A Cognitive-Based Approach for Teaching Programming to Computer Science and Engineering Students

Covington, R. and Benegas, L.
California State University Northridge, Northridge, CA, 91330

1. Introduction

An issue receiving attention in the undergraduate Computer Science curriculum over the past few years has been the high failure rate in the freshman programming course. This course generally corresponds to the ACM/IEEE course designation CS1. It is normally an introductory but fast-paced and challenging course for students who have not previously studied computer programming (programming novices), but who do have a minimum level of mathematical maturity (students who are calculus-ready). The course attracts an audience composed of majors from Computer Science, Information Technology, and Computer Engineering, for whom it is a requirement for their major, as well as students from other science and engineering departments. Failure rates of 15% to 30% are not unusual [10], and the problem is widespread, from top-tier private schools, through the state universities, all the way to the community and junior colleges. There are many possible causes, and some can be blamed on the students themselves (poor advisement, poor math preparation at the high school level, among others). But other causes must be contributors. While computer programming might be a more technically challenging skill to master than, say, writing a good English essay, it seems odd that it should suffer a higher failure rate than other challenging freshman-level courses in calculus, physics, or engineering.

Many educators have begun to assign the blame on the teaching approach. In this paper we critique some current teaching approaches and agree that this is one source of the problem. A glance at almost any textbook on introductory programming will reveal a presentation that starts from many flawed assumptions about the target audience, and that does not follow well-established principles for how to teach technical material. Computer programming education simply is not as mature as the teaching of the sciences and engineering, and this is reflected in the CS1 failure rate. In this paper we explore some promising approaches from well-established research in cognitive psychology, which has produced results on how students learn, and suggest some new ways to apply these results to computer programming instruction. We focus on two CS1 topics in particular – iterations (loops) and decisions (if-else) – and suggest ways to organize the presentation of these topics. The foundation of the approach is to use schemas (mental patterns) for teaching both how to solve problems in the abstract and how to convert those solutions into computer programs.

The organization of the paper is as follows. First, we review the relevant results from cognitive psychology for how students learn in general, as well as results specific to computer programming learning. Next we offer a short critique of the approach used by many current textbooks that we call **syntax-driven**. This approach places undue early emphasis on language syntax (i.e., “where to put the semicolons”) and not enough emphasis on understanding and solving problems. Next we give an overview of an alternative approach we call **schema-driven**, which is based on the results of the cognitive psychology research discussed earlier. This approach allocates more resources early on to the teaching of problem solving and solution approaches, saving discussion of language syntax specifics until much later. In Section 4, we offer some examples of how to apply this approach to the teaching of iterations and decisions. Finally, in Section 5, some directions for future work are outlined, including plans for evaluating the effectiveness of the approach with an assessment of student progress for CS1 students.

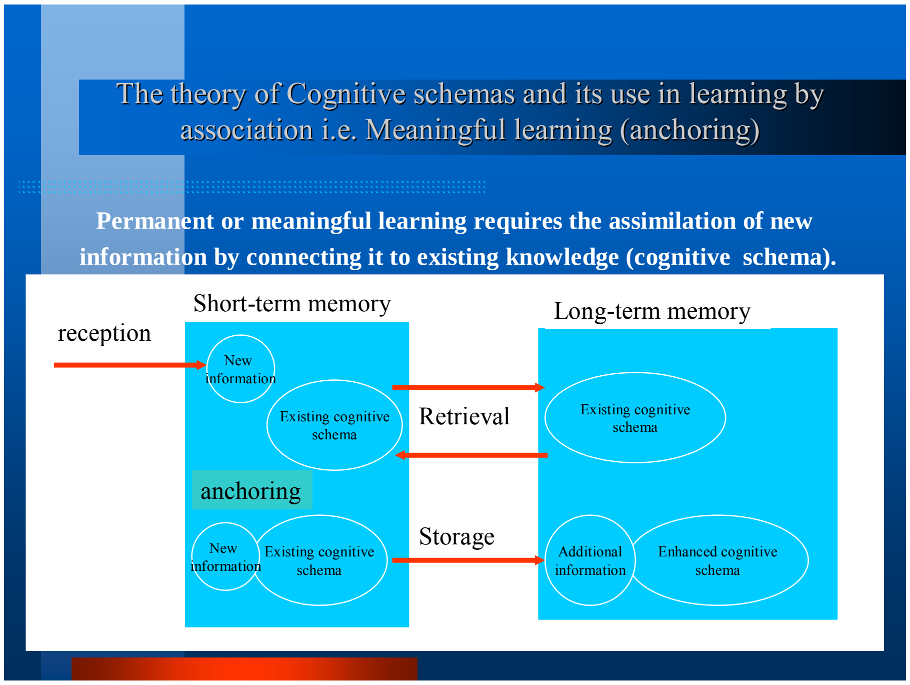
2. Background

We claim that the approach currently used to teach CS1 suffers from a lack of attention to well-known results from cognitive psychology on how students learn. In this section, we review some of the relevant models of learning in general and then focus on research that addresses learning computer programming in particular. Many of the general models depend on a central concept called a **schema**, which is a mental pattern or model of conceptual or abstract information. Based on these results, we present an approach we call **schema-driven,** which we contrast to the more traditional teaching approach we call **syntax-driven.**

Meaningful Learning (Schema-Based) vs. Rote Learning (Short-Term Memorization)

The cognitive learning process uses both short- and long-term memory, which differ in their capacity for holding and manipulating information: a temporary limited store for short-term memory and a more permanent and better-organized store for long-term memory. A model of how memory works called **cognitive load theory** claims that the short-term memory is limited to holding only a few items at a time, thus posing a fundamental constraint on human performance and learning capacity (sometimes called the **7 plus or minus 2** rule). The limitations of the working memory are compensated by capabilities of long-term memory. The most important of these is **schema acquisition**, which allows the mind to group information into meaningful units that are easy to store in long-term memory and which can be easily retrieved as needed to handle information processing and understanding [6]. These units are also called **chunks** in the literature. This result means that the teaching of any technical topic, including computer programming, depends on the student developing useful high-level abstractions. Approaches that depend on memorizing long lists of facts are doomed to failure because of the limitations of short-term memory.

According to Mayer [11], **meaningful learning**, occurs when the student assimilates new information by **connecting** it to existing knowledge, which is represented by a schema already in long-term memory. The learner first focuses on new information as it enters short-term memory (called **reception**). She then retrieves some relevant prerequisite concepts (called the assimilative context) from memory by a process of pattern matching. The new material is then connected to a pre-existing schema, both of which are now in short-term memory, resulting in an enhanced schema. The final step is to store the enhanced schema back into long-term memory. This attaching or **anchoring** of new information onto an existing schema is what differentiates **meaningful learning** from **rote learning**. Rote learning or memorization occurs when the learner simply stores the new information as a separate schema (not linked to other similar schema) in long term memory (see Figure 1).


The theory of Cognitive schemas and its use in learning by association i.e. Meaningful learning (anchoring)

Permanent or meaningful learning requires the assimilation of new information by connecting it to existing knowledge (cognitive schema).

Figure 1. Illustration of the process of meaningful learning

An example of the importance of meaningful learning for novice programmers is the use of the equals sign "=". The novice programmer begins the study of programming with an already established understanding of this symbol's algebraic meaning, but the symbol has very different meanings in the algebraic and programming domains. Instructors must avoid the assumption that the student can simply use inapplicable existing schemas from algebra as an anchoring schema for new programming concepts. At the same time, **cognitive overload** must be avoided by introducing concepts one at a time in a sequence of gradually increasing complexity.

Meaningful Learning Applied to Computer Programming

Results from this research that apply to learning programming show that the cognitive skills used in understanding problems and finding a solution differ from those required for composing a computer algorithm, and these in turn differ from those used in learning to use a programming language. The process of learning a programming language for the purpose of implementing a computer solution to an assigned problem involves:

- Ability to recognize a pattern that describes a solution
- Recognition and memorization of syntactical units, groups of statements or commands, and
- Grouping of syntactical units into larger conceptual entities or semantic units that serve as a procedure to implement the problem solution pattern.

As the novice programmer evolves into the expert, she begins to master not just the meaning of individual program statements but also the program schemas that encode series of statements into higher-level concepts. Acquiring syntactic knowledge involves a form of rote memorization, which can be mastered by familiarity through repetition (drilling), and error recognition. Semantic knowledge on the other hand involves a pattern recognition activity that relies on stored schemas as opposed to rote memorization (Figure 2). Once again, these results emphasize the importance of providing programming students with teaching support that fosters the creation of the appropriate high-level abstractions.

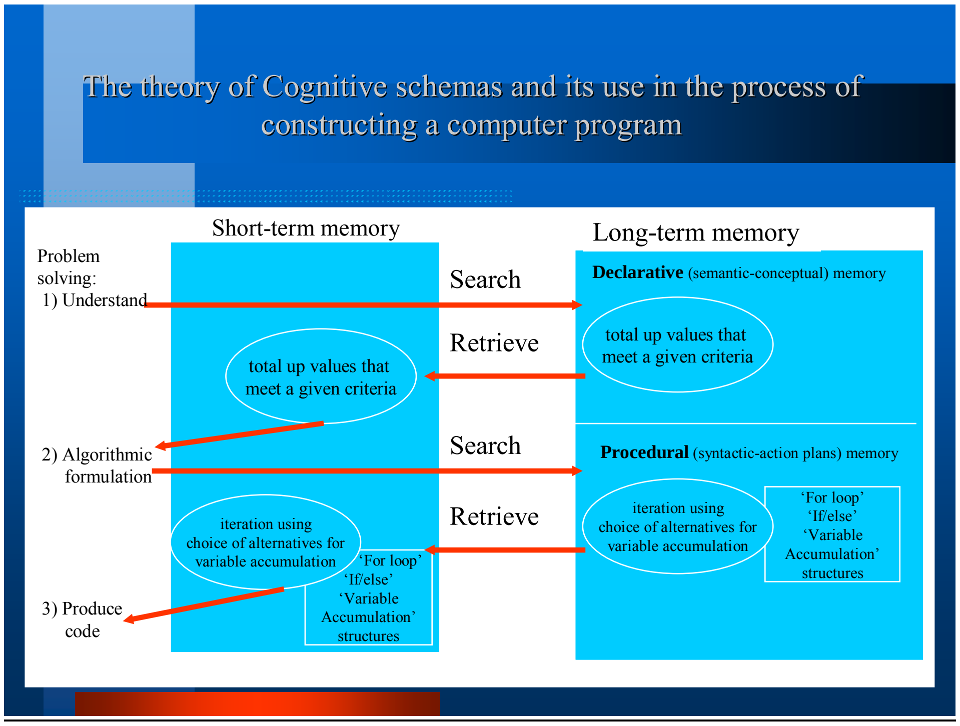

The theory of Cognitive schemas and its use in the process of constructing a computer program
Short-term memory
Long-term memory
Problem solving:
1) Understand
Search
Declarative (semantic-conceptual) memory
total up values that meet a given criteria
Retrieve
total up values that meet a given criteria
2) Algorithmic formulation
Search
Procedural (syntactic-action plans) memory
iteration using choice of alternatives for variable accumulation
'For loop' 'If/else' 'Variable Accumulation' structures
Retrieve
iteration using choice of alternatives for variable accumulation
'For loop' 'If/else' 'Variable Accumulation' structures
3) Produce code

Figure 2. Cognitive steps involved in programming

The Acquisition of Plans and the Role of Chunking

Mattson [14] poses the question "how does a programmer move from model to code?" He explains that this transition occurs in terms of **chunks** of operations called **plans** — code fragments that solve a particular problem according to some standard pattern. Plans have been recognized as a basic element in programming knowledge representation for many years. Mattson traces these ideas to Soloway et al. [3,4,5]. Some authors use the term **programming idioms** to refer to common groupings of low-level computer statements that frequently appear as a unit. According to the **plan theory of programming**, the ability to group low-level statements into chunks is precisely the skill that distinguishes experienced programmers from novices when performing the algorithm-to-code translation. As the programmer encounters new problems, new plans are built formed, one statement at a time, beginning with some key or fundamental statement [13].

The process of trying to make new mental models correspond to existing ones is known as the theory of **cognitive fit**, first researched by Soloway, et al. [2]. Existing experimental work using looping constructs led researchers to hypothesize that programmers "…will find it easier to program correctly when the [programming] language facilitates their preferred cognitive strategy".

Transfer of Existing Problem Solving Skills to New Contexts

Another cognitive aspect of learning that directly applies to programming involves the **skills transfer process**, which occurs after the student has combined new information with knowledge that is already in long-term memory. Further research on how novice students become experts shows that for novices, meaningful learning and understanding are too dependent on the learning context. Thus they are able to apply what they have learned only when the new problem is very similar to the learning situation. The result is a low transfer of creative skills. This demonstrates the need for **drilling** the student on a variety of similar problems to help them overcome this dependence.

Successful Learning Based on Schema Acquisition

Based on these results, the composition of the ideal course teaching programming should assume that skill acquisition takes place through separate threads of presentation for each computer programming skill – (1) learning to analyze and solve problems, (2) composing algorithms in a language-neutral notation, and (3) translating algorithms into code – and all the threads proceed in parallel. In practice, the third thread is often emphasized to the exclusion of the others. Because of the limitations of short-term memory, the early stages must be dominated by presentation of high-level abstract concepts that the student can successfully convert into a schema. Memorization-heavy topics such as syntax, which run the risk of subjecting the student to cognitive overload, must be pushed later in the process, when the student will have already developed the schemas to help him absorb this level of detail. In keeping with the concepts of chunking, gradient, and drilling, the design of course material should consider at all times how to best structure the organization, sequence and correlation of the concepts to be presented [7,8,9].

To reiterate, current methodologies for teaching programming skills depend heavily on what we call a **syntax-driven** approach, which place less emphasis on learning to solve problem and implement solutions through the use of a programming language, and instead place primary emphasis on learning the syntax of the programming language features. The proposed alternative we call **schema-driven** directs the student to spend relatively more time thinking about problem characteristics and problem solving solution patterns early on, and only later directs the student to focus on the details of language syntax. In the next section, we give more detail regarding the two approaches, plus some guidelines on how to organize teaching materials in a schema-friendly way.

3. Detailed Comparison of Syntax-driven and Schema-driven Approaches

For students, learning how to program is too intimately connected to learning the details of a specific language. Today, for the language of choice on most campuses for CS1 is Java. Since its formal announcement by Sun Microsystems at SunWorld ’95, Java’s popularity has soared, and it has quickly become one of the most widely adopted teaching languages. The popularity is such that Java has acquired the status of high-tech buzzword. Novice programming students come to the field with a desire to learn the details of a fashionable language that is stronger than a desire to learn how to solve problems. This creates a pressure that the educator must consciously resist. Another negative result of the popularity has been an explosion of textbooks and less formal how-to books ("Java in x hours", "Java for non-geniuses", etc.). Not all are poor, but many offer up teaching presentations that adhere to the syntax-driven pattern. Such texts consider the teaching of language syntax as their primary educational mission, to the exclusion of any broader range of topics – development of mathematical insight into the problems presented as examples in the text, presentation of problem solving strategies, and formulation of algorithms independent of any specific language.

The alternative approach advocated here we call **schema-driven**, and it is based on the results presented in the previous section, which emphasizes

- Presenting problem types, solution strategies, and related program language elements in integrated units.
- Teaching students to recognize problem and solution patterns (schemas) before attempting to translate those solutions into code
- Introducing the syntax of specific programming language features in the context of the problems they are meant to solve.

The approach is heavily top-down. Each unit initially emphasizes the study of problems and solution approaches at a high level, followed by discussion of algorithms and solution design. Discussion of specific programming language constructs and syntax is placed at the end of each unit.

3.1 Examples and Critique of Syntax-Driven Approach

The following is an example outline of topics from a popular textbook on Java, taken from the chapter on iteration. The text states that the goal of the chapter is to learn how to write programs that repeatedly execute one or more statements. It is typical example of a syntax-heavy approach that fails to develop insight into the computational nature of a problem solution before jumping into a detailed coded solution.

The first table in the section illustrates compound interest by showing how the balance on a CD account (initial balance of $10,000) grows over time, assuming some annual interest rate (5% in this example), compounded annually (see Table 1):

Table 1. Compound Interest Calculation Example

Year	Balance
0	$10,000
1	$10,500
2	$11,025
…	…

The question posed to the student is to determine how long it will take for the balance to reach $20,000. Next, the syntax of the **while** loop is introduced as the Java statement that is designed to compute the answer to such questions. The generic version of the loop is shown first:

```
    while (condition) { body }
```

This is followed by a version that has been customized to answer the question:

```
    while (balance < limit) {
          yrs++;
          double interest = balance * rate;
          balance = balance + interest;
    }
```

At the end of this loop, the answer is given by the value of the variable **yrs.** Next, a complete Java program is provided that more formally solves the original problem. In brief, the remainder of the chapter's presentation covers:

- Flow charts as a visualization aid to understand flow of control.
- Common syntax errors to avoid: infinite loop, off-by-one, extra and missing semicolons.
- Loop invariants.

Critique of the Syntax-Driven Approach

The presentation starts off well by illustrating the computation in a tabular form that is intuitive and that does not refer to the details of how to calculate it. Ideally, this table should be followed by a discussion of the repetitive nature of the computation required to reach the answer, while still avoiding a commitment to programming-specific notation, and how the repetitive approach is common to the solution of many similar problems. Unfortunately, the presentation instead moves immediately to the syntax of the Java while loop. In rapid succession come presentations of (1) a generic while loop, (2) a partially completed one, and (3) a two-page source code listing for the full program solution.

By devoting most of the page count to the details of while loop syntax, this approach implies that the primary skill for the student to master is syntax. In fact, students at this level have not yet learned how to analyze the problem statement itself to understand what the solution should even look like, and they have no body of problem solving techniques to fall back on. Therefore this approach is bound to fail. The true programming novice first needs to be taught (1) how to analyze the problem and choose a general solution approach and (2) how to formulate a computer algorithm as a more precise statement of the solution approach. Only after these topics are addressed should they study how to translate the algorithm into the final Java program. Our recommendation is that the text should expand upon its excellent initial example and develop examples of other iteration-oriented problems and present them in tabular form. Students need to see several examples that compute different values, yet arrive at their respective answers according to the rules of a common pattern or schema. The fatal flaw of the approach used by many programming texts is this: they assume that the student has already acquired these abstractions. Unfortunately, our experience in teaching these concepts over the last five years has convinced us that novice programmers, even mathematically advanced ones, **do not** already possess such mental abstractions, but instead must be **taught** them, and it is the responsibility of CS1 to do so.

3.2 Overview of Schema-Driven Approach

In this approach, we first teach the novice programmer that solving a programming problem begins with recognizing when a problem belongs to some already known problem class or "pattern". A pattern can be viewed as a generalized way of solving a particular class of problems. Pattern recognition — the ability to search for and identify existing solutions that worked for similar problems in the past — is at the center of the learning process. Typically, a pattern becomes more general through reuse. A general pattern is one that solves an entire class of similar problems. This usually only appears after successful application of a certain pattern a number of times, and then seeing a common pattern throughout these applications. Second, we teach the novice to recognize that corresponding to problem patterns, there are programming construct patterns. Finally, we show the student how to find a match between the problem solution pattern and the programming language pattern, from which the computer program solution may be easily generated. This approach is likely to filter out many pointless and uninteresting variations or mutations of the programming language constructs that become a distraction in a syntax-driven approach and fail to hook up with the problem solution patterns. See Table 2. for a contrast between the syntax-driven and the schema-driven approaches.

Table 2. Overview of Syntax-Driven and Schema-Driven Approaches

SYNTAX-DRIVEN APPROACH	SCHEMA-DRIVEN APPROACH
Syntax of language constructs is presented without a firm connection to the problems they solve	Problem description/solution for a group of related problems (problem type) that exhibit the same solution pattern are introduced in a gradual manner
Examples are presented and exercises are assigned that use the various syntactical variations (e.g., BNF)	Language statements and their syntax are presented with emphasis on groups of statements that perform a useful action
A sample problem is presented that uses the syntax	Useful and applicable syntactic models (programming patterns) are presented that directly relate to implementing the solution of the type of problem presented.
Programming problems that "fit" (make use of the syntax) are then assigned	Exercises are assigned that drill on chunks of code (programming patterns) followed by programming problems that will require the use of these programming patterns.

Programming schemas (language structure patterns) can be categorized as being behavioral in the sense that they represent the handling of particular types of actions, a procedure. They encapsulate processes that you want to perform, such as moving through a sequence (as in an iteration), or implementing a decision making step or repeatedly performing a set of actions. The nature of the different programming languages change the expression and understanding of these patterns. The programming patterns themselves use basic principles of organization. These principles are based on and reflect the structure of the machine in which they are implemented.

Although it may be easier to understand the patterns in terms of these structural principles, it is more useful to organize patterns in terms of the problems they solve, since there is a one to one relationship between problem solution patterns and programming construct patterns to implement them.

Using the cognitive schemas as the foundation, an instructional methodology can be designed that leads the student to a better understanding of the underlying patterns and capturing the schemas for himself. A ***schema-driven*** approach will result in a more natural and informative set of learning units. The student is more likely to develop correct schemas from these presentations, examples, drills, and exercises than from the haphazard series of examples from a ***syntax-driven*** approach (see Figures 3A and 3B).

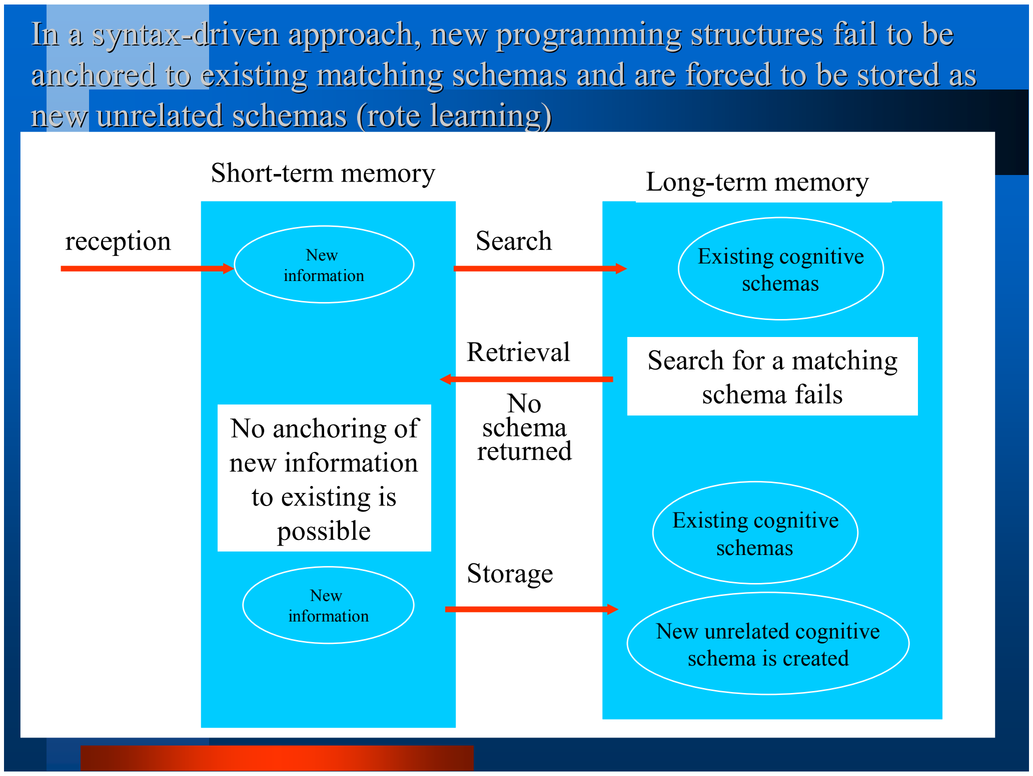
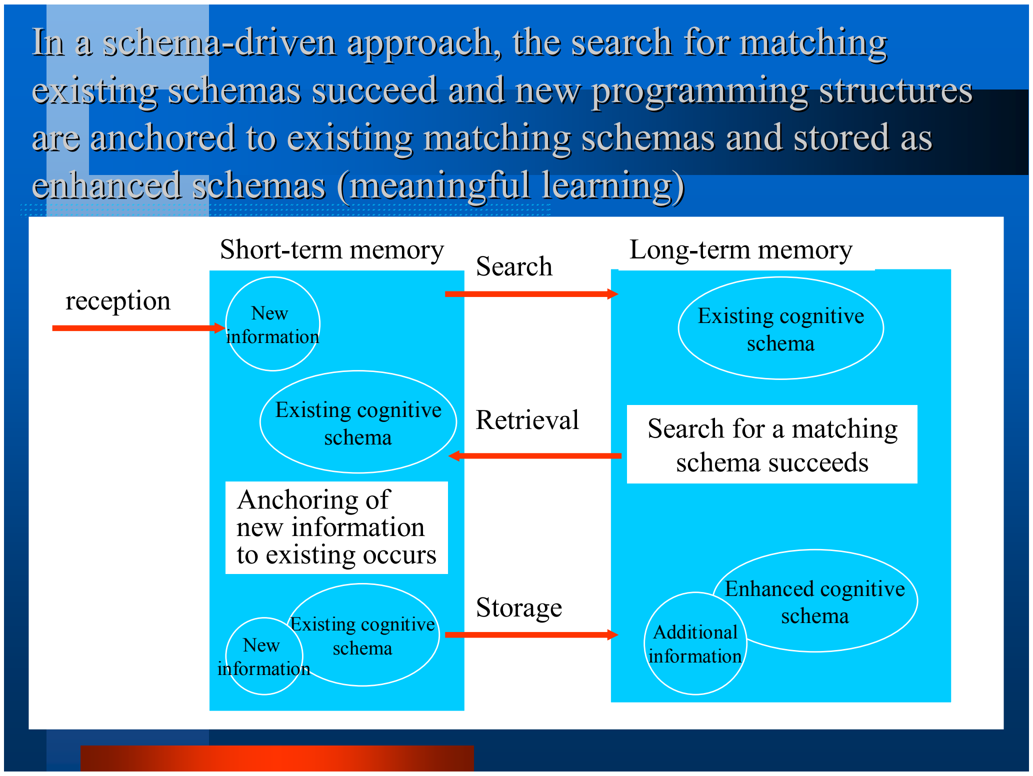

In a syntax-driven approach, new programming structures fail to be anchored to existing matching schemas and are forced to be stored as new unrelated schemas (rote learning)
Short-term memory
Long-term memory
reception
New information
Search
Existing cognitive schemas
Retrieval
Search for a matching schema fails
No schema returned
No anchoring of new information to existing is possible
Existing cognitive schemas
Storage
New information
New unrelated cognitive schema is created

Figure 3A. Syntax-Driven Learning


In a schema-driven approach, the search for matching existing schemas succeed and new programming structures are anchored to existing matching schemas and stored as enhanced schemas (meaningful learning)

Short-term memory
Long-term memory
reception
New information
Search
Existing cognitive schema
Existing cognitive schema
Retrieval
Search for a matching schema succeeds
Anchoring of new information to existing occurs
Storage
Existing cognitive schema
New information
Enhanced cognitive schema
Additional information

Figure 3B. Schema-Driven Learning

In the next section we present guidelines on how to use the schema-driven approach to develop presentation material. Order of presentation of each high-level topic can follow closely the recommended topics of ACM/IEEE CS1. But within each topic, the presentation should follow a general sequence described below. In section 4, these ideas are elaborated in more detail for the specific cases of decision problems and iteration problems. The ideas can be applied to any other topic from the CS1 curriculum as well.

3.3 Guidelines for Schema-Driven Approach in Preparation of Teaching Materials

Regardless of the teaching approach used, a CS1 course will cover a similar range of topics, and the final collection of teaching materials will of course have similarities. The differences between materials generated according to the two approaches will not be found by a comparison of the table of contents, or list of topics to be covered. They are found only through an examination of presentation characteristics for each unit: gradient, chunking, and repetition. The difference is to be found in what drives the presentation of the material and the learning process.

Creating Learning Units using a Schema-driven approach

The order of presentation in each unit should follow these steps:

- Presentation of several problems from a similar class of problems
- Explanation of the general similarity of solution approach

- Introduction of relevant programming language features for this class of problems
- Integration of the problem solution approach with the programming language feature to produce a working program.

The sequence and cognitive contents of the learning units should focus on a specific syntactical element, or group of elements, or structure of the language. Examples are a unit on branching (if/else decisions, etc.) or iteration (do/while loops, etc.). The collection of unit topics is drawn from the traditional structured programming paradigm. For example, we could use the following list of major topics: Storage Concepts (Variables and Types), Operators and Expressions, Assignment, Typecasting, Operator Precedence, Simple and Compound Statements, Decisions, Iteration, Arrays and Strings (other topic lists are possible). Depending on the structure and course coverage, additional units can be created to cover object-oriented features of the language, but these are not part of the current effort.

Below, we describe the components that should make up the learning units. The component list is comprehensive, but for a given unit, only a subset of the full list may need to be used.

1. Description of the general type of problems to be solved, presented in an incremental sequence, with emphasis on the common patterns
2. Presentation of the programming element correlated to the problem solution pattern.
3. Reformulation of a problem solution and the provided programming pattern to give the student the ability to adapt existing patterns to similar problems.
4. Definition and description of programming language constructs, their syntax, and semantics.
5. Representative examples of syntax use: Well-formed statements are used to illustrate the correct use of the syntax.
6. Commonly encountered syntax errors (typical syntax errors due to grammar or spelling), plus examples of common semantic errors (statements that are syntactically correct yet do not encode the intended solution).
7. Constraints, limitations or peculiarities: Whenever an element or structure has a counter-intuitive usage pattern or an underlying syntactical oddity, it is clearly noted.
8. Exercises and examples.

In the next section, we provide two examples of **schema-driven instructional units**. We show the steps that should be covered when designing the presentation and practice for these particular topics. We assume that the student has successfully acquired the schemas for the preceding material and starts with the appropriate prerequisite schemas, plans and code chunks in place. The content of these units is representative of units covering the other topics in the curriculum, which are not shown here.

4. Examples of Instructional Units Derived from the Schema-Driven Approach

4.1 Decision Schema

A **decision problem** is a problem whose solution can be described by choosing a solution to one of a set of subproblems. The original problem can be decomposed into a set of mutually exclusive subproblems that fully cover the solution space of the original problem. A **decision**

schema is a pattern that describes how to solve a decision problem. Concepts regarding solution spaces and subspaces are better communicated by means of tables (or Venn diagrams or other similar notation), and not by showing coded Java solutions prematurely. The solutions of this type of problem translated into code usually take the form of a series of if-else tests or decisions.

Each solution subspace of the original problem is characterized by some **expression** that identifies it as the solution to the problem instance. The solution to the original problem then reduces to determining which subspace possesses the expression that evaluates to **true** (there must be exactly one expression that evaluates to true because of the assumptions regarding mutual exclusion and complete coverage). The solution to the original problem instance then reduces the solution to this problem subspace.

The student must be shown that a problem schema suggests a solution pattern that applies to a group of problems that are all isomorphic (having the same shape). In other words, a schema identifies a group of related problems whose solutions all follow the same general pattern. The pattern must be customized with the details for a specific problem before it becomes a specific solution, but recognizing the general pattern is a critical goal for the student to achieve at this stage.

For the sake of illustration, we present the following group of decision problems:

- assigning a letter grade based on a numerical value.
- calculating the tax due based on an income amount.
- categorizing boxers based on their weight measurement.
- handicapping golfers based on their accumulated point scores.

Below we illustrate the **problem space** for each problem (see Figure 4), noting that each problem defines a domain (a set of input values) and a range (the corresponding result for that set of inputs). The student should find it easy to relate these diagrams to the notion of a mathematical function, which is a transformation of values from the domain to the range.

Letter Grade

Domain	Range
nonnegative integers 0-100	A, B, C, D, F

Tax Due

Domain	Range
nonnegative currency amount	nonnegative currency amount

Boxer Weight Category

Domain	Range
pounds 1-300	Super Heavyweight Heavyweight Middleweight Flyweight

Golf Handicap

Domain	Range
nonnegative integers 0-150	nonnegative decimal value 0.0-15.0

Figure 4. Problem Spaces for Decision Problems

Next, we refine each of the problem spaces to display mutually exclusive, non-overlapping subsets and the criteria that separate them, together with their corresponding results (see Figure 5).

Letter Grade

Membership Criteria for Sub domain	Result for Sub domain
0-59	F
60-69	D
70-79	C
80-89	B
90-100	A

Tax Due

Membership Criteria for Sub domain	Result for Sub domain
\$1 - \$7,200	\$0.00
\$7,201 - \$12,000	12% of Income
\$12,001 - \$22,000	18% of Income
\$22,001 and above	25% of Income

Boxer Weight Category

Membership Criteria for Sub domain	Result for Sub domain
1-120	Flyweight
121-160	Middleweight
161-200	Heavyweight
201-300	Super Heavyweight

Golf Handicap

Membership Criteria for Sub domain	Result for Sub domain
0-30	2% of points
31-60	4% of points
61-80	6% of points
81-100	8% of points
101-150	10% of points

Figure 5. Detailed Decision Plans for Decision Problems

Using tables, the range of possible inputs and results for each decision problem is now explicitly laid out, making it easy for the student to consider the logical connection that must exist between input and result for any given row.

The student is now shown that solving a decision problem can be described as:

- Consider a specific input value for the problem
- Scan or search the problem table to find the input range that the input value belongs to
- Take the corresponding result for that range as the result or answer.

This series of steps is the **decision problem schema.** The student should consider how that, once a decision problem has been captured by describing its problem subspaces and results, the solution for each problem has the **exact same form**. This is a much more important concept for the student to master at this stage than the details of the if-else syntax (to be covered below).

The student is now ready to consider in more detail what is required to take an input and perform range checking. We characterize the criteria for choosing a subspace (one row in the table) as directly relating to an action to be taken for each subproblem. Each subproblem can be described by a condition (boolean expression) that is true for that subproblem and false for any other sub problem. The student now begins to compose the range check condition statements for each row based on the ranges for that row (see Figure 6).

Input Ranges for Raw Input	**Range check as a True/False Condition: If input >= range min AND <= range max Or If input < range min OR > range max**
0-59	If input >= 0 AND input <= 59
…	…
90-100	If input >= 90 AND input <= 100
<0 or >100	If input < 0 OR input > 100

Figure 6. Composing Range Check Expressions

At this point, the student has an initial grasp of the decision problem schema. In the full presentation, the student will be exposed to a graded series of more complex variations of the basic schema described here. Once the problem schemas have been introduced, the student is ready to learn the correlated programming language construct designed to implement program solutions for decision problems, specifically the if-else statement.

Program Element Schemas for Decision Problems: Two-Branch If-Else

We first introduce a simple form of the if-else statement, which corresponds to a decision problem with only two subproblems (sometimes call yes/no problems):

```
if (condition) {... action statements if true ...}
else           {... action statements if false ...}
```

An illustration of the if-else structure related to a problem with only two explicitly mutually exclusive choices is given next:

- **Problem description**: determine if an integer is even or odd
- **Problem solution**: If the remainder of dividing the input value by two is zero then the result is even. Otherwise the result is odd.

We assume the expression **Input** represents the integer being tested. We describe the boolean-valued expression that determines which subspace defines the solution to the problem as the expression `Input%2 == 0` which is read “integer remainder after dividing Input by 2 is zero”. The mapping from condition to result is given by the following table (Table 3):

Table 3. Relating an Input to an Action

Condition	Result	Action
Input%2 == 0 is true	‘E’	`result = 'E';`
Input%2 == 0 is false	‘O’	`result = 'O';`

Placing these elements into the if-else structure yields:

```
char result;
if (Input%2 == 0) { result = 'E';}
else              { result = 'O';}
```

Multi-Branch If-Else

Next we introduce the multi-branch if-else statement:

```
if      (condition_1) action_1
else if (condition_2) action_2
else if (condition_3) action_3
...
else if (condition_n) action_n
else                  default_action
```

Once again, we illustrate the way the language structure relates to a decision making problem with only explicitly multiple mutually exclusive choices.

- **Problem description**: Assign a letter grade based on a numerical grade
- **Problem solution**: Sequentially check all listed criteria to see if the input value is equal or greater than the low range value AND less or equal to the high range value. Once a range is found for which the input value meets the criteria, then assign to the result the value corresponding to that range.

Next we relate the condition expressions in the if-else structure to the criteria for each grade. The criteria are mutually exclusive and involve a compound comparison which yields either a true or false result. A single input check condition followed by its corresponding action would have the form:

```
char result;
if (Input >= LowValue && Input <= HighValue)
      result = <letter corresponding to grade>;
```

The complete series of tests with correct range checks and actions is then:

```
char result;
if       (Input >=0  && Input <=59) { result = 'F';}
else  if (Input >=60 && Input <=69) { result = 'D';}
else  if (Input >=70 && Input <=79) { result = 'C';}
else  if (Input >=80 && Input <=89) { result = 'B';}
else  if (Input >=90 && Input <=100){ result = 'A';}
else  if (Input <  0 || Input > 100){ result = '?';}
```

The student can reinforce the solution approach pattern by implementing the corresponding solution to other similar problems (Income Tax, Boxer Category, Golfer Handicap).

More material for drilling the student on correct usage of the if-else structure can be developed according to guidelines presented above in Section 3.3. For example, a unit on if-else syntax might start with the syntax of the basic two-branch statement:

if (boolean variable/expression)   statement1
else   statement2

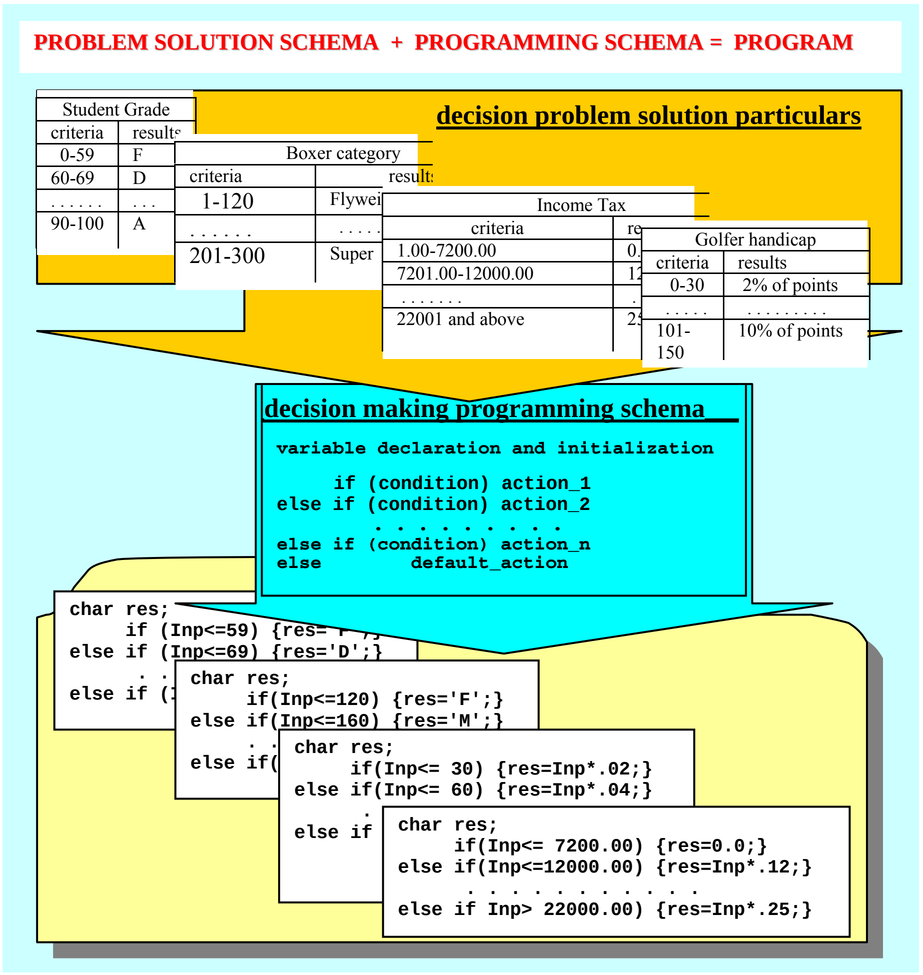
This could be followed by illustrative examples of correct syntax usage:
Given: int amount = 10, positive = 0, negative = 0;

- single statement1/single statement2
```
    if (amount > 0)  positive = amount;
    else             negative = amount;
```
- compound statement1/ statement2
```
    if (amount > 0)  { positive = positive + amount; negative = 0; }
    else             { negative = negative + amount; positive = 0; }
```

Finally, exercises for the student to drill on the syntactic patterns can be provided:

- Exercise #1: For each of the if-else variations in the above examples, produce several examples of your own.
- Exercise #2: Modify some of the correct examples given, to intentionally create both invalid and erroneous statements and indicate how erroneous statements will be interpreted.

By the end of this unit, the student will have expanded her problem solving and programming schemata to solve and code programming problems involving explicit or implicit mutually exclusive decision making (see Figure 7).


PROBLEM SOLUTION SCHEMA + PROGRAMMING SCHEMA = PROGRAM
decision problem solution particulars
decision making programming schema

Student Grade	
criteria	results
0-59	F
60-69	D
. . . . . .	. . .
90-100	A

Boxer category	
criteria	results
1-120	Flywei
. . . .	. . . . .
201-300	Super

Income Tax	
criteria	re
1.00-7200.00	0.
7201.00-12000.00	12
. . . .	.
22001 and above	25

Golfer handicap	
criteria	results
0-30	2% of points
. . . . .	. . . .
101-150	10% of points

```
variable declaration and initialization

     if (condition) action_1
else if (condition) action_2
         . . . . . . . . .
else if (condition) action_n
else         default_action
```

```
char res;
     if (Inp<=59) {res='F';}
else if (Inp<=69) {res='D';}
   . .
else if (I
```

```
char res;
     if(Inp<=120) {res='F';}
else if(Inp<=160) {res='M';}
     . .
else if(
```

```
char res;
     if(Inp<= 30) {res=Inp*.02;}
else if(Inp<= 60) {res=Inp*.04;}
          .
else if
```

```
char res;
     if(Inp<= 7200.00) {res=0.0;}
else if(Inp<=12000.00) {res=Inp*.12;}
      . . . . . . . . . . .
else if Inp> 22000.00) {res=Inp*.25;}
```

Figure 7. Solution Schema (for Decisions) + Program Schema = Program

4.2. Iteration Problem Schema

An **iteration problem** is a problem whose solution can be described primarily as the repeated application of a solution to an associated subproblem. Each application of the solution of the subproblem is called an iteration. The iterations may be and often are enumerated. For example, if a solution requires n iterations, the individual iterations can be individually identified by enumerating them from 1 to n (or sometimes from 0 to n – 1). The iterations are usually executed sequentially in enumeration order on typical computer hardware, but this is not a requirement.

We make a distinction for the student between iterations that are independent of the enumeration value, and those that are dependent on it. Even in the case of independent iterations, the iterations must still be at least counted in order to determine when to stop.

The novice programmer must master iteration as a fundamental abstraction as a key to achieving adequate programming competency. It is therefore crucial that he first learn to recognize problems for which iteration is the primary solution approach. We identify a hierarchy of schemas that are connected by an appropriate gradient of increasing complexity.

The Iteration Schema

In its most basic form, the iteration schema can be described as follows:

Given a complex problem:
- Identify a more basic subproblem (the base operation)
- Describe the solution to the original problem as n applications of the solution to the subproblem (perform the base operation n times)

At this stage, the student should be focusing on how to read the original problem and infer the existence of a more basic easier-to-solve problem. At this stage, the student should be aware that the solution to the basic problem must be performed multiple times, but there should not be too much emphasis on quantitative details such as how to count the iterations or how to stop them when a limit is reached.

Simple Iteration Schema

The first increase in complexity is to introduce some precision about enumerating and counting the iterations, and identifying a limit on how many iterations are required.

- Identify the base operation
- Identify the number of iterations **n**
- Perform the base operation **n** times.
- [base operation is independent of the iteration count, order doesn't matter]

At this step, we emphasize the role played by n, the number of times the base operation must be performed to achieve the solution to the original problem. The iterations are enumerated from 1 to n (or 0 to n – 1), but this is not emphasized, since the base operation has no dependency on the enumeration value.

Example: Simple Iteration Schema to Calculate Compound Interest

This is a very popular application of the iteration pattern and is seen in many programming texts. The base calculation is simply: Ending Principal = Starting Principal X (1 + Interest Rate) (see Table 4).

Table 4. Base Calculation for Compound Interest

Start Prin	**Int Rate**	**Int**	**Ending Prin**
1000	0.05	50 (=1000 X 0.05)	1050 (= 1000 + 50)

The base calculation gives the value of the account after 1 year of investment. The iterative step finds the value of the account after n years of investment, by simply applying the base calculation n times (see Table 5).

Table 5. "Unrolled" Repetition of Base Calculation for 10 Years

Year	**Starting Principal**	**Int Rate**	**Interest**	**Ending Principal**
1	$1,000	5%	$ 50 (=1000 X .05)	$1,050 (=1000+50)
2	$1,050	5%	$ 52 (=1050 X .05)	$1,102 (=1050+52)
3	$1,102	5%	$ 55	$1,157
…	…	…	…	…
9	$1,477	5%	$ 74	$1,551
10	$1,551	5%	$ 78	$1,629

The student may need to be shown that in the repeated calculation, the starting principal for each step is copied from the ending principal from the previous step. The **schema** that captures this computation is as follows:

```
Given
    Starting Principal SP
    Interest Rate R
    Base Computation is "Ending Principal EP = Starting Principal X (1 + Interest Rate)"
    No. of Years N
Do the Base Computation N Times
    EP = SP X (1 + R)
    SP = EP
Answer is EP = total value of investment after N years
```

In order to adapt the base computation to an iteration, the ending principal from the previous computation must become the starting principal for the next computation. With the base computation defined in this way, the values of the enumerations of the iterations (in this example, the "year" column) do not directly enter into the computation of the interest, but the student should notice that the year value is still relevant to the behavior of the iteration. It is the value of the year reaching a limiting value that causes the table to end at the row for year 10.

In other words, the value of the year column is introduced into the schema as the iteration counter. The schema then simplifies to:

- Set year to 0
- Do the base computation for each value of year from 0 to 10
    - EP = SP X (1 + R)
    - SP = EP

The Counter and Accumulator Schemas

At this stage, we become more explicit about the enumeration value and introduce a symbol or variable, called the **counter**, to represent it (traditionally, names such as "i" "j" "k" are picked, but this is not a requirement). In addition, we introduce the idea that the counter must be explicitly incremented as a way for the schema to dynamically keep track of how much work has occurred. The counter counts from 1 to n, allowing the iterations to each be enumerated. As mentioned earlier, the steps in the iteration must be performed sequentially in time. The counter schema still executes the base operation n times, but the iterations can now be distinguished from one another by the value of the counter. The iteration for which the counter has value 1 will usually be performed first, then the iteration for which the counter has value 2, and so on until the counter has value n. It's not always required to execute the iterations in the order 1 to n, but this is usually done, if for no other reason than it's usually the easiest to implement.

The schema is now:
- Identify the base operation
- Identify the number of iterations n
- Identify the counter and initialize it to 1.
- Perform the base operation n times, with the counter value increased by 1 each time.

To describe the accumulation pattern using summation notation, it is customary to identify the values with a subscripted variable. In computer programming, subscripted variables are usually implemented as arrays. The discussion of arrays is deferred in the current paper, although the treatment of problem and program schemas for arrays is a worthwhile area for future development.

We can then describe the **accumulator schema** as the **counter schema** augmented by an accumulator. The accumulator represents a value that is incremented by the base operation for each iteration. It is initialized to zero, and each iteration performs a computation whose result is added to the accumulating value. This pattern is useful for describing the calculation of any mathematical expression that is described in summation notation. To define the accumulator schema, we add another symbol to represent the accumulator, and we use the value of the counter as the subscript value. The general schema then looks like this:
- Initialization: initialize the accumulator to zero, and the counter to 1
- Base operation: compute the ith value (where i = the value of the counter), add it to the sum, and increment the counter by 1.
- Iteration step: perform the base operation n times.
- Result: final value of the accumulator

Example: Simple Series Approximation for π: 4*(1 – 1/3 + 1/5 – 1/7 …)

The solution to this problem is an example of the accumulator schema. The computation during each iteration depends on the iteration's enumeration value. As with any computation expressible in summation notation, it will be helpful for the student to:

- Place the terms of the expansion in one-to-one correspondence with the enumeration values
- Derive an expression for the general term as a function of the enumeration variable i.
- Describe how the accumulation schema can be applied to calculate the final result.

As with any nontrivial computation being presented to a novice programmer, the instructor should make sure that the student knows what the computation looks like before proceeding to any higher level of abstraction. A tabular presentation is a good way of "unrolling" or tracing most iterative computations that do not involve any detail (at this point) on how the computation will be performed on the computer (see Table 6). The student should be encouraged to calculate several terms in "pencil-and-paper" mode to gain confidence that the computation is not difficult to understand and is ultimately just a series of arithmetic operations, applied according to a precise recipe or algorithm.

Table 6. Illustration of the Iterative Computation of the π Series

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**…**
term	+1/1	-1/3	+1/5	-1/7	+1/9	-1/11	+1/13	…
sum	1.00000	0.666667	0.866667	0.72381	0.834921	0.744012	0.820935	…
4 X sum	4.00000	2.666667	3.466667	2.895238	3.339683	2.976046	3.04184	…

The next level of inference is to guide the student in the derivation of the general term as a function of the enumeration i. Since the first term is positive and the first term has an enumeration value of 1, the correct expression for sign is **`(-1)`**$^{(i + 1)}$. The student should then note that, ignoring the sign, each term is the reciprocal of the next odd integer, so the general term as a function of i is **`1/(2 * i - 1)`** Combining this with the expression for sign yields a general term of **`(-1)`**$^{(i + 1)}$ **`* 1/(2 * i - 1)`**

Substituting this base calculation into the accumulator schema then gives:

- Initialization: initialize the accumulator to zero, and the counter to 1
- Base operation: compute **`(-1)`**$^{(i + 1)}$ **`* 1/(2 * i - 1)`**, add it to the sum, and increment the counter by 1.
- Iteration step: perform the base operation n times.
- Result: final value of the accumulator

A final step for this particular series is to either multiply each term by 4, or multiply the final result by 4, since the series is an approximation for pi/4. The instructor can decide which is best for the student.

Programming Schemas

Once the student has had some practice with several examples that illustrate the common pattern of the iteration and accumulator patterns, he is ready to explore the related programming language constructs designed to simplify the implementation of the iteration schema in a computer program.

Schema for Counting with a While Loop

- Determine the base computation.
- Let n represent the number of iterations required to solve the problem
- Let i represent the enumeration value for the iteration.
- If i <= n then
    - Perform the base computation
    - Increase the value of i by 1

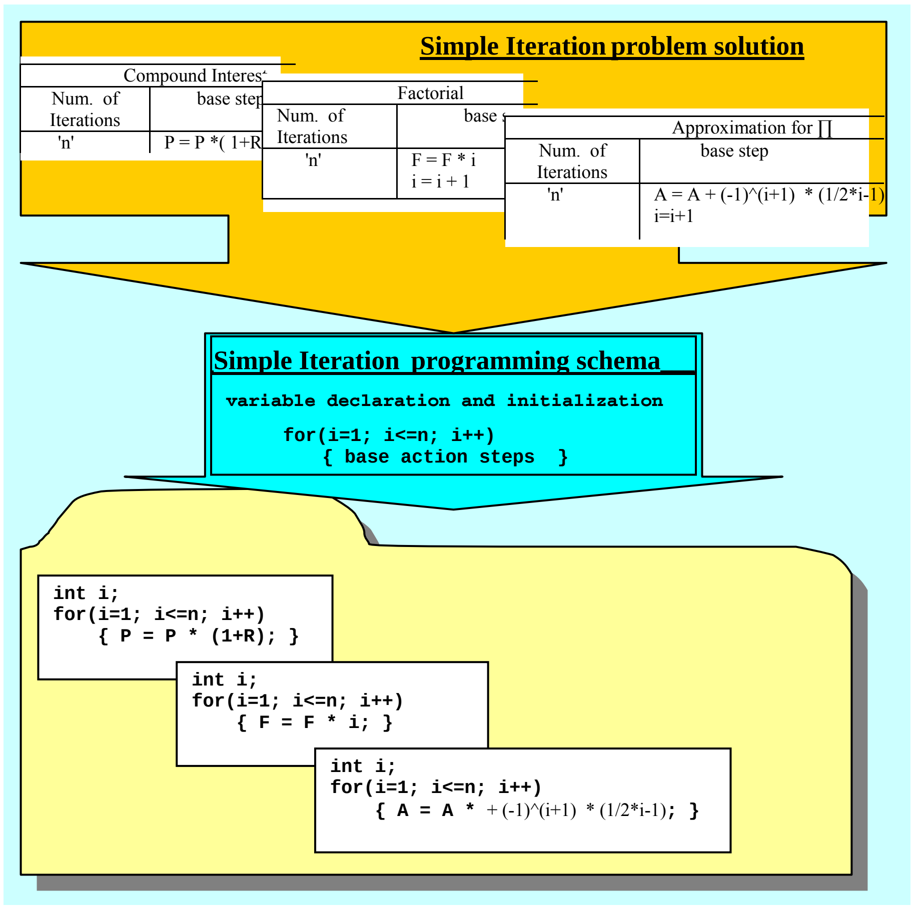
Schema Implemented with a While Loop

```
int n = 10; // set limit
int i = 1;
while (i<=n) {
      // Perform the base computation here …
      i = i + 1;
}
```

Schema Implemented with a For Loop

```
int n = 10; // set limit
for (int i=1; i<=n; i++) {
      // Perform the base computation here …
}
```

To complete the unit, the student must integrate the solution schemas with the program schemas to produce a complete program to solve that specific problem (see Figure 8). As was illustrated in the previous section on decision problems, the instructor should emphasize how the problem schema serves as the common pattern that underlies the program solution to a class of similar problems.



```
int i;
for(i=1; i<=n; i++)
    { P = P * (1+R); }
```

```
int i;
for(i=1; i<=n; i++)
    { F = F * i; }
```

```
int i;
for(i=1; i<=n; i++)
    { A = A * +(-1)^(i+1) * (1/2*i-1); }
```

Figure 8. Solution Schema (for Iterations) + Program Schema = Program

5. Conclusion

Need for a New Teaching Approach for the CS1 Curriculum

College-level CS educators are under increased pressure today to determine the causes of the perceived high failure rate in CS1. Assessment and accreditation activities are increasingly metrics based, and CS departments are expected to collect and analyze data on student achievement of learning objectives, and take corrective action if goals are not met. Although hard data is still lacking, many CS educators are suggesting that the current CS1 teaching approach needs to be adjusted to meet the needs of the current audience of novice programmers. This paper agrees, and we have focused on examining current methods (syntax-driven) and suggesting new ones (schema-driven).

Root Causes of the Problem

Today CS is a popular academic major, yet the CS teaching approach is not mature, compared to other math, science, and engineering disciplines. This is understandable, since many CS educators did not graduate with a degree in CS (such departments only began to appear in the late 1970s to early 1980s). The early student audience for the CS curriculum was composed of bright self-starters who mastered the material on their own, regardless of the teaching method, or in the absence of any method at all. Today, in contrast, students are no longer exclusively an elite who learn on their own without a method or approach. Abilities and aptitudes span the Bell curve. Current textbooks have not caught up to this reality yet, and they proceed from an inaccurate model of what a true programming novice knows or does not know. Specifically, textbooks greatly overestimate the novice student's problem solving skills, assuming that they have already mastered sophisticated problem solving strategies. Many of these texts do not teach such strategies, and focus instead on the presentation of the syntax of programming language constructs, using problems as illustrations.

Another modern phenomenon that calls for an adjustment in the current teaching approach is the apparent low quality of cognitive skills that the current generation of young novice programmers brings with them to CS1. This is popularly blamed on exposure to today's high-speed, short-attention-span entertainment and activities. Cognitive psychologist Elliot Soloway refers to young college students as the "Nintendo or MTV generation" and notes that the students' perception of technology and media has been profoundly influenced by these sources. He also concludes that current failure rates for CS1 "… echo our research from 15 years ago, so it's not clear we ever have figured out how to teach programming…". [10]

New Approach Based on Cognitive Learning Models

In order to propose a new approach, we took as our starting point well-known results from research in cognitive psychology on how students learn and acquire skills. The results show that all learners use short-term and long-term memory in characteristic ways, and that each type of memory works with its own learning mechanisms. We applied these well-known results to the specific problem of creating a methodology for teaching programming to novices. We concluded that existing approaches depend almost exclusively on the mechanisms of the student's short-term memory (the syntax-driven approach), having them memorize collections of facts about the syntax of the programming language. In this approach, the student is apparently expected to infer sophisticated problem solution patterns on their own. In contrast, we conclude the ability to recognize problems as belonging to class of problems with known solution patterns is what the student should be taught (the schema-driven approach), rather than assuming the student already knows it. Attention to language syntax comes later. In fact, memorization of syntax is worthless to the student, unless they have first acquired the problem and solution schemas to which these programming language facts can be anchored.

Preliminary Results

Ungraded homework exercises designed using the schema-driven approach have been presented to students of the CS1 course given at CSUN during the Spring and Fall 2002 semesters and the Spring 2003 semester. It is inconclusive whether these exercises improved the students' ability to complete their programming projects. To perform a more controlled evaluation with quantitative data, a closer coordination between lecture and lab material would be implemented. In addition, homework exercises would have to closely follow the lecture presentation of the material, a percentage of the final class grade would have to be assigned to the exercises, and the exercises would have to be graded, evaluated and correlated with the student overall performance.

In the meantime, we have strong anecdotal feedback that many students had a positive perception of the benefits of the exercises. During informal exchanges, students with no previous programming background who completed the assigned homework units reported that doing the exercises made it easier for them to code the assignments. Other students with some prior programming background reported that just doing a few exercises was sufficient to grasp the syntax for a particular language component. Still others, mostly novices, who complained of having problems with the programming assignments, candidly revealed that they either did not have the time, or thought it was not necessary to do the exercises or found the exercises boring. Our observations align closely with reports on experiments from the literature [12].

Ideas for Future Work

We hope to conduct controlled studies in the future in order to assess the validity and level of benefit of the schema-driven methodology. The results and feedback from these studies would further illuminate the design and organization of the curriculum. The studies would test the hypothesis that novice programmers learning to program will demonstrate better problem solving and coding abilities when the schema-driven methodology is used in preparing the presentation and designing the examples, exercises and programming projects. The experiment would include an experimental group—those that would be taught using the schema-driven approach—and a control group—those taught using the syntax-driven approach. Ideally, the two groups should be composed of students whose skills and backgrounds are relatively balanced and homogeneous. We feel confident that the results should show better performance for the experimental group. It is our hope that this paper will assist the efforts of other Computer Science educators in their common quest for finding the right answer for how to teach programming to today's novice programmers.

Bibliography

1. E. Soloway and J.C. Spohrer (Eds.), "Studying the Novice Programmer", Lawrence Erlbaum Associates, Publishers, 1989.

2. E. Soloway, et al, "Cognitive strategies and looping constructs", In [1].

3. E. Soloway, K. Ehrlich, J. Bonar, J. Greenspan, "What do novices know about programming?", In A. Badre and B. Shneiderman (Eds.) Directions in Human-Computer Interaction, Ablex, 1982.

4. K. Ehrlich, E. Soloway. "An empirical investigation of the tacit plan knowledge in programming", In J.C. Thomas, and M.L. Schneider (Eds.), Human Factors in Computer Systems, 1984.

5. E. Soloway, K. Ehrlich, "Empirical studies of programming knowledge", IEEE Transactions on Software Engineering, vol. 10 pp. 595-609, 1984.

6. Ilene Burnstein et al, "A role for Chunking and Fuzzy Reasoning in a Program Comprehension and Debugging Tool", Proceedings of the 9th International Conference on Tools with Artificial Intelligence, IEEE, 1997.

7. M. V. Doran and D. D. Langan , "A Cognitive- Based Approach to Introductory Computer Science Courses: Lessons Learned" , SIGCSE Bulletin, Vol 27, No 1, pp 218-222, 1995.

8. A. Carbone, et al., "Principles for designing Programming Exercises to minimise poor learning behaviours in students" ,2000. URL: http://www.acm.org/pubs/citations/proceedings/cse/359369/p26-carbone/

9. Angela Carbone, et al., "Characteristics of Programming Exercises that lead to Poor Learning Tendencies: Part II", 2001.URL: http://www.acm.org/pubs/citations/proceedings/cse/377435/p93-carbone/

10. Mark Guzdial and Elliot Soloway, "Teaching the Nintendo Generation to Program ", COMMUNICATIONS OF THE ACM, April 2002/Vol. 45, No. 4., pages 17-21.

11. Richard E. Mayer, "The Psychology of How Novices Learn Computer Programming",In [1].

12. Richard E. Mayer, Jennifer L. Dyck and William Vilberg, "Learning to Program and Learning to think: What's the Connection", In [1].

13. J.B. Black, D.S Kay, and E.M. Soloway, "Goal and Plan Knowledge Representations", In J.M. Carroll (Ed.), Interfacing Thought, The MIT Press, 1987.

14. Tim Mattson, "A Cognitive Model for Programming", URL: http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/Background/Psychology/CognitiveModel.htm

Biographical Information

RICHARD G. COVINGTON is an Assistant Professor in the Department of Computer Science at California State University Northridge. His research interests include Computer Architecture and Simulation, Graphical User Interfaces, and Non-Western Language Information Processing. He received his Ph.D. in Electrical and Computer Engineering from Rice University in 1989.

LEO BENEGAS is a Lecturer in the Department of Computer Science at California State University Northridge. His research interests include Artificial Intelligence, Mobile Agents, and Cognitive Science. He received his M.S. Degree in Computer Science from California State University Northridge in 2003.