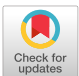
Journal of Computer Languages 69 (2022) 101087

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.elsevier.com/locate/cola

Language impact on productivity for industrial end users: A case study from Programmable Logic Controllers



Felipe Fronchetti [a,*], Nico Ritschel [b], Reid Holmes [b], Linxi Li [a], Mauricio Soto [c], Raoul Jetley [d], Igor Wiese [e], David Shepherd [a]

[a] *Virginia Commonwealth University, USA*
[b] *University of British Columbia, Canada*
[c] *Hitachi ABB Power Grids, USA*
[d] *ABB Corporate Research, India*
[e] *Federal University of Technology – Paraná, Brazil*

ARTICLE INFO

CCS Concepts:
Software and its engineering
Visual languages

Keywords:
Ladder Logic
Programmable Logic Controllers
Industrial end-user programming

ABSTRACT

Industrial workplaces increasingly require end-users to create programs for embedded systems, but little expert scrutiny has been devoted to studying this domain. As a result, industrial end-user programmers may rely on programming languages and development environments that do not necessarily follow the state-of-the-art of software engineering. Consider Ladder Logic, the most popular language used to program the most widely deployed type of industrial hardware, programmable logic controllers (PLCs). Ladder Logic's fundamental design is based on electric relay circuits that have long since disappeared from practice. Does Ladder Logic inhibit the productivity of end-user programmers, slowing progress in industrial settings like manufacturing sites and scientific labs where it is widely used? To better understand the usage of domain-specific languages in industrial practices, we conducted a survey with 175 technical employees from an international engineering conglomerate. This survey introduced participants to Ladder Logic and asked them questions that all programmers, including novices, should answer with ease. Nearly 70% failed, including those with previous Ladder Logic experience. We combined end-user performance with answers in an open-ended question, where many employees complained about the programming language. The breadth and depth of these struggles suggest that outdated languages, which industrial end users must increasingly use, could dramatically impact productivity and that further studies on these industrial end user programmers be necessary to better support them in their increasingly complex workplaces.

1. Introduction

Advances in technology make programming an essential everyday task in a growing number of industries [1]. To keep up with this, organizations depend on more of their employees to be able to program instead of relying only on dedicated software developers. These end-users require specific support by tools that are easy to learn and use, often referred to as low-code environments, as they allow end-users to create software while writing little or no traditional source code. This demand has led to a thriving market for domain-specific programming languages as part of the low-code movement [2].

New low-code solutions appear rapidly for applications in industry [3,4], business [5], computer science education [6], and a wide array of end-users domains [7–9]. However, not all of these areas regularly face scrutiny and methodological evaluation: While previous work has analyzed, categorized, and standardized educational (e.g., [10,11]) and business (e.g., [12–15]) languages to great extent, ***industrial*** end-user programming (e.g., [16–18]) remains comparatively unexplored. Except for pioneering work on analyzing code smells in LabVIEW [19,20], little work has focused on industrial languages. Worse yet, there is little standardization: new, supposedly end-user friendly, programming languages constantly enter the market, and once popular languages rarely disappear entirely [21].

Many industrial languages introduce new visual paradigms with the intent to be user-friendly [17] or relate to domain-specific notation [22]. Some languages also use or combine existing paradigms like dataflow-based [17,23,24], graph-based [25], block-based [26], or analogy-based programming [27]. Some of these paradigms been evaluated empirically, showing great promise [28,29]. However, most languages from the industrial realm that are used today are *Legacy Languages* that were created by practitioners in their domain and have

* Corresponding author.
E-mail addresses: fronchettl@vcu.edu (F. Fronchetti), ritschel@cs.ubc.ca (N. Ritschel), rtholmes@cs.ubc.ca (R. Holmes), lil24@vcu.edu (L. Li), mausotog@hotmail.com (M. Soto), raoul.jetley@in.abb.com (R. Jetley), igor@utfpr.edu.br (I. Wiese), shepherdd@vcu.edu (D. Shepherd).

https://doi.org/10.1016/j.cola.2021.101087
Received 30 June 2021; Received in revised form 24 December 2021; Accepted 24 December 2021
Available online 6 January 2022
2590-1184/© 2021 Elsevier Ltd. All rights reserved.

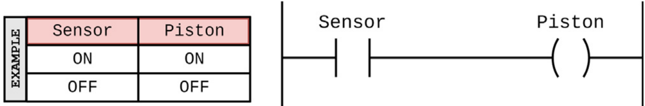


Fig. 1. Example code for a recycling plant in Ladder Logic, explained in detail in Section 2.

faced almost no scrutiny from language designers, especially with respect to the challenges industrial end-users, in contrast to professional software developers, experience using these languages. The few known studies of these languages were conducted decades after their creation [30,31] and have had no discernible impact on the languages or their programming environments. This raises the question, have we accidentally created a two-class system for end-user programmers? As business-oriented end-users enjoy huge leaps in productivity due to the innovations from academic research [32] and the low-code movement [5], have we left industrial end-users, who traditionally program manufacturing lines and other embedded systems, behind to languish in a 1970s programming purgatory?

This paper seeks to identify the challenges industrial end-user programmers face with the languages they must use. Specifically, we focus on industrial end users programming the most widely used programmable hardware in industry, *Programmable Logic Controllers (PLCs)*, as determined by its 11.21 billion USD market in 2020 [33]. PLCs, which are used to control small to medium scale industrial automation tasks, can be programmed using five different languages, as defined in IEC-61131-3 [22]. Of these Languages *Ladder Logic* [34] is by far the most used and is therefore the subject for this investigation. It is so popular that, while no other language from this standard is ranked in the top 100 programming languages in use today, Ladder Logic is ranked as the 50th most used language, ranking above CoffeeScript and Racket, among others [35], despite its domain-specific deployment.

Programs written in Ladder Logic, as shown in Fig. 1, look similar to hard-wired circuit diagrams that were popular in the 1970s. Originally, the popularity of these diagrams helped make Ladder Logic easy-to-learn and easy-to-use for engineers without formal programming training [22,36], but the prevalence of these domain-specific diagrams has faded over time [37]. Because Ladder Logic originated from these diagrams instead of a well-understood programming paradigm, no previous work has thoroughly investigated its learnability or usability, or its ultimate effect on end-user productivity.

In this paper we present an evaluation of Ladder Logic with several beginner-level programming tasks to assess their programming competence (often referred to as a *FizzBuzz* tasks) that we conducted with 175 professional engineers from a multi-national engineering conglomerate. FizzBuzz questions, made popular in technical job screening interviews, are meant to test the most basic level of skill in a programming language [38]. If a job candidate could not answer a set of given FizzBuzz questions with 100% accuracy they would not be considered competent in that language and likely fail the job screening. Our test consisted of 10 questions that we designed with a similar goal in mind. Our study results show that **nearly 70% of users failed our FizzBuzz test for Ladder Logic**, including users with previous Ladder Logic experience. Furthermore, these results are not the result of a single difficult question—10% to 25% of participants answered each individual question incorrectly—demonstrating a consistent failure rate across questions. Mirroring their difficulties in a follow-up satisfaction survey, users rated Ladder Logic's usability to be only "fair".

To better contextualize their performance, we also collected 98 comments from participants about usability challenges and concepts they considered particularly difficult. Multiple participants complained that Ladder Logic programs became too large and found that data dependencies made programs too hard to read. This is remarkable since all of the programs shown to participants were orders of magnitude

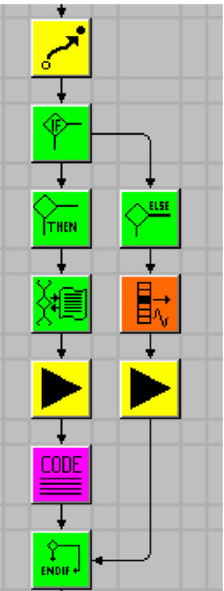


Fig. 2. MORPHA [39].

smaller and less complex than real world programs. While experience, both with programming and with PLC programming, helped users perform slightly better, many experienced users argued that Ladder Logic was really only useful as a learning language. These findings, together with the high failure rate, show that there is a measurable, negative impact on productivity caused by ignoring the hard-won lessons of modern programming languages.

Unfortunately, incremental improvements may not be enough to redeem legacy languages like Ladder Logic. Even with significant improvements to both the language and the supporting tooling fundamental challenges may remain, and thus we believe the most promising way forward would be to replace this language entirely. However, as any replacement language has the potential to become a legacy language itself in the future, we consider it crucial to learn from the mistakes of Ladder Logic.

Does this happen? Consider this, the MORPHA language for the intuitive programming of robots was created in 2002. Its primary innovation was its use of icons to represent robot instructions [39]. Even though it had the backing of two major robot manufacturers it failed to gain traction in industry, primarily because users found it difficult to remember the meaning of each icon [41]. Unfortunately, because this experience remains largely undocumented, industry was doomed to repeat this mistake. In 2018 a new robot programming language was created for the Franka Emika robot [40]. As shown in Fig. 3, other than superficial differences (e.g., horizontal vs vertical layout and improved graphics), this new language's use of iconography is nearly identical to MORPHA's failed approach, which is shown in Fig. 2.

Through our investigation of the challenges faced by industrial end-user programmers, we demonstrate that their effectiveness programming embedded systems is hampered by their current programming languages. By examining the industrial end-users feedback and task performance, we have identified a series of evidence-based language design observations that could improve the software development experience for these users. We hope that this feedback can help provide evidence to inform future language design, enabling improved state-of-the-art programming tools and languages for industrial end-users, ultimately increasing their productivity while performing these complex tasks.

The remainder of the paper provides the details necessary to understand and replicate our work. Section 2 provides the necessary

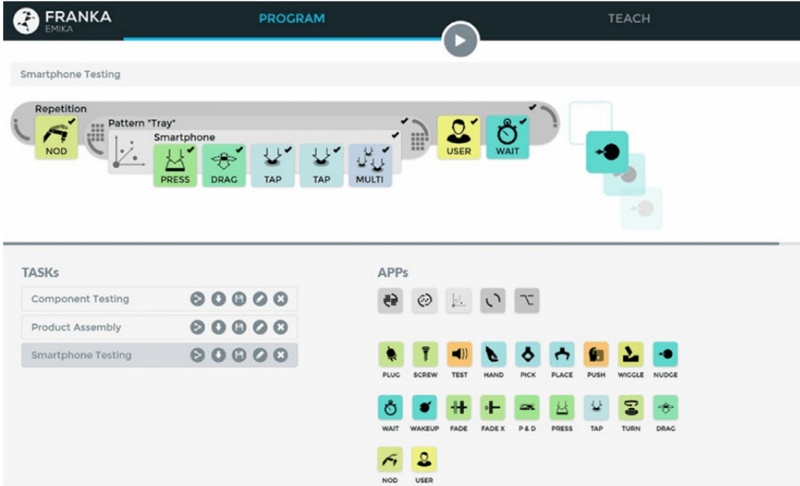


Fig. 3. Franka Emika [40], created 16 years after MORPHA, uses a similar approach to iconography even though MORPHA's iconography was a major contributor to its failure.

background to those unfamiliar with PLCs and Ladder Logic. Section 4 describes the research questions and the survey used to investigate these questions. Section 5 provides the details of our results and Section 6 discusses some of the implications of the results. Section 7 identifies the possible limitations of our research method and Section 3 places our work in the context of existing investigations on this topic.

2. Programming PLCs with ladder logic

To illustrate a typical use case for Programmable Logic Controllers and Ladder Logic, consider the following industrial process: A recycling center regularly receives a mix of metallic and plastic refuse that is dumped on a moving conveyor belt. Before the plastic can be recycled, the metallic materials need to be sorted out. While the recycling center could hire employees to manually identify and remove the metallic materials, this solution is not cost-effective and human workers have a low accuracy at this type of task [42]. Instead, the recycling plant could use an inductive sensor that detects metallic objects and a pneumatic piston that automatically pushes metallic items off the conveyor. This strategy is cheaper and more effective than using human workers, but it requires some initial effort to set up and connect the individual components.

Programmable Logic Controllers are digital computers that connect hardware components like the sensor, conveyor and piston in our example [43]. Although similar to conventional computers in terms of architecture, PLCs were designed to withstand to the adversities of industrial environments, including problems such as dirt, dust, variations of temperature, humidity and noise [44–46]. Compared to just wiring all the components together directly, PLCs reduce the effort to set up the system and, more importantly, make changes in the future. Since the inputs and outputs from all hardware flow through the PLC, the recycling plant could easily add more pistons or sensors and connect them without re-wiring the existing hardware. Instead, all they have to do is re-program the PLC.

PLCs are usually programmed using software provided by the PLC manufacturer. This software is installed on a conventional computer, and programmers only transfer their final, compiled code to the PLC for execution [47]. Although some PLC manufacturers support of traditional programming languages like C or Pascal, most PLCs follow the standards defined by the International Electrotechnical Commission (IEC) [48,49]. The IEC 61131-3 standard defines the five programming languages for PLCs, which include: Ladder Logic (LD), Function Block Diagrams (FBD), Structured Text (ST), Instruction Lists (IL) and Sequential Function Charts (SFL) [47,49,50]. Ladder Logic is the most popular of these five languages [35,36,51].

Ladder Logic is a visual programming language designed to resemble the relay logic schemes used in hard-wired circuits [52–54]. Ladder Logic visualizes programs as diagrams that use symbols representing input and output devices such as buttons, switches or motors. These symbols are connected by vertical and horizontal lines that describe the program logic as a circuit. Every diagram must at least have two vertical lines on opposite sides that are called power rails [52]. Horizontal lines connect the two power rails and create the rungs of the ladder. Horizontal lines can branch or merge and always connect one or more input and output symbols with each other.

Consider the industrial process of the recycling plant we presented earlier. Fig. 1 shows a truth table for scenario (left) that directly connects the inductive sensor and the pneumatic piston. The right side of the figure shows the encoding of this program in Ladder Logic. The `Sensor` is represented as a normally open input contact, which means that the circuit is only closed when the sensor is activated. The `Piston` is represented by an output contact. The overall diagram can be read as follows: "*When the inductive sensor is on, then the pneumatic piston is on.*".

Ladder Logic can be used to represent a wide variety of concepts. For instance, Fig. 4 shows three separate Ladder Logic programs and their truth tables, representing the operators NOT, AND, and OR in boolean logic. These can be used as building blocks for more complex programs. In the first diagram, the input contact labeled `Input` is a normally closed contact, meaning that the circuit is only closed when it is not activated [55]. The other two diagrams use more established circuit notation and can generally be read as follows: "*Output are on when they lie on a closed circuit that connects the two power rails.*". The training video[1] that we used in our survey can provide an additional understanding of the Ladder Logic language.

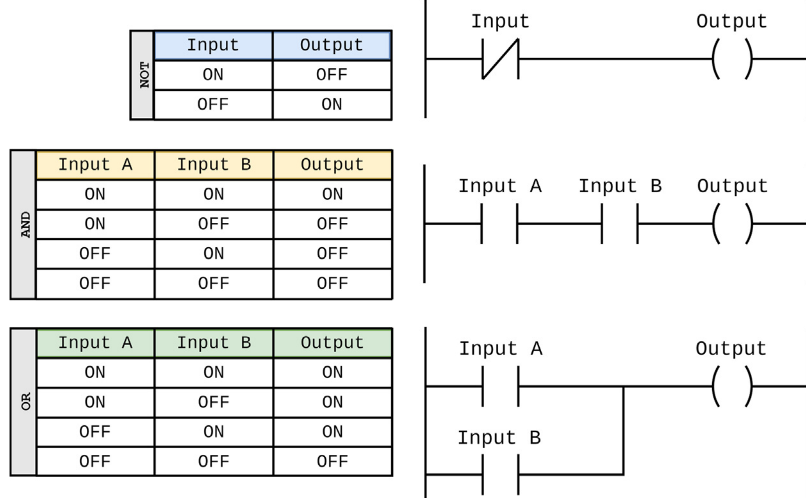
3. Related work

In this section we first discuss PLCs and their historical use of wiring as programs, and how that led to the ladder logic language. We then describe the many educational languages that have been developed to make programming easier for non-experts. Finally, we discuss recent uses of visual languages in industrial settings.

3.1. PLCs

As discussed in Section 2, PLCs were created to avoid the problem of constantly re-wiring relay logic. Instead, PLCs allow engineers to re-program logic virtually, by making updates to the program. Modern

[1] https://www.youtube.com/watch?v=V3xTa2sw0bk.



NOT	Input	Output
	ON	OFF
	OFF	ON

AND	Input A	Input B	Output
	ON	ON	ON
	ON	OFF	OFF
	OFF	ON	OFF
	OFF	OFF	OFF

OR	Input A	Input B	Output
	ON	ON	ON
	ON	OFF	ON
	OFF	ON	ON
	OFF	OFF	OFF

Fig. 4. NOT, AND, and OR operators encoded as Ladder Logic diagrams.

PLCs have been in use since the mid-1970s and standard programming languages, as defined by the IEC 61131-3 standard in 1993, have been in widespread use since the 1980s [56–58]. This standard defines five different languages, three of which are graphical. Ladder Logic, one of these languages, is perhaps the most popular language. It has been claimed that 50% of America's manufacturing capacity has been programmed in Ladder Logic [59] and it is the 50th most popular programming language overall [35].

In spite of its popularity, many have pointed out problems with the Ladder Logic language: there are many issues with race conditions [59] and bad actors can insert logic bombs into programs [60]. Researchers have suggested converting Ladder Logic into other languages as part of the compilation process due to its shortcomings [61] and have suggested it should be replaced with a Petri-net based programming approach [62].

3.2. Educational languages

Educators have made immense steps forward in designing languages that are easy-to-learn and easy-to-use for beginning programmers [6,10]. One of the leading paradigms for making new end-user languages is block-based programming, led by the popularity of tools like Scratch [32] and Alice [63]. Block-based programming uses a programming-command-as-puzzle-piece metaphor to provide visual cues as to how, where, and when a given programming command can be used. Writing programs in block-based environments takes the form of dragging-and-dropping commands together. When two commands cannot be assembled to create a valid programming statement, the environment prevents these commands from being snapped together, thus excluding syntax errors in the authoring of programs. Block-based programming is also becoming an increasingly wide-spread approach to programming physical computing devices and robotics systems. A new wave of robotics toolkits and environments are using the block-based modality, such as OpenRoberta [64], CoBlox [11] and the Modkit [65] and Arduviz [66] environments for programming the Arduino microcontroller.

Research has demonstrated the various ways that block-based tools support novice programmers [28,29] and how they can serve as an effective way to introduce novices to foundational programming practices and computer science ideas [67–70]. Particularly relevant to the work presented in this paper is the comparative research showing the block-based programming approach to be effective for introducing novices to industrial robotics programming [71,72]. Given its success in robotics, it raises the question of whether Ladder Logic should be improved or whether it should be replaced entirely by this new paradigm. We believe that a careful analysis of both Ladder Logic and block-based programming according to our design guidelines coupled with experimental comparisons may be the best way to answer this open question.

3.3. Visual languages in industry

In the 1990s, when several visual programming languages were being introduced in industry, many blindly considered them to be superior to text-based languages. This trend, dubbed superlativism, held that visual languages were superior for all tasks, with no supporting evidence. Not surprisingly, researchers quickly showed that, while visual languages like LabView were well-suited for certain tasks, the were not superior for all tasks [73]. Researchers have since created a foundational set of cognitive dimensions to evaluate visual languages, allowing language designers to better understand the tradeoffs that must be considered [74].

Since that time, many visual languages have been adopted for robotics, both in industry and academia [75]. Behavior-trees, block languages, dataflow languages, state-flow languages, and even trigger-based languages have been used to program robots. Beyond robotics, visual languages have been adopted in many different domains, and have even been analyzed in these settings [76]. A recent survey shows that IoT devices are perhaps the next big focus, as they found as many papers focusing on IoT as they did on education. One such approach, Smart Blocks, which uses block-based programming for smart home devices, uses program analysis to avoid common programming mistakes [77]. For IEC-61131-3 languages researchers have even defined metrics to help measure and avoid overly complex programs [30]. For another common industrial language, LabVIEW, researchers have investigated code smells and their impact on performance [19,20]. The recent focus on these languages, the study of their complexity, and how to use tools like program analysis to make them more usable reinforces our point that, without further help, end-users struggle to use industrial languages.

4. Research method

In the following subsections, we describe our approach and instrumentation. A replication package is available to increase the reproducibility of our study and results.[2]

[2] Replication package: https://github.com/vcuse/industrial.

4.1. Research questions

In our study, we investigated the learnability and usability of Ladder Logic. We further aimed to identify aspects of the language that are particularly problematic for end-users. For this purpose we conducted an interactive online survey with engineering employees working for a large, multi-national engineering conglomerate.

We trained participants through a short video tutorial and then asked them to solve a series of comprehension tasks. These tasks tested both the participants' ability to correctly read and to write Ladder Logic programs. We focused on toy-sized programs that could be read and understood quickly, making them appropriate for our survey-based approach. We studied the performance of participants on these smaller problems and will discuss the implications of our findings on larger programs later.

We further wanted to understand how participants perceive Ladder Logic, so we asked them to rate the usability of Ladder Logic using a standardized *System Usability Scale (SUS)* questionnaire [78]. We further asked participants for open-ended comments on their experience with the language. Some of our participants had already used Ladder Logic before our survey, so we asked them comment on issues they encountered in practice.

To guide our investigation, we focused on four research questions:

RQ1 Can engineers solve automation sub-problems (i.e., building blocks of full solutions) using Ladder Logic?
RQ2 What characteristics of Ladder Logic problems are most challenging for engineers?
RQ3 How easy-to-use and easy-to-learn did engineers rate Ladder Logic?
RQ4 What insights did engineers have concerning the use of Ladder Logic for automation?

We chose these questions to help us understand Ladder Logic's strengths and weaknesses while providing insight to help design future end-user programming languages.

4.2. Survey design

We structured our survey as follows:

4.2.1. Demographic questions

We provided participants with four multiple-choice questions in order to understand the demographic composition of our respondents. The first question asked the participants for their current occupation in industry. The second and third questions asked them for their respective experience, in years, with programming and Programmable Logic Controllers. The fourth question asked the participants which programming languages (if any) they used previously to write code for Programmable Logic Controllers.

4.2.2. Tutorial

After the demographic section, we provided participants with a short 4 min, 20 s video tutorial.[3] This video explained the concepts of Programmable Logic Controllers and Ladder Logic. The tutorial also included examples of Ladder Logic diagrams, an explanation on how to read them, as well as examples of industrial input and output devices.

4.2.3. Exercises

In the third part of our survey, respondents were asked to solve ten short tasks involving simple automation problems. We divided exercises into two groups. The first group contained five exercises focused on writing Ladder Logic diagrams while the second group contained five exercises on reading. We presented these exercises in a fixed order, with alternating questions from each group.

Writing. The goal of the five exercises in the writing group was to identify if the respondents could correctly write a Ladder Logic diagram for a given scenario. For each of the exercises, participants were given a truth table and an animated illustration of a simple automation problem. We then asked the participant to select the diagram out of eight options that correctly solves the proposed problem. To better evaluate the performance of our respondents, we made specific changes in the seven wrong options: We changed the program's structure, the position of labels and the position of symbols. In Fig. 5, we present one of the exercises of this group. The correct answer for this exercise is option "a".

Reading. The goal of the five exercises in the reading group was to identify if the respondents could correctly read and comprehend Ladder Logic diagrams. For each question participants were provided with a diagram and then asked to identify in which cases the output symbol of the diagram would be turned on. We gave participants five possible answers per exercise, each varying in the states for the input symbols that would trigger the output symbol to be turned on. Fig. 6 shows an example of a reading exercise. The correct answer is option "c".

4.2.4. System usability scale

After the set of exercises, participants completed a System Usability Scale (SUS) questionnaire. The SUS is a standardized, straight-forward questionnaire that contains ten statements designed to measure the perception of users on the usability of a system [78,79]. Participants score their agreement with each statement from one *(strongly disagree)* to five *(strongly agree)*. The statements, adapted to contain the Ladder Logic as the respective system under analysis, were shown to the respondents as follows:

- *I think that I would like to use Ladder Logic frequently.*
- *I found Ladder Logic unnecessarily complex.*
- *I thought that Ladder Logic was easy to use.*
- *I think that I would need assistance to use Ladder Logic.*
- *I found the various functions in Ladder Logic were well integrated.*
- *I thought Ladder Logic was too inconsistent.*
- *I would imagine that most people would learn to use Ladder Logic very quickly.*
- *I found Ladder Logic very cumbersome/awkward to use.*
- *I felt very confident using Ladder Logic.*
- *I needed to learn a lot of things before I could get going with Ladder Logic.*

4.2.5. Open-ended question

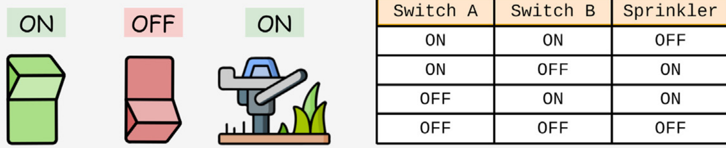
In addition to the system usability scale questionnaire, and in order to obtain a more comprehensive opinion from the participants about Ladder Logic, an open question limited to five thousand characters was provided, asking the respondents: *"What is your opinion about Ladder Logic? Feel free to provide any thoughts about this language"*.

4.3. Survey development

To develop our survey, we followed the three-stage process recommended by Dillman [80]. First, we had experts review our initial survey to ensure clarity and avoid misunderstandings. Next, we discussed the survey with a larger group of researchers and developers, focusing on clarity and motivation. Finally, we performed a pilot test with eleven respondents, including nine researchers (in industrial automation), one developer and one engineering architect, all with at least one year of programming experience. We asked the participants to provide feedback about the survey via email and online meetings. These respondents were not included in the final results.

[3] https://www.youtube.com/watch?v=V3xTa2sw0bk.

Suppose the following automation task containing two switches and one sprinkler, where to turn the sprinkler ON, the two switches must be in opposite states. In other words, while one switch is ON, the other must be OFF to activate the sprinkler. The possible states for the elements of this task can be described by the table and illustration below:



Switch A	Switch B	Sprinkler
ON	ON	OFF
ON	OFF	ON
OFF	ON	ON
OFF	OFF	OFF

Using your understanding of ladder logic, select the diagram that correctly solves this task:

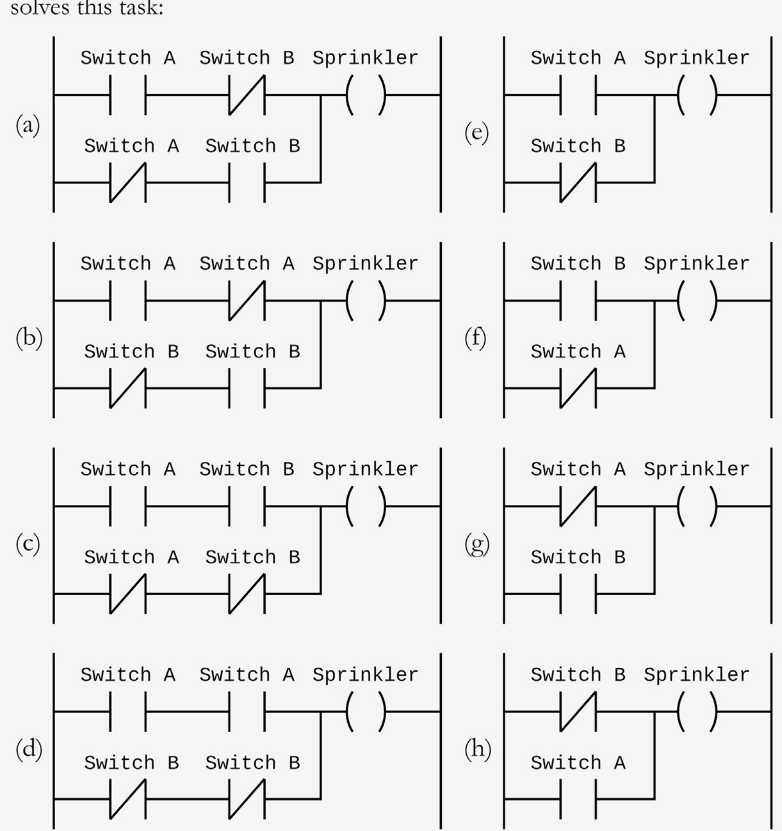


Fig. 5. Example of a writing exercise from the survey.

4.4. Survey execution

The final version of the survey, which is available in its entirety in the replication package, was emailed to employees of a multinational engineering conglomerate. We invited approximately 2,000 employees, including those from an engineering facility in India and those with an engineering-related job title from all over the world.

4.5. Data analysis

4.5.1. Overall performance

Focusing on the ten Ladder Logic exercises in the survey, we analyzed the overall performance of participants. We focused on the number of correct responses that each participant provided. We further analyzed how participants' experience, occupation, and programming experience affected their performance.

4.5.2. Performance per question

Next we evaluated the difficulty of each individual question to see how it contributes to participants' performance. We also grouped questions based on their characteristics, such as the number of rungs, number of symbols, and the type of symbols, to investigate the impact of these factors.

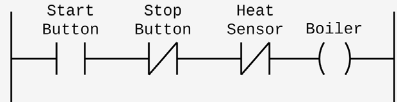
4.5.3. Usability evaluation

We calculated each participant's raw SUS score based on their responses to the standardized questionnaire. SUS scores cannot be interpreted linearly [81]. We therefore mapped the average score that participants gave Ladder Logic to the corresponding percentile of overall scores that other systems evaluated by the SUS received.

4.5.4. Analysis of open-ended responses

To analyze participants' open-ended responses we employed a card sorting approach [82]. Two researchers jointly analyzed and discussed two sets of 10 answers to establish common set of codes. Then, each researcher analyzed the remaining answers independently and discussed the disagreements until reaching consensus. We also employed continuous comparison [83] to ensure the consistency in our codes and results. Finally, a third researcher and an experience engineer inspected the final code classification.

Using your understanding of Ladder Logic, look at the diagram below and select the answer that correctly describes this diagram:



(a) Boiler will be turned ON when at least one of the input elements is ON: Start Button, Stop Button or Heat Sensor.

(b) Boiler will be turned ON when the Start Button and Stop Button are OFF, or when the Heat Sensor is ON.

(c) Boiler will be turned ON when the Start Button is ON, and the Stop Button and Heat Sensor are OFF.

(d) Boiler will be turned ON when the Start Button is OFF, and the Stop Button and Heat Sensor are ON.

(e) Boiler will never be turned ON, regardless the state of the input elements (Start Button, Stop Button and Heat Sensor).

Fig. 6. Example of a reading exercise from the survey.

5. Results

After a brief demographic overview of our participants, we structure the presentation of our results to follow our research questions outlined in Section 4.1.

5.1. Demographics

In total we received 175 complete responses (approx. 9% response rate). Fig. 7 shows the survey participants' self-reported occupation, programming experience, PLC experience, and known PLC programming languages. As expected due to our targeted emails, most (123 of the 175) respondents were engineers. While there were also 25 respondents who selected the "Other" option, these respondents' often wrote in engineering-related jobs such as project managers and project lead engineers. There were also 9 software developers, 9 researchers, 6 technicians and 1 engineering student who responded.

Nearly half of the respondents (100 of 175) claimed to have over 5 years of programming experience, and almost all (153 of 175) claimed at least some experience. Note that while many engineers program as part of their job, it often represents only a small percentage of their overall responsibilities. We therefore do not assume that their experience is equivalent to that of a professional software developer that gave the same response. In addition to programming experience, many respondents had experience programming PLCs directly, with 133 of 175 reporting at least some experience and 67 reporting greater than five years of experience.

Respondents had experience with a variety of languages that are often used in PLC programming. The most popular languages among the respondents were: Function Block Diagrams (FBD), used by 124, Structured Text (ST), used by 92, Ladder Logic (LD), used by 66, and Sequential Function Charts (SFC), used by 60 respondents. Besides the given languages, which were defined by the IEC 61131-3, a few respondents used languages such as C, C++, Flow Code and Control Module Diagrams to program PLCs. 12 respondents also defined themselves as with no experience with programming languages commonly used for PLCs.

5.2. RQ1: Can engineers solve automation sub-problems using ladder logic?

To answer this question we investigated the performance of individuals across the entire survey. As Fig. 8 shows, only about 30% (52 of 175) are able to answer all ten questions correctly, with the average score being 8.5 (SD 1.8). The majority of users (69%) answered at least one question incorrectly, with about 35% answering two or more incorrectly. In Fig. 10 we show the breakdown of correct (green) vs. incorrect (red) responses per question. Most questions had a significant number of users who answered incorrectly (Mean 26.5, SD 15.8), indicating that there was not a single hardest question that caused most participants to fail.

To investigate the influence of participants' background on their performance, we studied the performance of different participant groups. Fig. 9 shows their performance according to each demographic characteristic. These characteristics are experience with IEC-61131-3 languages (top), PLC programming experience (middle), and general programming experience (bottom). As Fig. 9 shows, users' experience has some impact on their performance. To quantify this impact, for each category we grouped users with no experience and users with any experience, and then compared these two groups using the Mann Whitney U Test, a nonparametric test that is widely used to compare non-normal distributions [84]. Using this approach, we saw a difference for all categories: known languages ($W = 553, p = 0.02596$, Cliff's delta: -0.37 medium), PLC experience ($W = 2068, p = 0.01199$, Cliff's delta: -0.24 small), and programming experience ($W = 971, p = 0.001099$, Cliff's delta: -0.41 medium).

> **!** Most participants failed the Ladder Logic competence test. Users with no experience in PLC or general programming performed worse than those with at least some experience.

5.3. RQ2: What characteristics of ladder logic problems are most challenging for engineers?

To investigate this question we categorized each of our ten questions according to the characteristics of Ladder Logic problems. For each characteristic under investigation, we grouped questions into two groups. We investigated many characteristics, but we present only the most relevant comparisons: reading vs. writing-based questions, one-rung vs. two-rung questions, and questions that contained closed-contact inputs vs. those that did not. We then compared the percentage score for each group using the Wilcoxon Signed-Rank Test, where the hypothesis that two groups are identical is rejected for p-values < 0.05. Finally, to measure the effect size we used Cliff's Delta and interpreted the values using accepted thresholds [85].

For the read vs. write comparison we grouped questions according to whether they required users to read Ladder Logic diagrams or whether they required users to write diagrams. Based on the notion that mastery proceeds through the use-modify-create spectrum [86] we expected user performance to be better for reading. The mean performance for the five reading questions was 87.8 compared to 81.9 for the five writing questions, indicating that respondents indeed performed better on reading questions ($W = 17668$, $p = 0.006181$). The effect size between these two means was small (delta estimate= 0.1538286), indicating that while it was harder for users to write Ladder Logic than to read it, the difference was not large, likely because the same knowledge is required for both.

For the one-rung vs. two-rung comparison we grouped questions according to the number of rungs in the questions (for reading problems) or in the correct solution (for writing problems). We expected user performance to be better for one-rung questions because these programs are less complex, having no possibility for intermediate values and being less complex overall. On the five questions with one rung, respondents had a mean score of 87.3 whereas on two-rung questions they scored 82.4, indicating that there was some difference ($W = 17122$, $p = 0.03556$). However, in this case the effect size was negligible (Cliff's delta: $= 0.11$), and thus further investigation, possibly with multiple-rung programs, is needed to establish a clear difference.

When comparing questions with normally-closed contact inputs (i.e. boolean negation) vs. those without, we believed that users would

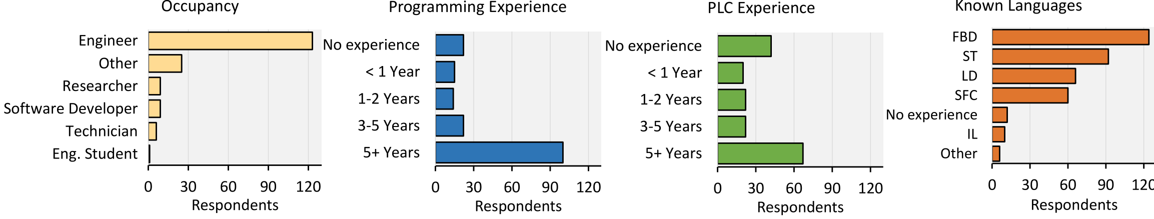


Fig. 7. Demographic composition of participants: Occupancy, programming experience, PLC programming experience and known programming languages.

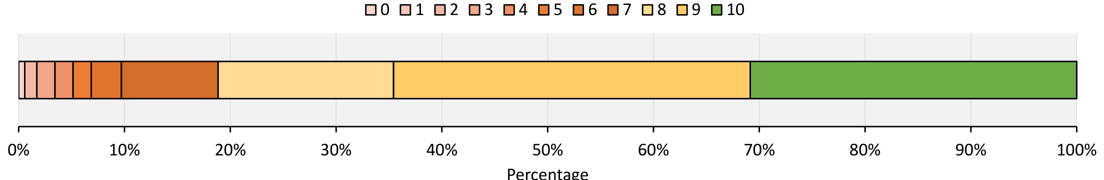


Fig. 8. Breakdown of participant performance (e.g., only about 30%, in green, answered all 10 questions correctly).

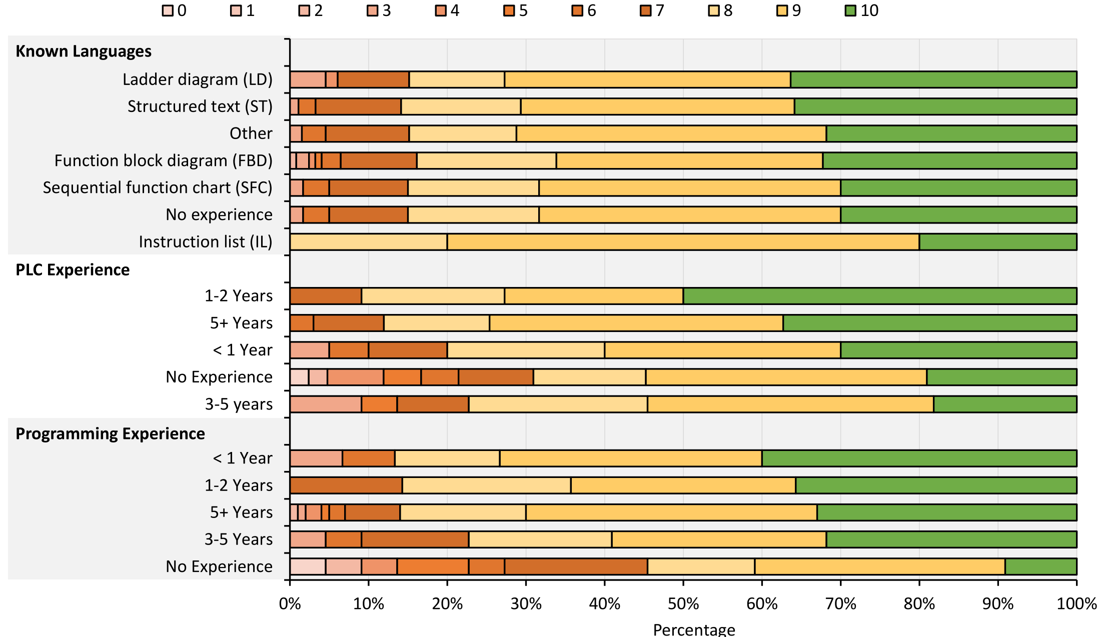


Fig. 9. Breakdown of performance, as effected by previous experience (e.g., only about 10% of those with no programming experience answered all 10 questions correctly, as shown in green on the bottom third, bottom bar labeled “No Experience”).

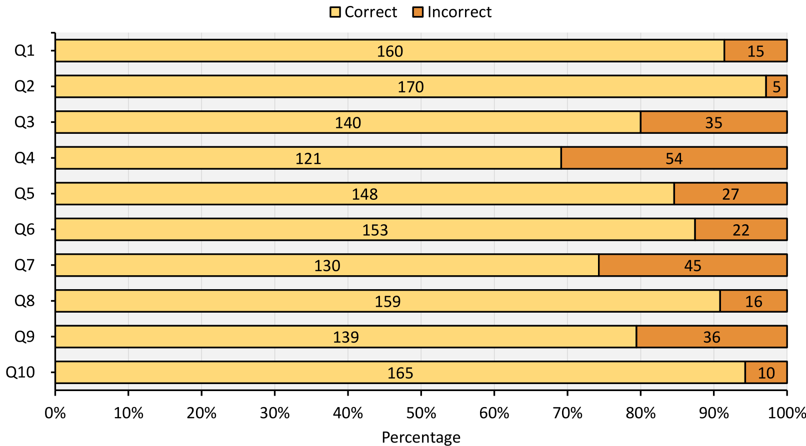


Fig. 10. Performance of respondents for each individual question. The number inside each bar indicates the absolute number of respondents in each group.

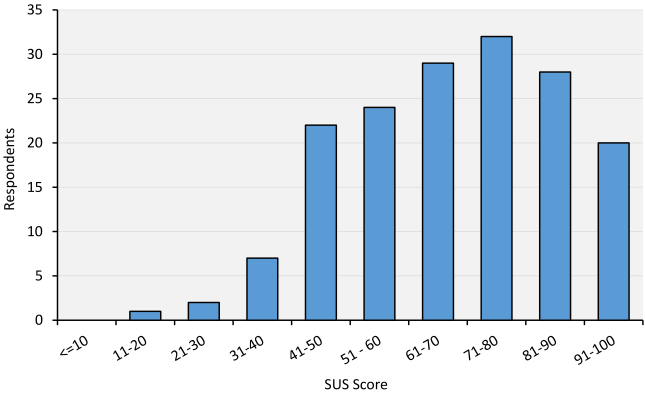


Fig. 11. Distribution of SUS scores.

be more likely to struggle with the former because in our own experience this type of contact is not as intuitive, and because previous work has shown that negation can be challenging for users [87]. On questions with normally-closed contacts respondents had a mean score of 81.3 whereas otherwise they received scores of 93.2, suggesting that normally-closed contacts were particularly difficult for users (W = 830, p = 1.13e−13). The effect size was medium (delta estimate = −0.4075102), indicating that this type of input was the most influential characteristic of Ladder Logic programs on user performance.

> **!** Participants performed slightly better when reading vs. writing programs, worse for more complex programs, and particularly poorly on problems that involved negation.

5.4. RQ3: How easy-to-use and easy-to-learn did engineers rate ladder logic?

We measured participants' preferences using a SUS questionnaire (see Section 4.2.4). Participants gave Ladder Logic a mean score of 69 (SD: 18, max. 100, min. 12.5). Fig. 11 shows the overall distribution of scores that participants gave Ladder Logic. Considering the distribution of SUS ratings for other systems, Ladder Logic's score is only slightly above the average score of 66 [81], placing Ladder Logic in the 50th percentile of systems. According to a popular scale for interpreting SUS scores, mapping scores to adjectives from "worst imaginable" to "best imaginable" Ladder Logic's score maps to "OK" [88]. Another technique for interpreting scores maps it to "marginal" [89]. In both cases both the raw score and the interpreted score are not considered strong.

> **!** Participants rated the usability of Ladder Logic only as "OK" or "marginal", putting the language in the 50th percentile of usability ratings across all systems.

5.5. RQ4: What insights did engineers have concerning the use of ladder logic for automation?

For the open-ended question regarding Ladder Logic we received 98 non-blank answers. To curate the dataset we filtered out 17 messages that did not expressed an opinion about Ladder Logic. For instance we filtered out such as personal history (e.g., "Got training on LD in early days but never used in any project"; "never used before, but I will give it a try in the future"), opinions about the survey (e.g "Very simple and useful teaching. Thumbs up".; "These examples are quite simple, I'd like to see some more complex examples") or other programming language not evaluated in the survey (e.g "I believe ST [structured text] will be future even in PLC world"). Our final dataset contained 79 comments which were, on average, 186 characters long. After performing open coding, several major themes emerged. The top eight themes, in terms of number of occurrences, are listed in Table 1. As shown in this table, the most common theme was that Ladder Logic is not well-suited for large or complex projects.

Table 1
Most common open-ended comments.

Comment	Occurrences
Not well-suited for large/complex projects	27
Easy to use and understand	23
Less appropriate than other programming languages	17
Works well for small/simple projects	17
Easy to learn	9
Hard to debug	6
Difficult to use and understand	5
Bad user experience	5

Ill-Suited for Complex Projects. 27 respondents indicated their agreement with one participants' assessment that "for understanding simple programs like turning [on] motor/lights, Ladder is ok but when we get into complex logics of process control or some complex solution.. [Ladder Logic] is cumbersome & time consuming" or, more directly, "[Ladder Logic is] not suited for complex tasks". These 27 respondents often elaborated on why they believed Ladder Logic did not work for complex situations. Several indicated that Ladder Logic programs quickly become too large, saying "usability in more complex softwares isn't that good as even with simple functions one ends up with huge number of [rungs] which makes the general view of the code more challenging" and "..more complex functions are way too crowded/complex in Ladder Logic". Another elaborated that even to create a simple "..X-Or block I need to use 2 NC & 2 No contact [gates]" and a different respondent explained that "program length also increases due to limitations in the NO/NC contacts in each rung, then in this case we need to do cascading of Rungs (this increases program length)". One respondent summarized the issue as "[Ladder diagrams] takes too much space to program therefore is only suitable for very simple programs".

Another issue that respondents regularly cited as to why Ladder Logic was not well-suited for complex projects is that the data dependencies make programs difficult to understand. One explained: "[Ladder Logic] does not work as well for sequential programming. Trying to write this style of programming often involves a lot intermediate of coils, edge detections and latches, which can be hard for some to follow". Others echoed these challenges, saying "..it gets more complicated when an output function is dependent on another's output function. Then reading the ladder top to bottom becomes tricky in search for the dependable" and "..[it] becomes too difficult to follow for logic functions that interact between each other". In all but toy-sized projects these dependencies are unavoidable.

> **!** Participants stated that the language doesn't scale well for larger programs, and that hidden dependencies make complex diagrams hard to read.

Easy to Use and Learn. Interestingly, another popular theme in comments from respondents was that Ladder Logic was easy to use and understand. For example, one respondent said "Ladder [logic] is simple & easy to learn". However, upon closer inspection, respondents' comments were almost always qualified, such as Ladder Logic is easy to use and learn, but it has key limitations. For instance, one respondent said "Ladder Logic is best for the beginners to learn & understand PLC programming.". but then immediately continued "...it is difficult to

build more complex PLC programming logic using Ladder concept". Similarly, another qualified when Ladder Logic should be used: "Ladder Logic should be used for less complex solution in automation. In this case [it] is clear, readable and simple". Another echoed this, saying "Ladder Logic is good for simple motor run/stop, light on/off logic...[but] can't be used for more complex or coordinated control programming. The complexity of the logic will make the Ladder Logic complex or rather useless". While on its surface this theme appears in conflict with the first theme, supporting details show the findings support each other, as they agree that Ladder Logic is not suited for complex solutions but can be easy to use and easy to learn for simple solutions.

Several respondents argue that Ladder Logic is only a learning language for programming PLCs. One said "..from my point of view ladder diagram is the simplest method for learning PLC" and another "Ladder Logic is best for the beginners to learn & understand PLC programming". However, they seem to disregard it as a serious language. One said: "..it was like basic programming that one learns in his academics ..but when there are more simplified and better languages that can be used in current scenario, no one wants to look towards ladder". Another claimed that "Ladder Logic was designed to get electricians to learn PLC-programming in an easy way" and yet another dismissed it, saying "..to teach the basics of PLC applications, it's obsolete".

> **!** Many participants said that Ladder Logic feels like a toy language that is good for beginners but not for realistic tasks.

Other comments. In addition to the points already mentioned, a few respondents also stated the difficulty to identify bugs in Ladder Logic, giving focus to the idea that such language is not appropriate for large and complex projects. One respondent said: "Initially it is easy understand when for small programs... but when we enter in complex programs it is very difficult to play around and find bugs". Two respondents also mentioned electrical circuits as a prerequisite for understanding Ladder Logic diagrams. One said: "A programmer will need to have knowledge about electric circuits to be able to digest this programming language". The second responded reaffirmed the view of the first one by saying: "If you have experience with electrical diagrams, ladder it's a straight forward language... for programmers without an electrical background, it can get complicated or not so intuitive".

> **!** Some participants pointed out other issues of Ladder Logic, such as poor debugging capabilities or the usage of notation that only electrical engineers can understand easily.

6. Discussion

Ladder Logic is widely assumed to be easy-to-learn and easy-to-use for end-users [46]. Our study results do not support this assumption for industrial end-user programmers: Not only did many participants fail the competence test, but they also rated Ladder Logic's usability as "OK" or "Fair" on the SUS scale. Some of the responses that we described in Section 5.5 further indicate that participants would find Ladder Logic difficult to apply to real-world projects. In this section we discuss some of the aspects of Ladder Logic that the 175 industrial end-user participants in our study struggled with. We further present several language design observations derived from our participants performance and feedback that could improve the programming effectiveness of these industrial end-users.

6.1. Using domain-specific notation

The first systems that used Ladder Logic as a visual programming language were introduced in the 1970s. At that time, the language appealed to engineers and technicians due to its closeness to the domain-specific notation for relay diagrams. This similarity reduced the learning effort for these user groups since they could apply their existing knowledge of circuit design directly without having to learn a new notation.

Over the subsequent decades, the popularity of hard-wired relays has declined [37]. This has led to fewer engineers being familiar with the specific notation that was used by relay diagrams. The consequences of this was apparent throughout our study: multiple participants indicated that they found it difficult to read programs, particularly when they use negation.

Although Ladder Logic's history may explain why its notation became obsolete, this raises the question: *is it possible to design languages to be more future-proof?* Ladder Logic's heavy reliance on symbols that only have a meaning for users with domain-specific knowledge seems to be a particular weakness in this regard. We believe that a lesson that can be learned from Ladder Logic is that designers of new languages should select symbols and notation with caution, especially with an eye to those symbols that are unlikely to change or lose their meaning over time. If it is not possible to find symbols that are universally known, it can further be useful to support domain-specific notation with text.

Previous work has established some best practices for designing block-based programming languages. Some of these findings might be applicable to languages like Ladder Logic as well. For example, recent work has found that end-users prefer text-based commands over icons, even if they can understand both notations equally well [90]. Some popular block-based languages make programs look very similar to natural language [32]; some let users choose between a more domain-specific notation or one that is closer to natural language [63]; many of them have in common that they only use a small number of visual features that make them accessible to their target audience.

> End-user languages with high usability ratings often combine a small number of carefully selected visual features with text. In Ladder Logic, the reliance on a notation that fell out of favor has made the language harder to understand and use.

6.2. Scalability

Many study participants with experience in Ladder Logic commented that programs quickly become too complex to read. They specifically complained about the large number of rungs that even relatively simple programs require. We identified two separate usability issues that users face as their programs grow: First, as each rung requires significant vertical space, only a certain number of rungs fit on a single screen. While users can zoom out or scroll through the rungs, it becomes increasingly hard to maintain an overview of the whole program. Second, because many rungs, or lines of code, are similar due to the regularity of Ladder Logic programs, it becomes hard for users to identify the specific part of a program that they want to edit.

Another observation is that, when a notation designed for another purpose, like detailing relay diagrams, is co-opted as a programming notation, this adaptation can have drawbacks. For example, Ladder Logic's lack of scalability only became a problem when Ladder Logic was adopted as a programming language. When these diagrams were simply printed (on paper), engineers could easily compare multiple diagrams by laying them out side-by-side. They could organize subsystems into separate binders, effectively creating modules. However,

when used as a programming language, Ladder Logic is missing these features.

We believe that Ladder Logic illustrates the importance of not just considering the design of a language itself but also the context in which it is used. This can also benefit language designers, since development environments can complement a language and overcome its weak points. For example, many integrated development environments for professional programming languages aid visibility and navigation in ways that a language could not offer directly. Ladder Logic, many other visual languages, could also benefit from a development environment that provides such features.

Industrial end-user programmers may have different scalability needs than other users of similar notations. Development environments can fill this gap by providing additional aids that improve visibility and navigation.

6.3. Encapsulation and code reuse

When developers design a new end-user language, visibility should be ideally considered early-on. One reason why Ladder Logic programs grow fast and become hard to read is that the language has limited support for encapsulation or code reuse. While Ladder Logic supports *add-on instructions* that allow users to reuse code [91], our survey responses show that many experienced Ladder Logic users are not aware of this possibility. We found that add-on instructions are rarely covered in end-user manuals and tutorials, and are particularly not recommended as a way to structure code. We therefore see them more as an after-thought in Ladder Logic's design than a first-class feature.

When designing new languages for end-users, it might be challenging to find ways to reuse code that work for them. For example, while the popular block-based language Scratch supports user-defined procedure blocks, only a small percentage of programs use the feature [92]. Instead, many end-user programmers rely on code clones that are prone to causing the same visibility issues that are found in Ladder Logic [93]. It is not well-understood why end-users rarely use procedures in Scratch. One reason might be that they are not aware of the feature since many tutorials don't cover it sufficiently. Another explanation could be that the feature requires too much up-front effort to use while its benefits only play out over time. It remains a challenge for designers to find ways to encourage end-users to reuse code in their language.

Designers may need to consider encapsulation and code reuse early on and advertise them prominently in their end-user languages. Otherwise, they might risk that users miss these features or do not see their benefits.

6.4. Implicit dependencies

Our survey results show that even small Ladder Logic programs can be difficult to read or write. Participants stated that they find it difficult to understand programs where the output of one rung is used as an input in others. They described that they frequently miss the implicit dependency between such rungs when reading programs. They further explained that they find it hard to write programs with this type of dependencies since they do not have a good intuition for the resulting behavior.

Unfortunately, the design of Ladder Logic makes it hard to avoid dependencies between rungs. In particular, re-using any intermediate results in Ladder Logic requires users to connect them to an output and use it as an input in another rung or, worse yet, store values in registers. Unlike variables in traditional programming languages, registers are always accessible globally and can be read and written at any point. Therefore, when a user modifies one rung of a program, any other part of the program might be potentially affected. Since Ladder Logic rungs are continuously evaluated, cycling through the program hundreds of times per second, changes to registers (i.e., global variables) can even affect previous rungs, or introduce unintended dependency cycles.

Programming environments can help users visualize and understand dependencies. As previously discussed, solutions to make Ladder Logic more scalable, like highlighting rungs with the same elements, can also help to make dependencies explicit. Marking cyclic dependencies and automated refactoring of rungs into dependency order may make programs easier to read. However, programming environments can only fight the symptoms of underlying language design issues: Ladder Logic has very limited support for scoping or modularity. As we know from the study of other languages, these features are essential to program usability, and an end-user language should actively encourage the use of scoping and modularity. Designers of new languages that model dependencies or data flow need to incorporate features that support end-users in structuring their programs.

Development environments can visualize implicit dependencies, but this aid has limitations. Languages can therefore support industrial end-users by making dependencies explicit and allowing them to manually structure their programs.

7. Threats to validity

As with most survey-based investigations, a primary threat to external validity is the appropriateness of the surveyed population. Our hypothesis was that Ladder Logic programming should be understandable to any engineering employee in an engineering company. To minimize this threat, we emailed surveys only to those with engineering-based jobs (according to their official job titles) in a multi-national engineering conglomerate. As shown in Fig. 7, all self-reported job titles were explicitly engineering or another technical role.

Another threat to external validity is an ecological threat: can the results of studying this single industrial end-user language generalize to other industrial end-user languages. Ladder Logic shares many characteristics with other industrial end-user languages such as LabView and other IEC 61131-3 languages. It is a visual language, it was designed to be easy-to-learn and easy-to-use by non-programmers, it was not designed or studied by programming language experts, and it is in widespread industrial use. Because of these parallels, many findings are likely to generalize to other languages; however, Ladder Logic-specific issues, such as its notation being familiar only to electrical engineers, will not.

A threat to internal validity centers around the quality of our training. If our training video was unclear or confusing, it could hinder our participants ability to answer Ladder Logic questions. We mitigated this threat by focusing only on the most basic concepts in Ladder Logic in the training video and subsequent survey, in addition to piloting and improving our survey and training prior to deployment. Data from our experiment suggests that our training was effective. As we can see from Fig. 9, those with prior knowledge of Ladder Logic (top row on top group) performed only slightly better (analysis in Section 5.2) than those with no prior knowledge of Ladder Logic (all other rows in top group), including those with no experience with PLCs.

One more threat to internal validity was found in the reliability of the questionnaire applied in this study. Using the Cronbach's Alpha test in our set of answers for the SUS questionnaire, we detected that the

alpha value found was slightly low ($alpha = 0.327$), suggesting some potential bias in our results. Since the SUS questionnaire is based on a set of standard usability questions that are widely used by many other studies with great Cronbach's Alpha scores [89], we decided to keep such results as part of our study.

Fortunately, there are no large threats to the construct validity of this experiment. Our primary results measure the performance of technical employees on reading and understanding Ladder Logic diagrams, which matches exactly the construct we intended to measure. The use of multiple choice answers could introduce a minor threat, as it is difficult to differentiate between respondents that answered randomly (and thus mainly incorrectly) and those that were answering authentically, but simply performing poorly. Fortunately, almost all respondents answered three or more questions correctly, which indicates that they did not select answers arbitrarily. Another threat is that users could accidentally choose wrong answers (i.e., a mis-click). This threat was minimized by allowing respondents to revisit questions and update their responses.

Finally, there is a possible threat to our conclusion validity. One of our major conclusions posits that most respondents fail our test, answering at least one of ten simple questions incorrectly. It is possible that this standard is too high, that engineers are likely to make a mistake on one of ten questions on a survey, even if they known and understand the material. We reject this threat, as engineers tend to be meticulous in their work and in their responses. Furthermore, if engineers make one mistake for every ten Ladder Logic programs they read, then programming a realistic-sized system would be impossible. Thus, we dismiss this threat.

8. Conclusion

In our study, we evaluated the Ladder Logic programming language by surveying industrial engineers. We found that although popular among users of Programmable Logic Controllers, Ladder Logic represents the essence of what is available in industrial end-user programming: a set of outdated and counter-intuitive visual programming languages that have become popular due to a lack of alternatives rather than their high quality. In this paper we showed that nearly 70% of end-user industrial engineers who answered our survey made substantive errors when trying to solve simple automation problems using Ladder Logic, even if they had significant prior experience with that language. Among the difficulties observed, we noticed that certain aspects of Ladder Logic are more challenging than others for most users, such as the concept of negation.

We also investigated the usability of Ladder Logic via a system usability questionnaire, where we found that Ladder Logic receives a "Fair" rating from users. Last but not least, we also asked our respondents to openly discuss their concerns about Ladder Logic. The respondents pointed out key limitations of Ladder Logic, including difficulties in scalability, dependency management, and modularity, summarizing Ladder Logic as a language that should be used only for educational purposes.

Based on our findings and feedback from industrial end-user engineers, we identified four design observations for future language and tool designers to consider when creating languages for industrial end-users. We hope these observations can help improve the correctness of industrial end-user programs which could ultimately improve their productivity in this increasingly important domain.

We see our findings as supplementing existing evidence of the shortcomings of industrial end-user languages. Adding to the pioneering work analyzing LabVIEW [19,20], another widely popular industrial end-user language, our work shows that another major language is failing its users. In General Electric's advertising campaign, "What's the Matter with Owen?", the protagonist, a young computer scientist, is faced with convincing his friends that working for an industrial company, one that produces our power, creates physical products, and treats our water, is just as important as working for a FAANG company [94]. Unfortunately, his friend, who works for Zazzies, the mobile app which allows users to put hats made of fruit on cute animals, steals the show from Owen, and receives all the attention from their peers.[4] Our work is driven by this fear, that software engineering research overlooks industrial end-users, focusing on more traditional software development teams and tools. We hope that this work demonstrates that industrial end-users face real problems, but that solutions can be found, and drives more research interest towards this important field.

CRediT authorship contribution statement

Felipe Fronchetti: Conceptualization, Methodology, Investigation, Data curation, Formal analysis, Validation, Writing – original draft, Writing – review & editing, Supervision. **Nico Ritschel:** Conceptualization, Methodology, Investigation, Formal analysis, Validation, Writing – original draft, Writing – review & editing. **Reid Holmes:** Conceptualization, Methodology, Investigation, Formal analysis, Validation, Writing – original draft, Writing – review & editing. **Linxi Li:** Data curation, Formal analysis, Validation, Writing – review & editing. **Mauricio Soto:** Project administration, Funding acquisition. **Raoul Jetley:** Project administration, Funding acquisition. **Igor Wiese:** Data curation, Formal analysis, Validation, Writing – review & editing. **David Shepherd:** Conceptualization, Methodology, Investigation, Formal analysis, Validation, Writing – original draft, Writing – review & editing, Project administration, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This study is supported by the National Science Foundation, USA under Grant NRI-2024561.

References

[1] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, Susan Wiedenbeck, The state of the art in end-user software engineering, ACM Comput. Surv. 43 (3) (2019) http://dx.doi.org/10.1145/1922649.1922658, URL https://doi.org/10.1145/1922649.1922658.

[2] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, Alfonso Pierantonio, Supporting the understanding and comparison of low-code development platforms, in: Proceedings of the Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2019, pp. 171–178.

[3] Felicien Ihirwe, Davide Di Ruscio, Silvia Mazzini, Pierluigi Pierini, Alfonso Pierantonio, Low-code engineering for internet of things: A state of research, in: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, 2019, pp. 1–8.

[4] Raquel Sanchis, Óscar García-Perales, Francisco Fraile, Raul Poler, Low-code as enabler of digital transformation in manufacturing industry, Appl. Sci. 10 (1) (2019) 12.

[5] John R Rymer, Rob Koplowitz, Salesforce Are Leaders, Kony Mendix, Salesforce are Leaders, GeneXus ServiceNow, Strong Performers, WaveMaker MatsSoft, Thinkwise are Contenders, The Forrester Wave™: Low-Code Development Platforms For AD&D Professionals, Q1 2019, Forrester Research, 2019.

[6] Caitlin Kelleher, Randy Pausch, Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers, ACM Comput. Surv. 37 (2) (2019) 83–137.

[7] Christopher Scaffidi, Andrew Dove, Tahmid Nabi, LondonTube: Overcoming hidden dependencies in cloud-mobile-web programming, in: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, 2019, pp. 3498–3508.

[4] An exemplar commercial from this campaign: https://www.youtube.com/watch?v=SW_7Q_tAk9I.

[8] Nan Zang, Mary Beth Rosson, Vincent Nasser, Mashups: who? what? why? in: CHI'08 Extended Abstracts on Human Factors in Computing Systems, 2019, pp. 3171–3176.
[9] Sean McDirmid, Living it up with a live programming language, ACM SIGPLAN Not. 42 (10) (2019) 623–638.
[10] Chris Proctor, Paulo Blikstein, Grounding how we teach programming in why we teach programming, in: Constructionism in Action, 2019, pp. 127–134.
[11] David Weintrop, Uri Wilensky, Comparing block-based and text-based programming in high school computer science classrooms, Trans. Comput. Educ. (TOCE) 18 (1) (2019) 1–25.
[12] P Vincent, K Lijima, Mark Driver, Jason Wong, Yefim Natis, Magic quadrant for enterprise low-code application platforms, Retrieved December, 18, 2019, p. 2019.
[13] Simon Peyton Jones, Alan Blackwell, Margaret Burnett, A user-centred approach to functions in excel, in: Proceedings of the International Conference on Functional Programming (ICFP), 2019, pp. 165–176.
[14] Robin Abraham, Martin Erwig, Steve Kollmansberger, Ethan Seifert, Visual specifications of correct spreadsheets, in: In Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2019, pp. 189–196.
[15] Barbara Rita Barricelli, Fabio Cassano, Daniela Fogli, Antonio Piccinno, End-user development, end-user programming and end-user software engineering: A systematic mapping study, J. Syst. Softw. 149 (2019) 101–137, http://dx.doi.org/10.1016/j.jss.2018.11.041.
[16] Michael Tiegelkamp, Karl-Heinz John, IEC 61131-3: Programming Industrial Automation Systems, Springer, 2019.
[17] Jeffrey Travis, LabVIEW for Everyone, Pearson Education, 2019.
[18] Steven T Karris, Introduction to Simulink with Engineering Applications, Orchard Publications, 2019.
[19] Christopher Chambers, Christopher Scaffidi, Utility and accuracy of smell-driven performance analysis for end-user programmers, J. Vis. Lang. Comput. 26 (2019) 1–14.
[20] Christopher Chambers, Christopher Scaffidi, Impact and utility of smell-driven performance tuning for end-user programmers, J. Vis. Lang. Comput. 28 (2019) 176–194.
[21] c3controls, PLC programming then & now: the history of PLCs, 2019, URL https://www.c3controls.com/white-paper/history-of-programmable-logic-controllers/.
[22] Michael Tiegelkamp, Karl-Heinz John, IEC 61131-3: Programming Industrial Automation Systems, Vol. 14, Springer, 2019.
[23] Edouard Tisserant, Laurent Bessard, Mário de Sousa, An open source IEC 61131-3 integrated development environment, in: Proceedings of the International Conference on Industrial Informatics, 1, 2019, pp. 183–187.
[24] Daniel D Hils, Visual languages and computing survey: Data flow visual programming languages, J. Vis. Lang. Comput. 3 (1) (2019) 69–101.
[25] Bryan J Smith, Conceptual graphs as a visual programming language for teaching programming, in: Proceedings of the International Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2019, pp. 258–259.
[26] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, Evelyn Eastmond, The scratch programming language and environment, ACM Trans. Comput. Educ. (TOCE) 10 (4) (2019) 1–15.
[27] Clifford J Peshek, Michael T Mellish, Recent developments and future trends in PLC programming languages and programming tools for real-time control, in: [1993] Record of Conference Papers Cement Industry Technical, 2019, pp. 219–230.
[28] Satabdi Basu, Using rubrics integrating design and coding to assess middle school students' open-ended block-based programming projects, in: Proceedings of the Technical Symposium on Computer Science Education (SIGCSE), 2019, pp. 1211–1217.
[29] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, Franklyn Turbak, Learnable programming: Blocks and beyond, Commun. ACM (CACM) 60 (6) (2019) 72–80.
[30] Juliane Fischer, Birgit Vogel-Heuser, Heiko Schneider, Nikolai Langer, Markus Felger, Matthias Bengel, Measuring the overall complexity of graphical and textual IEC 61131-3 control software, IEEE Robot. Autom. Lett. (2019).
[31] Herbert Prähofer, Florian Angerer, Rudolf Ramler, Friedrich Grillenberger, Static code analysis of IEC 61131-3 programs: Comprehensive tool support and experiences from large-scale industrial application, IEEE Trans. Ind. Inf. 13 (1) (2019) 37–47.
[32] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al., Scratch: Programming for all, Commun. ACM (CACM) 52 (11) (2019) 60–67.
[33] Mordor Intelligence, Programmable logic controller (PLC) market - growth, trends, COVID-19 impact, and forecasts (2021 - 2026), 2019, URL https://www.reportlinker.com/p06062818/Programmable-Logic-Controller-PLC-Market-Growth-Trends-COVID-19-Impact-and-Forecasts.html.
[34] Jeremy R Pollard, Ladder logic remains the PLC language of choice, Control Eng. 41 (5) (2019) 77–79.
[35] Nick Diakopoulous, Mythili Bagavandas, Gurdeep Singh, Preeti Kulkarni, Interactive: The top programming languages, 2019, URL https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020.
[36] C. Edvard, 4 most popular PLC programming languages for implementation of control diagrams, Electr. Eng. Portal (2019).
[37] T.R. Alves, M. Buratto, F.M. de Souza, T.V. Rodrigues, OpenPLC: an open source alternative to automation, in: Proceedings of the Global Humanitarian Technology Conference (GHTC), 2019, pp. 585–589.
[38] Kevin Brock, The 'FizzBuzz' programming test: A case-based exploration of rhetorical style in code, Comput. Cult. (5) (2019).
[39] Rainer Bischoff, Arif Kazi, Markus Seyfarth, The MORPHA style guide for icon-based programming, in: Proceedings. 11th IEEE International Workshop on Robot and Human Interactive Communication, IEEE, 2019, pp. 482–487.
[40] Martin Markovič, Uporaba Robotskega in Operativnega Sistema Za Programiranje Kolaborativnega Robota Franka Emika (Ph.D. thesis), Univerza v Ljubljani, Fakulteta za strojništvo, 2019.
[41] Gregory F Rossano, Carlos Martinez, Mikael Hedelind, Steve Murphy, Thomas A Fuhlbrigge, Easy robot programming concepts: An industrial perspective, in: 2013 IEEE International Conference on Automation Science and Engineering, CASE, IEEE, 2019, pp. 1119–1126.
[42] Frank Lamb, Automation and manufacturing, Industrial Automation: Hands on, McGraw-Hill Education LLC, New York, N.Y, 2019.
[43] James F. Lea, Rowlan Lynn Jr., Programmable logic controllers, chapter 14.3.4.2, Gas Well Deliquification, third ed., Elsevier, 2019, URL https://app.knovel.com/hotlink/khtml/id:kt01226C76/gas-well-deliquification/programmable-logic-controllers.
[44] Jon L. Dossett, George E. Totten, Advantages of task-specific controllers, chapter 9.4.1.1, ASM Handbook, Volume 04B - Steel Heat Treating Technologies, ASM International, 2019, URL https://app.knovel.com/hotlink/khtml/id:kt00U4H0M2/asm-handbook-volume-4b/advantages-task-specific.
[45] K.K. Appuu Kuttan, Programmable logic controller, chapter 5.8, in: Introduction to Mechatronics, Oxford University Press, 2019, URL https://app.knovel.com/hotlink/khtml/id:kt008VP3P2/introduction-mechatronics/programmable-logic-controller.
[46] K.T. Erickson, Programmable logic controllers, IEEE Potentials 15 (1) (2019) 14–17, http://dx.doi.org/10.1109/45.481370.
[47] Edward H. Smith, Ladder logic programming, chapter 3.8.3, in: Mechanical Engineer's Reference Book, twelfth ed., 2019, URL https://app.knovel.com/hotlink/khtml/id:kt002YJTS1/mechanical-engineers/ladder-logic-programming.
[48] A.K. Gupta, S.K. Arora, Jean Riescher Westcott, The input/output module, chapter 11.14.1.1, Industrial Automation and Robotics, Mercury Learning and Information, 2017, https://app.knovel.com/hotlink/khtml/id:kt0119K1R1/industrial-automation/input-output-module.
[49] C.L. Albert, D.A. Coggan, Boolean logic, chatper 9.9.4, Fundamentals of Industrial Control, second ed., ISA, 2019, URL https://app.knovel.com/hotlink/khtml/id:kt00BHDAK1/fundamentals-industrial/boolean-logic.
[50] D. Koshal, Ladder logic programming, chapter 11.7.3, Manufacturing Engineer's Reference Book, Elsevier, 2019, URL https://app.knovel.com/hotlink/khtml/id:kt002UQ2O7/manufacturing-engineers/ladder-logic-programming.
[51] Kelvin T Erickson, Programmable Logic Controllers: an Emphasis on Design and Application, Dogwood Valley Press, 2019.
[52] Austin Scott, Ladder logic overview, chapter 5.1, in: Learning RSLogix 5000 Programming, Packt Publishing, 2019, URL https://app.knovel.com/hotlink/khtml/id:kt011DLXT2/learning-rslogix-5000/ladder-logic-overview.
[53] Nicholas P. Sands, Ian Verhappen, Ladder logic, chapter 4.2, Guide to the Automation Body of Knowledge, third ed., ISA, 2019, URL https://app.knovel.com/hotlink/khtml/id:kt011NZ7V8/guide-automation-body/ladder-logic.
[54] S. Bobby Rauf, Relay ladder logic, chapter 10.12, Electrical Engineering for Non-Electrical Engineers, second ed., Fairmont Press Inc. 2019, URL https://app.knovel.com/hotlink/khtml/id:kt0113HO8V/electrical-engineering/relay-ladder-logic.
[55] Philip D. Rufe, Programming, chapter 43.4, Fundamentals of Manufacturing, third ed., Society of Manufacturing Engineers (SME), 2019, URL https://app.knovel.com/hotlink/khtml/id:kt00BKEXP2/fundamentals-manufacturing/programming.
[56] Thomas A. Hughes, Measurement and control basics (4th edition), chapter 11.6: International standard for PLC languages, in: Measurement and Control Basics, fourth ed., ISA, 2019, URL https://app.knovel.com/hotlink/khtml/id:kt00C8NPD2/measurement-control-basics/international-standard.
[57] Nicholas P. Sands, Ian Verhappen, Guide to the automation body of knowledge (3rd edition), chapter 16.7: The languages, Guide to the Automation Body of Knowledge, third ed., ISA, 2019, URL https://app.knovel.com/hotlink/khtml/id:kt011NZFE3/guide-automation-body/the-languages.
[58] Haralambos Mouratidis, Software engineering for secure systems: Industrial and research perspectives, chapter 12.2.2: PLC and IEC 61131-3, Software Engineering for Secure Systems: Industrial and Research Perspectives, IGI Global, 2019, URL https://app.knovel.com/hotlink/khtml/id:kt00U7UO21/software-engineering/plc-and-iec-61131-3.
[59] Alexander Aiken, Manuel Fähndrich, Zhendong Su, Detecting races in relay ladder logic programs, in: Proceedins of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2019, pp. 184–200.
[60] Naman Govil, Anand Agrawal, Nils Ole Tippenhauer, On ladder logic bombs in industrial control systems, in: Computer Security, Springer, 2019, pp. 110–126.
[61] Albert Falcione, Bruce H Krogh, Design recovery for relay ladder logic, IEEE Control Syst. Mag. 13 (2) (2019) 90–98.

[62] Kurapati Venkatesh, MengChu Zhou, Reggie J Caudill, Comparing ladder logic diagrams and Petri nets for sequence controller design through a discrete manufacturing system, IEEE Trans. Ind. Electron. 41 (6) (2019) 611–619.
[63] Stephen Cooper, Wanda Dann, Randy Pausch, Alice: A 3-D tool for introductory programming concepts, J. Comput. Sci. Coll. 15 (5) (2019) 107–116.
[64] Markus Ketterl, Beate Jost, Thorsten Leimbach, Reinhard Budde, Tema 2: Open roberta - a web based approach to visually programming real educational robots, in: Tidsskriftet LæRing Og Medier (LOM), Vol. 8, No. 14, 2019.
[65] Amon Millner, Edward Baafi, Modkit: Blending and extending approachable platforms for creating computer programs and interactive objects, in: Proceedings of the International Conference on Interaction Design and Children, 2019, pp. 250–253.
[66] Adin Baskoro Pratomo, Riza Satria Perdana, Arduviz, a visual programming IDE for arduino, in: Proceedings of the International Conference on Data and Software Engineering (ICoDSE), 2019, pp. 1–6.
[67] Diana Franklin, Gabriela Skifstad, Reiny Rolock, Isha Mehrotra, Valerie Ding, Alexandria Hansen, David Weintrop, Danielle Harlow, Using upper-elementary student performance to understand conceptual sequencing in a blocks-based curriculum, in: Proceedings of the Technical Symposium on Computer Science Education (SIGCSE), 2019, pp. 231–236.
[68] Shuchi Grover, Satabdi Basu, Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic, in: Proceedings of the Technical Symposium on Computer Science Education (SIGCSE), 2019, pp. 267–272.
[69] Thomas W Price, Tiffany Barnes, Comparing textual and block interfaces in a novice programming environment, in: Proceedings of the International Computing Education Research Conference (ICER), 2019, pp. 91–99.
[70] Zhen Xu, Albert D Ritzhaupt, Fengchun Tian, Karthikeyan Umapathy, Block-based versus text-based programming environments on novice student learning outcomes: A meta-analysis study, Comput. Sci. Educ. 29 (2–3) (2019) 177–204.
[71] Tracey Booth, Simone Stumpf, End-user experiences of visual and textual programming environments for arduino, in: International Symposium on End User Development (IS-EUD), 2019, pp. 25–39.
[72] José María Rodríguez Corral, Iván Ruiz-Rube, Antón Civit Balcells, José Miguel Mota-Macías, Arturo Morgado-Estévez, Juan Manuel Dodero, A study on the suitability of visual languages for non-expert robot programmers, IEEE Access 7 (2019) 17535–17550.
[73] Thomas RG Green, Marian Petre, Rachel KE Bellamy, Comprehensibility of visual and textual programs: A test of superlativism against the'match-mismatch'conjecture, in: Empirical Studies of Programmers: Fourth Workshop, Vol. 121146, Ablex Publishing, Norwood, NJ, 2019.
[74] Thomas R.G. Green, Marian Petre, Usability analysis of visual programming environments: a 'cognitive dimensions' framework, J. Vis. Lang. Comput. 7 (2) (2019) 131–174.
[75] Enrique Coronado, Fulvio Mastrogiovanni, Bipin Indurkhya, Gentiane Venture, Visual programming environments for end-user development of intelligent and social robots, a systematic review, J. Comput. Lang. 58 (2019) 100970.
[76] Mohammad Amin Kuhail, Shahbano Farooq, Rawad Hammad, Mohammed Bahja, Characterizing visual programming approaches for end-user developers: A systematic review, IEEE Access (2019).
[77] Nayeon Bak, Byeong-Mo Chang, Kwanghoon Choi, Smart block: A visual block language and its programming environment for IoT, J. Comput. Lang. 60 (2019) 100999.
[78] John Brooke, SUS: A retrospective, J. Usability Stud. 8 (2) (2019) 29–40.
[79] Rex Hartson, Pardha S. Pyla, The system usability scale (SUS), chapter 12.5.3.3, in: UX Book - Process and Guidelines for Ensuring a Quality User Experience, Elsevier, 2019, URL https://app.knovel.com/hotlink/khtml/id:kt00BP2G59/ux-book-process-guidelines/system-usability-scale.
[80] Don A Dillman, Mail and Internet Surveys: the Tailored Design Method–2007 Update with New Internet, Visual, and Mixed-Mode Guide, John Wiley & Sons, 2019.
[81] James R Lewis, Jeff Sauro, The factor structure of the system usability scale, in: Proceedings of the International Conference on Human Centered Design, 2019, pp. 94–103.
[82] Donna Spencer, Card Sorting: Designing Usable Categories, Rosenfeld Media, 2019.
[83] Anselm L. Strauss, Juliet M. Corbin, Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory, Sage Pub, Thousand Oaks, 2019.
[84] Daniel S Wilks, Statistical Methods in the Atmospheric Sciences, Vol. 100, Academic Press, 2019.
[85] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the NSSE and other surveys, in: Proceedings of the Annual Meeting of the Florida Association of Institutional Research, 2019, pp. 1–33.
[86] Irene Lee, Fred Martin, Jill Denner, Bob Coulter, Walter Allan, Jeri Erickson, Joyce Malyn-Smith, Linda Werner, Computational thinking for youth in practice, ACM Inroads 2 (1) (2019) 32–37.
[87] John F Pane, Brad A Myers, Tabular and textual methods for selecting objects from a group, in: Proceeding 2000 IEEE International Symposium on Visual Languages, IEEE, 2019, pp. 157–164.
[88] Aaron Bangor, Philip Kortum, James Miller, Determining what individual SUS scores mean: Adding an adjective rating scale, J. Usability Stud. 4 (3) (2019) 114–123.
[89] Aaron Bangor, Philip T Kortum, James T Miller, An empirical evaluation of the system usability scale, Int. J. Hum. Comput. Interact. 24 (6) (2019) 574–594.
[90] Nico Ritschel, Vladimir Kovalenko, Reid Holmes, Ronald Garcia, David C. Shepherd, Comparing block-based programming models for two-armed robots, IEEE Trans. Softw. Eng. (TSE) (2019) 1, http://dx.doi.org/10.1109/TSE.2020.3027255.
[91] Neal Babcock, PLC programming with RSLogix 5000, in: Modern Media, Toronto, Canada, 2019.
[92] Alaaeddin Swidan, Alexander Serebrenik, Felienne Hermans, How do scratch programmers name variables and procedures? in: Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM), 2019, pp. 51–60.
[93] Gregorio Robles, Jesús Moreno-León, Efthimia Aivaloglou, Felienne Hermans, Software clones in scratch projects: On the presence of copy-and-paste in computational thinking learning, in: Proceedings of the International Workshop on Software Clones (IWSC), 2019, pp. 1–7.
[94] Rob Salkowitz, The Alarming Economic Trend Behind Ge's Odd Ad Campaign, Forbes, 2019.