An Inventory of Goals from CS1 Programs Processing a Data Series

Cruz Izu
The University of Adelaide
Adelaide, Australia
cruz.izu@adelaide.edu.au

Violetta Lonati
University of Milan
CINI Informatica e Scuola
Milan, Italy
lonati@di.unimi.it

Anna Morpurgo
University of Milan
CINI Informatica e Scuola
Milan, Italy
morpurgo@di.unimi.it

Mario Sanchez
Universidad de los Andes
Bogota, Colombia
mar-san1@uniandes.edu.co

***Abstract*—This Research Full Paper presents a study of programming strategies to manipulate data series presented in a range of CS1 courses. We collected and analyzed programs from multiple institutions in different countries, covering a range of programming languages (Python, Java, Go, and C). We started from a list of core strategies (that is, strategies that process the series as a whole, e.g., goals/plans for counting, linear search, etc.) drawn from the literature. We then expanded and refined the list, based on the analysis of the collected programs.**

We used a mixed method: we first analyzed the programs qualitatively to identify the recurring goals; then we conducted a quantitative analysis of the frequencies of goals. The result of the qualitative analysis is a data-driven list of basic goals that are required in order to cover the input, storage, and processing of data series, independently of the chosen programming language. The list has 13 well-separated goals (no goal in the list is a sub-goal of another one) that share a similar structure and the same level of abstraction. The quantitative analysis shows that the category of core goals is, not surprisingly, the most frequent one; around half of the programs contain a core goal, with total, count, and linear search being the most recurrent ones. Besides them, goals related to input and storage play a significant role as well: taken together, they occur with a frequency similar to that of core goals.

The list of goals, if used to select or design practical exercises and teaching materials, can be a valid reference for CS1 instructors who want to foster the acquisition of strategic knowledge by their students. Implication for teaching and learning are discussed.

***Index Terms*—CS1, programming strategic knowledge, goals and plans, iteration patterns.**

I. Introduction

Learning to program is a complex task, involving the acquisition of the syntax and semantics of core language constructs, in conjunction with building a mental model of the notional machine and developing algorithmic design and problem solving skills. Many introductory computer science (CS1) courses explicitly focus on how to structure and write syntactically correct code, that is on how to code a given algorithmic solution, while basic programming design skills (*strategic knowledge* as referred to in [1]) are often only implicitly taught. Multiple studies advocate for teaching strategic knowledge explicitly by introducing programming plans and plan composition in CS1 courses [2]–[4]. *Programming plans* describe templates for achieving simple common tasks or problems' sub-goals. For example, if the goal is to add all the elements of a list, a suitable plan typically indicates that a gatherer variable must be initialized, and this must be followed by a loop where the variable is incremented in each iteration with the contribution of the series' current item. Once students understand basic plans, they can focus on decomposing problems into sub-goals, identifying plans to implement each sub-goal, and then composing those plans, something which may be not trivial for novices [5], [6].

Programming plans are well supported by the literature [2], [7], [8] but there is a large variability in what is considered a plan and there is no consensus on which plans are required at CS1 level. Additionally, the line between considering a plan as an extension of another one or as a new one is not well-defined. In his thesis, deRaadt [9] studied the transition from a traditional teaching approach, where plans are implicit, to an approach based on the explicit usage of plans. One of his purposes was to "identify authentic expert programming strategies that are relevant to novice programmers". This resulted in a list of 28 plans of varying granularity.

Our study revisits this purpose, and aims at identifying a list of well separated and defined goals that are addressed in programs from CS1 courses. Unlike most of the related literature, which focuses on *plans* or *patterns*, we prefer to focus on *goals* whenever possible: in a given programming problem, goals and sub-goals are defined by the problem itself, whereas the plans to achieve such goals may present differences depending on the features of the selected programming languages. Moreover, plans present variants that depend on the style used (e.g., *while* vs *for-in*/*for-each* or using *break* [10]).

We limited the scope of our study to goals that are achieved by iterating over a series, either coming from a stream or stored in a linear structure, e.g. computing the sum of a series of numbers, or counting the number of items in a series that present a certain property. Note we use the term "series" for a sequence of homogeneous items that can be iterated on, regardless of how they are stored or fed into the program. We have two reasons for selecting this scope: (1) programs that process series of data represent an important part of CS1 course content [11]; (2) all the plans for the goals that fit the scope share a similar structure and level of abstraction, and this facilitates a uniform approach to teach them.

This study addresses two research questions:

RQ1 What are the recurrent goals to be achieved in CS1 programs that process one data series?

RQ2 How often do those recurrent goals occur in programs from CS1 courses?

The main contribution of this work is an evidence-based list of basic goals that (1) is short enough, so that it could be used as an effective reference list by CS1 instructors, (2) it covers all the typical problems that occur when processing series in CS1 programs, and (3) allows to uniquely describe each program that processes one series of data in terms of the goals in the list. Additionally, we provide an analysis of the occurrences of such goals and their combinations in CS1 courses' material.

II. Related Work

In his pioneer work [12], Soloway proposed to analyze problems in terms of typical recurrent *goals* that need addressing, and *plans*, i.e., stereotypical canned solution for such goals. He pointed out that the difficulties novice programmers encounter are not the language constructs themselves, but rather "putting the pieces together, composing and coordinating components of a program" [12, Page 850]. Thus, he suggested to *explicitly* teach stereotypical solutions and strategies for using and composing them. Moreover, he claimed that introducing goals and plans would provide the terminology to talk about strategic knowledge, still missing in the practice of programming teaching. This is the theoretical framework at the basis of our study.

Although this idea has been widely revisited in the literature on novice programmers, several issues are still pending: (1) there is no precise definition of what a plan is, (2) plans can take different forms in different programming languages, and (3) it's not obvious at what abstraction level plans should be identified and described.

Haberman and Muller described pattern-oriented instruction (POI) as "a pedagogical approach by which patterns are incorporated in a CS1 course in order to support the learning of algorithmic problem solving" [8, Page 2]. They explained that such patterns refer to goals (such as "Checking the existence of an item that satisfies a condition" [8, Page 2]) that occur in recurrent problems. Each pattern provides an *expert solution* that implements that given goal. Together with decomposition of a problem into sub-goals, patterns provide the building blocks to developing algorithms that solve new problems.

The list of 28 plans proposed by deRaadt to explicitly teach and assess strategic knowledge [2] covers a broad range of CS1 content with diverse granularity. For example, it includes single commands such as the divisibility plan (use of modulo operator) and the sub-algorithmic triangular swap plan, together with the count, sum and average plans, as well as a generic recursion plan and some algorithmic strategies such as bubble sorting. If we ignore the single statement plans and the algorithmic strategies, the list size is halved.

The list of *loop patterns* proposed by Astrachan and Wallingford [13] has a more restricted scope, and presents several useful patterns involving iteration. The list ranges over different levels of abstraction, including some patterns that describe generic methods to iterate through a series (e.g., Process All Items) and other patterns that are goal-oriented (e.g., linear search or best value).

Iyer and Zilles [14] provide the most recent attempt to identify the strategic knowledge needed at CS1 level by cataloguing different patterns in 12 different CS1 courses. They used a rather flexible definition of *pattern* as "any code block that is chunked by experts", ranging from complex boolean expressions and the swap pattern to ubiquitous plans such as counting or summing the elements of a set. They have both goal-oriented plans (e.g, sum, count, average) and generic patterns (e.g., processAllItem). Additionally, there are overlapping plans (e.g, findBestInCollection is a special case of processAllItems). The paper also analyzes how these patterns occur in teaching materials from CS1 courses, mostly focusing on when the patterns are introduced in the courses and on the commonalities between courses [14].

Textbooks usually focus on syntax and semantics of programming languages; a significant exception is Horstmann's textbook on Java [15]. The book aims at fostering strategic knowledge by explicitly presenting and naming a set of recurring goals related to iteration and presenting their typical implementations, however without explicitly talking of goals and plans.

III. Methods

This section describes the process used to extract the inventory of goals. We aimed at making a language-agnostic list but restricted ourselves to procedural programming languages.

A. Data collection

To capture the programming practice at CS1 level we started by collecting materials from our local courses' practice labs. This covered courses in three universities, from three continents, taught in three different languages (Python, Go, and C). At first we considered both programs written by instructors and (functionally correct) programs submitted by students. However, a preliminary analysis of students' programs showed that, even when they produced the expected results, they often contained convoluted code which needed to be disentangled before goals could be identified. Therefore, we decided to consider only exemplar solutions written by instructors.

We then complemented students' practice with examples that were shown in the courses' lectures and with programming tasks from final exams. Moreover, we extended the practice components by adding two datasets from *CodeLab* courses (https://www.turingscraft.com), one in Python and another one in Java.

We excluded all the programs that do not include loops and that do not process a data series. The resulting dataset has 348 programs; Table I summarizes its main features.

B. Qualitative Analysis of programs

We then analyzed the programs qualitatively, searching for the goals related to processing one series of data. The

TABLE I
DATASET CHARACTERISTICS

Dataset	Origin	Size	Language	Source
Go-Lect	U. of Milan	40	Go	Lectures
Go-Labs	U. of Milan	50	Go	Practice
C-Exams	U. of Milan	37	C	Exams
P1-Lect	U. de los Andes	34	Python	Lectures
P1-Labs	U. de los Andes	10	Python	Practice
P2-Labs	U. of Adelaide	22	Python	Practice
CL-python	CodeLab course	93	Python	Online
CL-java	CodeLab course	62	Java	Online

analysis used a mixed deductive-inductive coding process. In a preliminary step, we built a tentative initial table of typical goals, starting from the catalogues in the literature and our own experience as researchers and CS1 instructors (the authors represent over 50 years of collective experience in teaching programming-related courses and are active researchers in computing education). All authors independently analyzed a subset of the programs from P1-Lect and coded them using this initial table of goals (deductive component). We then refined the coding system in two rounds, working inductively on the basis of the data set.

- First round: The results of the first round of analysis were vigorously discussed among all authors and the table of goals was revised accordingly.
- Second round: the remaining sets of programs were coded independently by two researchers each. The results were discussed between the pairs, the coding were reconciled whenever possible; difficult cases were discussed among all researchers, which led to a final revision of the table of goals and coding of programs.

The results of this qualitative analysis are twofold. First, there is a table of recurring goals/plans types which will be described in Section IV. Secondly, there is a coding that associates each program with the set of goals that it addresses. As an example, Listing 1 shows one of the analyzed programs, which is coded as {**inputUntil**, **total** with filter}. In other words, this program achieves two goals: (1) reading some data until a condition is met, and (2) computing the total (the sum) of the read values, filtered according to some condition.

Listing 1. One of the coded Python programs

```
sum = 0
x = int(input())
while x>0 :
  if  x%2 == 0 :
    sum += x
  x = int(input())
print(sum)
```

C. Quantitative Analysis of programs

After coding all programs, the data related to the occurring goals/plans were quantitatively analyzed to address RQ2. We focused in particular on the frequency of goals. The results of this analysis are presented in Section V.

IV. INVENTORY OF RECURRENT GOALS

Table II summarizes the list of basic goals that resulted from our coding process. Each goal in the list actually represents a class of similar tasks; for instance, the **findBest** goal may occur as "find the shortest word" or "find the greatest number". At first, we considered only the list of *core* goals (or, more precisely, a preliminary version of that list). During the inductive part of the analysis we decided to add also other categories of goals, namely those referring to source, storage and the generic **repeatAction**. Most of them are mentioned in some form in the literature, as reported in the last column of the table.

A. Repeating actions

Often a series is processed by executing some specific action separately on every single item of the series; the effect of the action on one item has no impact on the effect of the action on the other (previous or next) items. We refer to this goal as **repeatAction**.

Examples of actions for this **repeatAction** goal are: printing all items, modifying them in place (e.g. doubling them), or specific actions defined by the problem at hand. To implement the **repeatAction** goal, there suffices a loop that iterates through the series, with the body of the loop describing the specific action on the current item. This plan is described by [15] as one of the typical recurrent goals that require iteration. One may argue that this is too simple to be even called a plan. However, writing loops where the body is parametric is already something beginners struggle with. Moreover, being aware of this plan is useful when dealing with more complex problems that require plan composition (e.g., print the number of vowels of each string in a list of strings, which requires nesting a **count** plan into a **repeatAction** plan).

Note this goal does not cover special actions as reading items from input, or storing them in a new structure, which define their own specific goals, described later.

B. Core goals

In contrast to **repeatAction** which processes each series item independently, *core* goals process the series as a whole to produce a single result.

The **total** goal refers to computing a global numeric value, as the product, the sum or other cumulative operation of the series elements. The corresponding plan uses a *gatherer* variable that is initialized before the loop with a suitable value and updated at each loop's iteration with the contribution of the series' current item.

The **count** goal refers to counting how many items in the series have some property (or count them all, resp.). The corresponding plan uses a loop iterating over the series, possibly with an `if` statement that tests the property on each item of the series, together with a counter variable that is initialized at 0 before the loop and is incremented at the iterations when the condition is met (or at each iteration, resp.). The **count** and **total** goals are already mentioned in [12].

TABLE II
LIST OF BASIC GOALS THAT OCCUR WHEN PROCESSING A SERIES

	Goal	Description	Result type	Mentioned in
	repeatAction	Repeat an action on each item of the series	n/a	[15]
CORE	**total**	Compute a global value (e.g. sum/product)	number	[2], [5], [12], [14]–[16]
CORE	**count**	Count the number of items	int	[5], [12], [14]–[16]
CORE	**multiCount**	Count multiple kinds of items using a single data structure	array/list/set	[2]
CORE	**findBest**	Find the best item in the series, or its position	item	[2], [5], [8], [14]–[16]
CORE	**findFirst**	Find the first item in the series that satisfies a given condition	item	[2], [5], [13]–[15]
CORE	**contains**	Check if the series has one item that satisfies a condition (breaks when found and returns true)	boolean	[8]
CORE	**validate**	Check if all items satisfy some condition (break when it fails and returns false)	boolean	[12]
SOURCE	**polling**	Read a series of items until a valid value is entered	item	[12], [13], [15]
SOURCE	**inputUntil**	Read a series of items until a condition is met	series	
SOURCE	**inputFixedN**	Read a fixed number of items	series	
SOURCE	**generate**	Generate a new series (eg. random numbers, token extraction)	series	
	storeNew	Store a series into a new structure	array/list/...	[15]

The **multiCount** goal provides an alternative and efficient way to implement multiple instances of the **count** goal. This occurs for instance when one wants to count the occurrences of each alphabet letter in a sentence. The plan for **multiCount** refactors the **count** plan with multiple counters by collapsing them into a data structure (array, map, or set) so that no filter statement is need for each counter increment. This is similar to the *tallying* plan described in [2]. This plan should be introduced as an optimization once the students have seen the suitable data structures (array, map, set) and encounter a problem with multiple counters. An example of program that use this plan is reported in Listing 2.

The **findBest** goal requires to select the best item (or its position) according to a specific criterion, e.g., the maximum number, the minimum number or the longest word. The plan for **findBest** uses a variable to keep the best value found so far. Such variable is suitably initialized (possibly to the first item of the series), then each item in the series is checked against it through a loop. Whenever a better value is found, the variable is updated in order to store it.

The **findFirst** goal also requires to select an item (or its position); differently from **findBest**, **findFirst** selects the first item that satisfies a given condition. The plan for **findFirst** iterates through the series testing the property on the current item. As soon as the test succeeds, the loop is interrupted and the current item (or its position) is the result. This goal is referred to as *linear search* in [13]. The plan typically has either an `if` statement with a `break`, or uses the property as the terminating condition for the loop itself.

Both the **contains** and the **validate** goals require to *check* whether the series items satisfy a certain property or not. The **contains** goal checks if *there is an item* that satisfies the given property; the corresponding plan returns true if such an item is found, interrupting the loop as soon as this happens, and false otherwise. It is worth noticing that this goal is similar to **findFirst**, but here the result of the computation is a boolean value and not an item (or a position) of the series. The **validate** goal checks if *all elements* satisfy the given property; the corresponding plan returns false if an item is met that does not satisfy the property, interrupting the loop as soon as this happens, and true otherwise. Both plans are frequently implemented with a `break` statement. As a variant, one can use the property to be checked as the terminating condition for the loop itself.

It is worth noticing that the plans that implement the **validate** and **contains** actually coincide, since "all items in the series satisfy property P" is the negation of "the series contains an item that satisfies property not(P)". Thus, one plan can actually be obtained from the other by negating the condition that tests the property on items and inverting the value of the boolean variable. However, this duality does not look easy for students to grasp, when they have little familiarity with logic expression equalities. For this reason we treat them as separated goals.

C. Source goals

In many cases, the series to be processed is provided, for example by passing a list as an argument to a function or method. *Source* goals occur when one needs to create the series from a source, such as reading the series items from the input stream.

The **polling** goal consists of examining a series of candidate values until a valid value is read. The series of invalid values is discarded and only the last value is used in the remaining part of the program.

The **inputUntil** goal is used to read a series of items from the input stream, until some condition is met (the simplest case is reading until a *sentinel* value is read [2], [12], [13], [15]); the series of read items is processed by the program. This is usually implemented by a `while` but can also be implemented with an `if` and a `break` statement.

The **inputFixedN** goal is similar to the **inputUntil** goal, but it is used to read an item series of a given fixed length. This is usually implemented with a `for` or a `for-range`.

The **generate** goal is used to generate a new series of items to be processed. The generation may be done for example by generating random values, or by generating some special progression of items according to some rule (e.g., Fibonacci).

Another way to generate the series is by tokenizing a single entity into separate items, e.g., when extracting digits from an integer [2], or words from a line.

D. Storing the series

The **storeNew** goal occurs when a new series needs to be stored in a structure (e.g, array, list, vector, dictionary). The goal includes declaring/defining such data structure. Note that **storeNew** occurs also when one wants to create a (possibly modified or filtered) copy of an already existing structure, but it does not occur when the existing series is modified in place.

Sometimes storing is required by the problem at hand, e.g., when you read a sequence of characters and you want to re-print them but omitting the occurrences of the last one. In other programs this is not the case, and the series is processed *on the fly* i.e. keeping in memory only one item at a time.

Listing 2. A program with **inputUntil**, **count** with filter, **multiCount**, and **contains**

```
func main() {
  var str string
  var numDigits [10]int
  numString := 0
  for str != "stop" {
    fmt.Scan(&str)
      if countDigits(str, &numDigits) {
        numString++
      }
    }
  fmt.Println(numString, numDigits)
}
func countDigits(s string, numDigits *[10]int)
     (hasDigit bool) {
  var i int
  for _, c := range s {
    if unicode.IsDigit(c) {
       i = int(c - '0')
       (*numDigits)[i]++
       hasDigit = true
    }
  }
  return
}
```

E. Other strategic knowledge

Apart from the goals listed in the inventory, we now report on other interesting issues related to strategic knowledge that arise from the analysis.

1) Filtering: In some cases goals need to be adapted. A recurrent adaptation is *filtering*, that applies when one wants to consider only some items of the series (e.g., count the odd values); this is typically implemented by equipping the relevant plan with an `if` statement that checks the filter condition.

2) Other possible goals: During our analysis we also found programs that have specialized goals related to specific features: (i) processing 2D arrays (matrices); (ii) implementing specific algorithms on series, e.g. Euclid's algorithm for greater common divisor, binary search, bubble sort; (iii) processing two series (and not one), e.g. merge two series into a new one. We coded the programs that address such particular goals as **other**.

V. Quantitative results

This section presents the quantitative analysis of the coding, which aims at answering RQ2. From the 348 programs in our dataset, only 26 (7%) were coded as **other**. They won't be included in the quantitative analysis.

The 322 programs contain 543 goals, that is in average there are 1.68 goals per program. Figure 1 shows plan frequency for each goal. The most frequent plans are **repeatAction**, **storeNew**, **total**, and **inputUntil**, followed by **count** and **findBest**. The figure also shows the meaningful frequency of the variants for the goals that admit filtering: **total**, **count**, **findBest**, **repeatAction**, and **storeNew**.

In terms of categories, 38% are *core* goals, 26% are the generic **repeatAction** goal, 21% are *source* goals and 15% is the **storeNew** goal. These percentages do not reflect whether the goals are clustered in a few programs or spread evenly over them. The programs that contain core goals are the most frequent ones (51%), but also source and store goals are significantly present, with source goals occurring in 33% of programs, and data series being stored in 24% of them (note that these goals may be combined, hence the sum is greater that 100%). Only 19% of programs have no other goal than the action-repeating goal.

Figure 2 provides another perspective of plan frequency by showing the average number of goals (per category) for programs presented in each different course. There are clear differences between courses that could be attributed to different course designs and the programming languages they used. For example, P1-Labs dataset comprises a set of Python functions that receive data as parameters, hence they do not read from input; this explains the low frequency of source plans (only **generate**). This being not central for our research questions, we left the further analysis of such variations to future work.

Table 3 shows the number of sub-goals that are achieved by single programs. We can see that the number of sub-goals ranges from 1 to 5, with 52% of programs addressing one single goal and 32% addressing two sub-goals. Considering only core sub-goals, only 10% of programs address more than one core goal, and almost half of the programs don't address any.

VI. Discussion

The discussion is organized according to the research questions and is concluded by a brief discussion on this study's limitations.

RQ1 What are the recurrent goals to be achieved in CS1 programs that process one data series?

The inventory of goals presented in section IV provides the main answer to the first question. All the goals in Table II share a similar level of abstraction and are clearly separated, in that no one is a sub-goal or a special case of another one. Moreover the inventory identifies goals, instead of plans, being thus language agnostic. We believe these are desirable properties for such an inventory, that are not shared by the catalogs

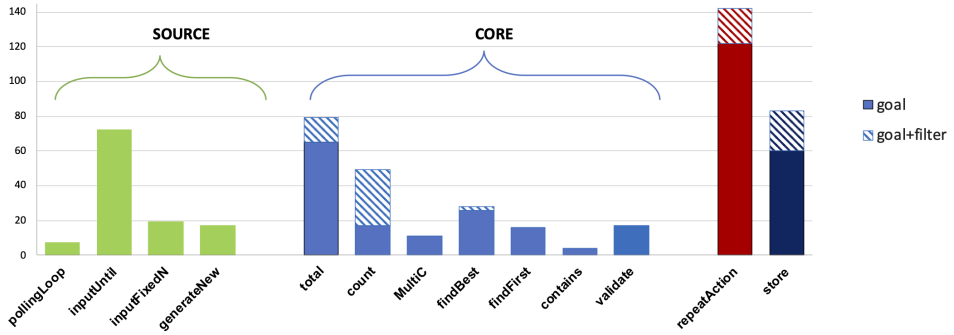

Fig. 1. Goals frequency with and without filter

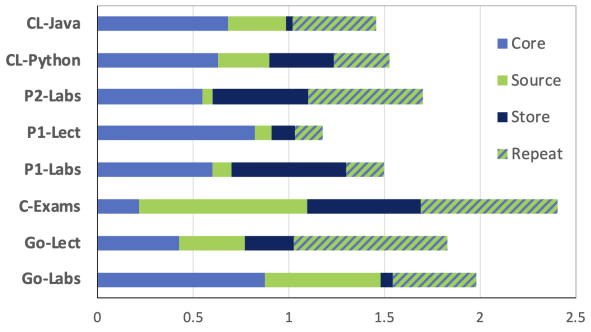

Fig. 2. Program's average goal usage for each dataset

Number of sub-goals	1	2	3	4	5
Program tally	167	102	41	10	2
Proportion	52%	32%	13%	3%	1%

Number of core sub-goals	0	1	2	3	4
Program tally	158	134	22	6	2

Fig. 3. Programs tallied by the number of addressed sub-goals or core sub-goals.

of plans from the literature. For instance, let us consider two significant catalogs: the list of *strategies* in de Raadt's thesis [9] and the *loop patterns* presented in [13]. de Raadt covers many topics in CS1, producing a diverse list that ranges over rather different levels of abstraction: from swapping variables and selection patterns to recursion, including some of the iterative strategies that we considered. Among the *loop patterns* presented in [13] (and recently reviewed in [14]) one can find the **findBest** and **findFirst** plans (under different names) together with generic plans such as "process all items", that overtly includes them both, and "loop and a half", to handle some read-and-process loops; in other terms, their list contains both plans to achieve specialized goals and generic patterns to iterate over a series.

Half of the goals in Table II are *core* goals, in that they process the series as a whole and their results can be seen as an attribute of the series. There are four basic core goals that have largely been discussed in the literature, possibly with different names (see the references cited in Table II): **total**, **count**, **findBest**, and **findFirst**. Note **multiCount** have lower frequency as it is introduced later in the courses as the refactoring of multiple instances of **count**. The remaining plans **contains** and **validate** share similar patterns to **findFirst** but have different objectives. Besides, **validate** can be seen as "does not **contain**". However, the inclusion of both is important as novices are not familiar with equivalences between logic expressions and might not recognize that the two goals are indeed dual. Once they have mastered both, we could explain the relation between them.

Filtering is a recurrent extension that restricts the goal to a subset (e.g., from "summing all items" to "summing all odd items"). A similar pattern is mentioned in [14], but as a pattern instead of an extension to an existing goal.

Besides core goals, in our table we listed as many other goals, which play an important and recurrent role when handling series of data, e.g., to read the series from an input, or store it in a structure (array, list, ...). Except for **inputUntil** and **polling**, they received much less attention in the literature.

During our analysis we also found programs that present particular features that make them different from any of the other goals: they are typically unique in each course (see Section IV-E2) and represent topics that are frequently covered towards the end of CS1 courses (e.g., matrices, merging two series). Even though they contain loops to process series, they present goals and plans that seem too specialized. For instance, summing up all columns in a matrix can be described as a **total** plan nested into a **repeatAction** plan, but we think that nested loops like these should be most appropriately seen (and taught) as plans *for processing matrices*. Similarly, our inventory does not cover some problems and algorithms that are typical of CS1 courses, but are not easy to generalize or reuse (e.g., Euclid's GCD, checking for palindromes). Thus, we coded such goals as **other**.

RQ2. How often do those recurrent goals occur in programs from CS1 courses?

When we drafted the initial list of goals for our analysis, based on the literature and our own experience as CS1 instructors, we focused on core goals. As expected, the most frequent core goals are **count**, **total**, and **findBest** which matches the most common plans in the literature.

However, in the inductive phase of our qualitative analysis, we decided to extend the inventory to include source and store goals, which have a fundamental role as well. As a matter of fact, it turns out that the frequency of source and store goals together is higher than the frequency of core plans. In particular, **storeNew** and **inputUntil** occur with high frequency, while other source roles are similar in frequency to the less common core goals. That is, besides asking how to process the data in the series, one needs to focus also on other two fundamental questions: where the series comes from (source goals) and whether it is necessary to store the data.

The quantitative analysis of the collected programs also gives us a measure of the complexity of programs that manipulate a data series in terms of number of sub-goals. Almost half of the programs address more than one goal, which need to be combined, and this increases the difficulty in designing and writing the program (see [5], [6]).

Limitations

The first limitation of this work has to do with how representative our dataset is, as we are aware that we might have not captured the task variety that can be found on CS1 courses. However, the process that we used to discover and compile the table of goals can be used to analyze other programs coming from other institutions. Moreover, the focus on goals instead of plans means that the goal inventory is applicable to other programming languages and even other programming paradigms.

Another limitation concerns the fact that most programs come from courses at the authors' institutions. However, it's worth mentioning that such institutions are in three different continents, and in the analysis none of the authors was assigned programs from their own courses. Moreover, part of the analyzed programs come from courses unrelated to the authors, as for those taken from CodeLab.

VII. Implication for teaching and learning

Schema acquisition. Goals and plans are not generally taught in an explicit way; students are simply exposed to examples where these recurrent goals occur, and are expected to recognize them and build mental schemata from those examples. On the contrary, it is argued that schema acquisition must be supported by explicit instructions and terminology [12], [17]–[19]. Additionally, goal identification plays a key role in program comprehension tasks such as "Explain In Plain English" [20] and directly supports schema acquisition. We believe that our inventory can offer CS1 instructors an effective reference to design and select teaching materials and practical exercises that support students in the acquisition of these schemata and in general in the acquisition of strategic knowledge.

Core goals. Using a loop to repeat an action on every item of a series (i.e., the **repeatAction** goal) is something that is generally explicitly taught. However, this does not provide much strategic knowledge, since the plan is very generic, and the action to be performed must be devised depending on the specific requirements of the program. On the contrary, *core* goals are quite specific while being applicable in many different situations. We found that the number of distinct core goals to process a data series is low. This means that it is possible to name and address them all during CS1 classes. They can be presented by using examples and making explicit the similarities the examples share [18]. For instance, for each core goal, the instructor can present the goal and its plan by considering different problems that share the same goal, discussing how such problems are similar, presenting a program for each problem, and illustrating the similarities in the programs' structure, thus highlighting the corresponding plan.

Not only core goals. Covering both core goals and other goals (source, store, repeat actions) is important in teaching because they are all needed when handling series of data. In particular, non-core goals typically have to be merged with core goals, which is knowingly a source of struggle for students [5], [12]. Being aware and having a clear understanding of the properties and differences of these goals is a necessary first step in learning to compose them together. However, when introducing core goals, in order to make the schema more visible, it is a good idea to reduce at first the context differences, i.e., to keep the source and store goals unaltered.

From goals to plan. From a practical point of view, when presenting goals/plans to students, focusing on the goal is the most important thing. However, the goal is typically presented through a solution implemented in a particular programming language. This makes it difficult to separate what the program is actually trying to achieve (the goal) from the coding decisions linked to the language. As a result, students learn that particular solution only, and most of them have a hard time abstracting the solution to apply it in a similar or a larger problem. For example, they may be able to find the maximum number in a series and to count the occurrences of a number in the same series, but they find it difficult to count the occurrences of the maximum number without doing two passes over the series. In this regard, the inventory gives each goal a name and makes it easier to discuss about them, independently of the implementation.

Designing programs. Teaching goals and plans explicitly provides the language to discuss design issues and suggests a concrete starting point to design programs from requirement specifications. However, it is important for educators to explain that they are not set in stone and can be modified to fit the program requirements. In fact, we found that the standard traversal of the series (from the first to the last item) is customized in 38 occasions to fit the problem at hand: by iterating *backwards* from the last item to the first one (e.g., to find the last occurrence of an item); by increasing the loop index by a particular value, thus *skipping* some elements in the series, to refactor a filtering by index position; or by iterating only *until* some condition is met (e.g., to sum up until a threshold is reached).

VIII. Conclusions and Future Work

This paper presents an inventory of 13 goals obtained after classifying 322 programs in Python, Java, Go and C, used in introductory programming courses in at least 3 different countries. This inventory differs from alternative proposals in that it focuses only on programs that process series, and leaves aside both simpler and much more complex goals. Our inventory is composed of goals that are well separated and operate at the same level of abstraction. More importantly, the inventory identifies goals (the objectives of programs), instead of plans (the implementation of strategies to achieve those objectives); this distinction is critical and makes our proposal agnostic with respect of programming languages.

Although the number of courses and programs studied is just a sample, the inventory represents an important step towards understanding how strategic knowledge can be explicitly taught to novices: the identification of a clear and complete set of goals is a solid starting point for instructors to build instructional material that fosters strategic knowledge related to data series processing.

The goals in the inventory are classified in 4 groups that cover over 93% of the exercises in our datasets. Among the remaining 7%, we saw problems and algorithms that are typical of introductory courses but cannot be generalized, or programs that deal with more than one series –or series of series (e.g., matrices)– and hence require specialized goals and plans.

Half of the analysed programs contain 2 or more goals. Therefore, students need to be taught not just the list of goals, but also composition strategies to combine the related plans. The analysed set of CS1 programs provide a rich set of composition patterns that will be analyzed in future work.

References

[1] T. J. McGill and S. E. Volet, "A conceptual framework for analyzing students' knowledge of programming," *Journal of Research on Computing in Education*, vol. 29, no. 3, pp. 276–297, 1997.

[2] M. de Raadt, R. Watson, and M. Toleman, "Teaching and assessing programming strategies explicitly," in *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95*, ser. ACE '09. AUS: Australian Computer Society, Inc., 2009, p. 45–54.

[3] M. Hu, M. Winikoff, and S. Cranefield, "Teaching novice programming using goals and plans in a visual notation," in *Proceedings of the Fourteenth Australasian Computing Education Conference - Volume 123*. AUS: Australian Computer Society, Inc., 2012, pp. 43–52.

[4] K. Fisler, S. Krishnamurthi, and J. Siegmund, "Modernizing plan-composition studies," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 211–216. [Online]. Available: https://doi.org/10.1145/2839509.2844556

[5] D. Ginat, E. Menashe, and A. Taya, "Novice difficulties with interleaved pattern composition," in *Informatics in Schools. Sustainable Informatics Education for Pupils of all Ages*, I. Diethelm and R. T. Mittermeir, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 57–67.

[6] A. Ebrahimi, "Novice programmer errors: language constructs and plan composition," *International Journal of Human-Computer Studies*, vol. 41, no. 4, pp. 457–480, 1994. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S107158198471069X

[7] F. E. V. Castro, S. Krishnamurthi, and K. Fisler, "The impact of a single lecture on program plans in first-year cs," in *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 118–122. [Online]. Available: https://doi.org/10.1145/3141880.3141897

[8] B. Haberman and O. Muller, "Teaching abstraction to novices: Pattern-based and adt-based problem-solving processes," in *2008 38th Annual Frontiers in Education Conference*, 2008, pp. F1C–7–F1C–12.

[9] M. deRaadt, "Teaching programming strategies explicitly to novice programmers," Ph.D. dissertation, USQ eprints, 2008. [Online]. Available: https://eprints.usq.edu.au/4827/

[10] J. Sorva and A. Vihavainen, "Break statement considered," *ACM Inroads*, vol. 7, no. 3, p. 36–41, Aug. 2016.

[11] Y. Cherenkova, D. Zingaro, and A. Petersen, "Identifying challenging cs1 concepts in a large problem dataset," in *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM, 2014, pp. 695–700.

[12] E. Soloway, "Learning to Program = Learning to Construct Mechanisms and Explanations," *Communication of the ACM*, vol. 29, no. 9, pp. 850–858, 1986.

[13] O. Astrachan, G. Mitchener, G. Berry, and L. Cox, "Design patterns: An essential component of cs curricula," *SIGCSE Bull.*, vol. 30, no. 1, pp. 153–160, Mar. 1998. [Online]. Available: http://doi.acm.org/10.1145/274790.273182

[14] V. Iyer and C. Zilles, "Pattern census: A characterization of pattern usage in early programming courses," in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 45–51.

[15] C. S. Horstmann, *Java Concepts: Early Objects*. New York: John Wiley & Sons, Inc., 2015.

[16] M. Lopez, J. Whalley, P. Robbins, and R. Lister, "Relationships between reading, tracing and writing skills in introductory programming," in *Proceedings of the Fourth International Workshop on Computing Education Research*, ser. ICER '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 101–112. [Online]. Available: https://doi.org/10.1145/1404520.1404531

[17] J. J. Van Merrienboer and F. G. Paas, "Automation and schema acquisition in learning elementary computer programming: Implications for the design of practice," *Computers in human behavior*, vol. 6, no. 3, pp. 273–289, 1990.

[18] D. N. Perkins, G. Salomon *et al.*, "Transfer of learning," *International encyclopedia of education*, vol. 2, pp. 6452–6457, 1992.

[19] G. Lewandowski, A. Gutschow, R. McCartney, K. Sanders, and D. Shinners-Kennedy, "What novice programmers don't know," in *Proceedings of the First International Workshop on Computing Education Research*, ser. ICER '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 1–12. [Online]. Available: https://doi.org/10.1145/1089786.1089787

[20] C. Izu, C. Schulte, A. Aggarwal, Q. I. Cutts, R. Duran, M. Gutica, B. Heinemann, E. T. Kraemer, V. Lonati, C. Mirolo, and R. Weeda, "Fostering program comprehension in novice programmers - learning activities and learning trajectories," in *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education, ITiCSE 2019, Aberdeen, Scotland Uk, July 15-17, 2019*, B. Scharlau, R. McDermott, A. Pears, and M. Sabin, Eds. ACM, 2019, pp. 27–52. [Online]. Available: https://doi.org/10.1145/3344429.3372501