development by the masses (EP/P011799/2).

References

Akiki, P. A., Bandara, A. K., & Yu, Y. (2014). Adaptive model-driven user interface development systems. *ACM Comput. Surv.*, *47*(1), 9:1–9:33. doi: 10.1145/2597999

Akiki, P. A., Bandara, A. K., & Yu, Y. (2016). Engineering adaptive model-driven user interfaces. *IEEE Trans.* on *Software Eng.*, *42*(12), 1118–1147. doi: 10.1109/TSE.2016.2553035.

Akiki, P. A., Bandara, A. K., & Yu, Y. (2017). Visual simple transformations: Empowering end-users to wire internet of things objects. *ACM Trans. Comput.- Hum. Interact.*, *24*(2), 10:1–10:43. doi: 10.1145/3057857.

Brooks, F. P. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, *20*, 10–19.

Brown, C., & Parnin, C. (2019). Sorry to bother you: designing bots for effective recommendations. In *Proceedings of the 1st international workshop on bots in software engineering* (pp. 54–58).

Deimel, L., & Neveda, J. (1990). *Reading computer programs: Instructors guide and exercises* (Tech. Rep.). Carnegie Mellon University.

Dil, A. (2019). *Layered online feedback on code quality.* Horizons in STEM, July 2019 https://ukstemconference.wordpress.com/

Douce, C., (2008) The Stores model of Code Cognition. In PPIG, Lancaster. http://www.ppig.org/library/paper/stores-model-code-cognition

Fernandes, P., Allamanis, M., & Brockschmidt, M. (2019). Structured neural summarization. In *International conference on learning representations.* Retrieved from https://openreview.net/forum?id=H1ersoRqtm

Furnas, G. W. (1999). Readings in information visualization. In S. K. Card, J. D. Mackinlay, & B. Shneiderman (Eds.), (pp. 312–330). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. Retrieved from http://dl.acm.org/citation.cfm?id=300679.300769

Haiduc, S., Aponte, J., & Marcus, A. (2010, May). Supporting program comprehension with source code summariza- tion. In *2010 acm/ieee 32nd international conference on software engineering* (Vol. 2, p. 223-226). doi: 10.1145/1810295.1810335

Hadwen-Bennett, A. et al. 2018. Making Programming Accessible to Learners with Visual Impairments : A Literature Review. 2, 2 (2018). DOI:https://doi.org/10.21585/ijcses.v2i2.25.

Kernighan, B. W., & Plauger, P. J. (1978). *The elements of programming style (2. ed.)*. McGraw-Hill.

Knuth, D. E. (1968). *The art of computer programming, volume I: fundamental algorithms*. Addison-Wesley.

Lapouchnian, A., Yu, Y., Liaskos, S., & Mylopoulos, J. (2016). Requirements-driven design of autonomic application software. In *Proceedings of the 26th annual international conference on computer science and software engineering, CASCON 2016, toronto, ontario, canada, october 31 - november 2, 2016* (pp. 23–37). http://dl.acm.org/citation.cfm?id=3049879

Li, Y., Tarlow, D., Brockschmidt, M., & Zemel, R. (2016, November). Gated graph sequence neural networks. In *Iclr.* (arXiv: 1511.05493)

Mou, L., Li, G., Zhang, L., Wang, T., & Jin, Z. (2016, February 12-17). Convolutional neural networks over tree structures for programming language processing. In *AAAI* (pp. 1287–1293).

Yu, Y., fAST: Flattening Abstract Syntax Tree for Efficiency. In G. Mussbacher, J. M. Atlee, & T. Bultan (Eds.), In: *Proceedings of the 41st international conference on software engineering: Companion proceedings, ICSE 2019, montreal, qc, canada, may 25-31, 2019.* (pp. 278–279). IEEE/ACM. Retrieved from https://dl.acm.org/citation.cfm?id=3339783

Yu, Y., Tun, T. T., & Nuseibeh, B. (2011). Specifying and detecting meaningful changes in programs. In *26th IEEE/ACM international conference on automated software engineering (ASE 2011), lawrence, ks, usa, november 6-10, 2011* (pp. 273–282). Retrieved from https://doi.org/10.1109/ASE.2011.6100063 doi: 10.1109/ASE.2011.6100063

Nghi, B. D. Q., Yu, Y., & Jiang, L. (2019). Bilateral dependency neural networks for cross-language algorithm classification. In *Saner* (pp. 422–433). doi: 10.1109/SANER.2019.8667995

Nghi, B. D. Q., Yu, Y., & Jiang, L. (2019). SAR: Learning cross-language API mappings with little knowledge. In *ESEC/FSE 2019.* (Accepted full paper)

Zen, H., Senior, A., & Schuster, M. (2013). Statistical parametric speech synthesis using deep neural networks. In *Proceedings of the ieee international conference on acoustics, speech, and signal processing (icassp)* (pp. 7962–7966).

Mou, L., Men, R., Li, G., Xu, Y., Zhang, L., Yan, R., & Jin, Z. (2016, August). Natural language inference by tree-based convolution and heuristic matching. In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 2: Short papers)* (pp.130–136). Berlin, Germany: Association for Computational Linguistics. doi: 10.18653/v1/P16-2022

Quinn, B., Barroca, L., Nuseibeh, B., Fernandez-Ramil, J., Rapanotti, L., Thomas, P., & Wermelinger, M. (2006, 12). Learning software engineering at a distance. *Software, IEEE*, *23*, 36-43. doi: 10.1109/MS.2006.169

Raman, T. V. (1996). Emacspeak: A speech interface. In M. J. Tauber, V. Bellotti, R. Jeffries, J. D. Mackinlay, & J. Nielsen (Eds.), *Proceedings of the conference on human factors in computing systems: Commun ground, new york, 13-18 april 1996* (p. 66-71). ACM Press. Retrieved from http://emacspeak.sourceforge.net

Smith, A.C. et al. 2000. A Java programming tool for students with visual disabilities. Proceedings of the fourth international ACM conference on Assistive technologies - Assets '00. (2000), 142–148. DOI:https://doi.org/10.1145/354324.354356.

Robins, A., Rountree, J., & Rountree, N. (2003, 06). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*, 137-. doi: 10.1076/csed.13.2.137.14200

Winkler, R. & Söllner, M. (2018): Unleashing the Potential of Chatbots in Education: A State-Of-The-Art Analysis. In: Academy of Management Annual Meeting (AOM). Chicago, USA.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., & Liu, X. (2019). A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st international conference on software engineering* (pp. 783–794). Piscataway, NJ, USA: IEEE Press. doi: 10.1109/ICSE.2019.00086