July 2015

Volume 6 Issue 1

JoCsE

Journal Of Computational Science Education

Promoting the Use of Computational Science Through Education

ISSN 2153-4136 (online)

Volume 6, Issue 1 **July 2015**

JOCSE

Journal Of Computational Science Education

The Journal Of Computational Science Education (JOCSE), ISSN 2153-4136, is published quarterly in online form, with more frequent releases if submission quantity warrants, and is a supported publication of the Shodor Education Foundation Incorporated. Materials accepted by JOCSE will be hosted on the JOCSE website, and will be catalogued by the Computational Science Education Reference Desk (CSERD) for inclusion in the National Science Digital Library (NSDL).

Subscription: JOCSE is a freely available online peer-reviewed publication which can be accessed at `http://jocse.org`.

Contents

Introduction to Volume 6 Issue 1

Steven I. Gordon
Editor
Ohio Supercomputer Center
Columbus, OH
sgordon@osc.edu

Forward

The articles in this issue of the Journal of Computational Science Education provide two approaches to teaching introductory programming. In addition, it provides insights in a student article summarizing work to identify potential drugs to treat tuberculosis.

The article by Viera et. al. discusses the use of worked examples as an approach to teaching introductory programming. Student performance with and without worked examples was compared using three exercises. The article provides insights into the construction of such examples as well as their impacts on learning outcomes.

Ballesteros et.al. describe an introductory programming language "Picky" that is designed to help students learn introductory programming concepts more easily.

Finally, the student article by Priest et.al. details their experience in using drug docking applications to screen potential drugs that target the enzymes involved in tuberculosis. The students were able to screen over 4 million potential drug molecules against two enzymes critical to the survival of *Mycobacterium tuberculosis.*

Exploring Design Characteristics of Worked Examples to Support Programming and Algorithm Design

Camilo Vieira
Purdue University
401 N Grant St Office 372,
West Lafayette, IN 47907
+1(765)250-1271
cvieira@purdue.edu

Junchao Yan
Purdue University
401 N Grant St Office 372,
West Lafayette, IN 47907
+1(765)337-8867
yan114@purdue.edu

Alejandra J. Magana
Purdue University
401 N Grant St Office 256,
West Lafayette, IN 47907
+1(765)494-3994
admagana@purdue.edu

ABSTRACT
In this paper we present an iterative research process to integrate worked examples for introductory programming learning activities. Learning how to program involves many cognitive processes that may result in a high cognitive load. The use of worked examples has been described as a relevant approach to reduce student cognitive load in complex tasks. Learning materials were designed based on instructional principles of worked examples and were used for a freshman programming course. Moreover, the learning materials were refined after each iteration based on student feedback. The results showed that novice students benefited more than experienced students when exposed to the worked examples. In addition, encouraging students to carry out an elaborated self-explanation of their coded solutions may be a relevant learning strategy when implementing worked examples pedagogy.

Categories and Subject Descriptors
K.3.2 [**Computers And Education**]: Computer and Information Science Education – *Computer science*

General Terms
Algorithms, Human Factors.

Keywords
Computational Thinking, Programming Education, Worked Examples.

1. INTRODUCTION
Computational thinking [24] has emerged as a set of concepts and skills that enable people to understand and create tools to solve complex problems [20]. Many programming and algorithm design processes have been proposed as part of this set of understandings and skills [7, 8]. Hence, it is relevant to introduce programming and algorithm design as part of undergraduate courses; however, learning to program is a complex task [19]. Thus, it is necessary to explore scaffolding strategies to introduce these computational thinking skills. The use of worked examples has been demonstrated to be an effective approach for supporting complex learning when it is guided under certain principles [5]. It can reduce the extraneous cognitive load, which is not beneficial to learning. Therefore, it allows the learner to devote cognitive resources to useful loads.

This study explores how worked examples can be paired with programming and algorithm design. The guiding research questions are:

- How can worked examples be effectively designed to introduce programming concepts to novice learners?
- How do students self-explain worked examples when approaching a solution to a programming assignment?

2. BACKGROUND
Learning how to program is a difficult task [19]. Programming courses are considered the most challenging at the undergraduate level as they often have the highest dropout rates. In order to learn to program, a student has to understand (a) the purpose of a program, (b) how the computer executes programs, (c) syntax and semantics of the programing language, (d) program structure, and (e) how to actually build a program [9]. Since the learning process involves many steps, these myriad steps may generate a high cognitive load for students who have no previous experience in algorithm design or programming languages.

Researchers have identified differences in the way novices and experts experience programming tasks. Experts use specialized schemas to understand a problem based on its structural characteristics [19]. They use problem solving strategies, such as decomposing the program and identifying patterns, in order to approach a solution [18]. Language syntax and analyzing line-by-line details of programs tend to be the focus of novices due to the superficiality of these skills in the hierarchy of knowledge. [18]. They usually have problems related to language constructs, such as variables, loops, arrays, and recursion.

The use of worked examples (WE) has been recognized as a relevant strategy for supporting novices in learning tasks that involve a high cognitive load. Worked examples approach is guided by principles associated with Cognitive Load Theory (CLT). CLT is a recognized theory that focuses on cognitive load processes and instructional design [15]. CLT establishes a cognitive architecture to understand how learning occurs. The cognitive architecture structures memory that comprises a limited working memory and a vast long-term memory [10]. CLT states

that there is a cognitive load generated when learning occurs. This load can be affected by the learner, the learning task, or the relation between the learner and the learning task [10].

There are different types of cognitive loads: (1) intrinsic load, which is the inherent load to the difficulty of the learning task; (2) germane load, which is comprised of required resources in the working memory to manage the intrinsic load, which, in turn, support learning; and (3) extraneous load, which refers to the load that arises from the instructional design and does not directly support learning [17]. To improve the learning process, the extraneous load should be minimized so that the germane resources can be maximized.

Figure 1 shows a representation of the learning process based on the CLT. In Phase 1, a student is working on a task that has different forms of representations. In Phase 2, the cognitive process that takes place is depicted. In Phase 2a, different senses process the information the student is receiving. In Phase 2b, the limited working memory (three to five chunks of information) is assigned to germane or extraneous resources depending on how the load arises from the learning experience. In Phase 2c, schemas are created and stored in the long-term memory when learning takes place.

Novices usually work backward to solve problems using a means-ends analysis. Students need to fill the gap between the initial problem state and the final goal, and this search process generates a cognitive load. Some of the instructional design techniques that have been proposed to reduce the cognitive load are: goal free effect, worked examples effect, and completion effect [10]. The goal free effect suggests that, by removing a specific goal from the problem, the learners will work forward, state by state as experts do. Thus, the cognitive load is reduced because the students only have to consider the current state and the next state of the problem. The worked example effect occurs when students are exposed to an expert solution to the problem. These examples allow learners to start solving similar problems by analogy, thereby reducing the cognitive load. Finally, the completion effect refers to problems with a given partial solution that are provided to students to complete. Completion effect examples are gradually modified to present less information to the student. This approach is also called faded worked examples (FWE), and it has shown positive results in reducing cognitive loads in the domains of mathematics and programming.

Atkinson et al. [5] proposed instructional principles for the design of WE based on several studies. According to these principles, a WE should include: (1) A problem statement; (2) A procedure for solving the problem; and (3) Auxiliary representations of a given problem. Atkinson's principles and their adaptations for this study are summarized in Table 1.

When WE are used as part of the learning process, the student goes through a four-stage process described by the theory of Adaptive Control of Thought-Rational (ACT-R) [4]. According to this theory, the skill acquisition process is composed of four stages in which knowledge transitions from declarative to procedural [5]. During the first stage, the students solve problems by analogy using worked examples. Then, in the second stage, the students use abstract declarative rules gathered from the examples. When students get to the third stage, declarative knowledge is already acquired and stored in their long-term memory. Procedural rules have also started to become clearer to students by practice. Therefore, students are able to respond automatically and faster to familiar problems. During the last stage, once students have been exposed to several examples, they are able to solve many different problems on their own.

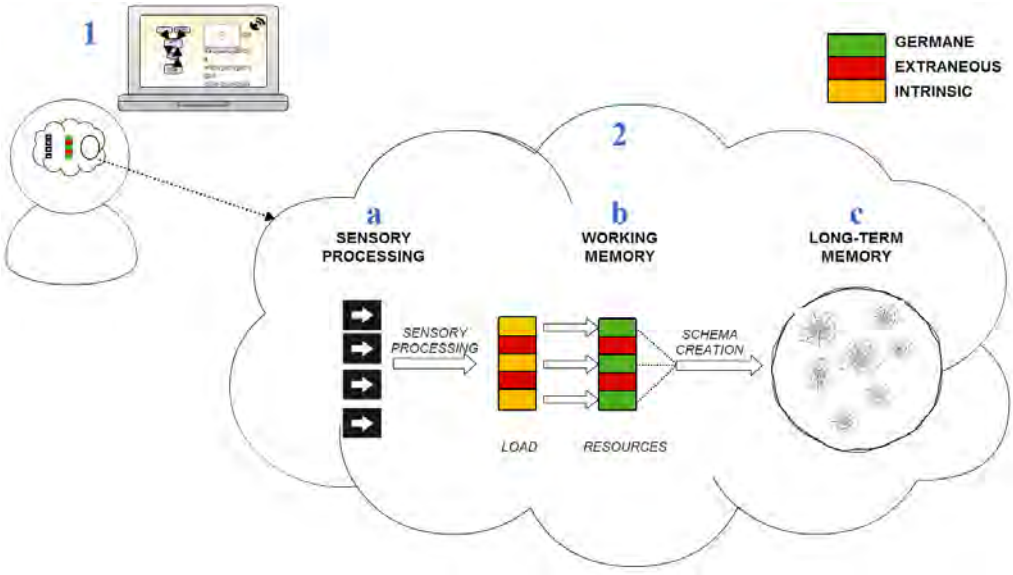


Figure 1. Learning process from a CLT perspective. In (1) the learner is studying the materials. In (2) the learning process takes place: (a) Senses capture information; (b) Different forms of cognitive load make use of working memory; (c) Schemas are created and automatized in long-term memory

Table 1. Effective features of worked examples described by Atkinson and collaborators [5]. The right column describes our adaptation for these learning tasks

Feature	Description	Our Adaptation
Intra-Example	• The use of multiple formats and resources is important when designing WE; however, different formats should be fully integrated to avoid extra cognitive load generated by the split attention effect. • The example should be divided in sub goals or steps to make it easier for the student to understand. Labels and visual separation of steps can be used for this purpose.	• The examples contained multiple forms of representations including C# code, visual flowchart algorithm description, and verbal explanations of the approach. • Each representation was segmented in steps toward the solution. These steps were aligned with the representations.
Inter-example	• The variability of problems during a lesson can offer learning benefits, but it is important to reduce the cognitive load when using techniques such as WE. • The use of multiple WE (at least two examples) with structural differences can improve the learning experience. The WE should be presented with similar problem statements that encourage the students to build schemas based on analogies and the identification of declarative and procedural rules.	• Different problems were approached during each lab session. • Two examples were provided and students were required to complete at least three additional programming challenges. • All the examples and challenges were focused on a specific topic for each lab session (e.g. loops, creating arrays, searching in arrays).
Environmental	• Students should be encouraged to self-explain the WE in order to be actively engaged with them. • Some strategies that support this process are: (1) Labelling WE and using incomplete WE; (2) Training Self-Explanations; and (3) Cooperative Learning.	• One of the examples did not have the verbal explanation (i.e. in-line comments of the programming code). • Before starting to solve the assignment, students were asked to comment on the code for the example that did not have verbal explanation. This activity was intended to encourage self-explanation of the examples. • Some of the assignments could be built from the examples.

2.1 Previous Experiences

Guzdial [11] has advocated for an approach to programming education other than a common approach, which asks students to just start building a program. Based on Kirschner, Sweller and Clark [12], he argued that, "expecting students to program as a way of learning programming is an ineffective way to teach" (p.11). As an alternative, he proposed an approach based on the work of Pirolli and Recker [16] who used WE and cognitive load theory to introduce programming concepts. In one of their experiments, Pirolli and Recker explored how transfer occurs in learners, starting with examples and moving on to programming problems on Lisp. To implement the examples, each lesson started by having students read the textbook and analyze WE. The students then used this knowledge to find a solution for an assigned problem. Authors hypothesized that the problem solving process enriched declarative knowledge as well as procedural knowledge.

The declarative knowledge in programming includes code structure, programming abstractions, functionality of the abstractions, and purposes and operation of the program. All these elements are represented as a mental model.

On the other hand, the procedural knowledge comprises the construction, manipulation, and interpretation of this model. In their experiments, Pirolli and Recker [16] found that worked examples were useful in building these mental models by "providing [students with] concrete referents for abstract discourse and newly introduced concepts and propositions" (p.273).

In another study, Moura [13] used Portugol, a tool for learning algorithms, for students to understand a given example by visualizing the execution of the algorithm. She found that, although students took some time to get used to the tool, once they did get used to it, they performed better on assessment tests when learning computing science fundamentals. Regarding the implications of the study, Moura suggested that an effective way to help students learn how to program requires an easy-to-use tool as well as assigning some pre-training time for the students to get familiar with it.

This study focuses on a strategy for providing worked examples to an introductory programming course to support student learning of process of loops and arrays concepts. The worked examples were designed following the principles by Atkinson and colleagues [5] as described on Table 1.

3. METHODS

This study followed a Concurrent Mixed Methods Research process design [23]. This design includes one quantitative strand (pretest, posttest, survey, and lab scores) and one qualitative strand (open ended questions and comments in the code of the examples). Each strand was analyzed independently. At the end, the identified commenting styles were related with the quantitative measures to evaluate whether there was a trend in the way students experienced the use of examples.

3.1 Participants

The participants of this study included thirty-five undergraduate students majoring in Computer and Information

Technology at a large midwest university. As part of an introductory programming course, they were exposed to weekly lab sessions where they applied programming concepts learned in lecture. Three of these weekly sessions (8th, 9th, and 10th) were used to evaluate the WE approach. The sample size as well as the participants slightly varied from session to session since not all students attended all the sessions or completed pretest and posttest assessments.

These students were divided into two different groups. For lab session #8, both groups used worked examples. For lab sessions #9 and #10, one group was considered the experimental group (using WE) while the other one was the control group. The control group continued doing the lab session as they were used to; that is, solving the assigned problems based on what was learned during the lectures without additional scaffolding but only the help provided by the teaching assistant. Table 2 summarizes participants' information and configurations for each of the sessions.

Table 2. Number of participants per session

Session	Group	Number of Participants	Programming Experience
8th	Experimental	28	12
	Total	28	12
9th	Experimental	19	10
	Control	15	7
	Total	34	17
10th	Control	16	9
	Experimental	14	8
	Total	31	17

3.1 Materials

Two examples designed by following the instructional principles of worked examples [5] were provided to the students in the experimental group. The examples were composed of a Visual Studio Solution with the programmed worked examples as well as a matched flowchart representing the solution. Figure 2 depicts an example of what was provided to the students. On the left side, the C# code adds the even numbers from 0 to a variable n1. The code included comments to explain each section. The right side of the figure showed a flowchart describing the algorithm design for this particular implementation. This design was coupled with textual description. All the elements in the example were identified by a code that allowed the students to match the different representations (i.e. steps in the flowchart with segments of code).

Two examples were provided to the students per session. Both had the distribution depicted on figure 2; however, the comments within the code were not included in the second example because the students were required to complete that portion as part of the assignment. With this design, we expected that students would start using the examples to solve the problems by analogy. Then, having acquired some declarative and procedural rules, students were expected to be able to solve the different and more challenging problems on their own.

3.2 Procedures

The research protocol consisted of a series of tasks. The first task was a pretest aligned to the learning objectives of the lab session. The students were given 10 minutes to complete the test before starting the session. The next tasks consisted of exploring examples and commenting on the code in one of the examples in order to self-explain it. The fourth task was to solve three additional assignments using the examples whenever they were necessary. Finally, students completed a posttest and survey related to their perceptions regarding the use of worked examples. The students could take as much time as they needed to complete this task within a period of two hours. The assignments had to be turned in before proceeding to the next assignment. The collected data included the pretest, posttest, survey data, commented example, and programming projects.

Following design-based research approaches [21], this study includes three iterations, one for each lab session. Right after the session, the tests and survey were analyzed. This information was used to refine the examples, assignments, and instruments for the subsequent iteration. For example, after the first iteration, some students mentioned that the comments in the code were very detailed decreasing the code's readability. Therefore, the next iteration examples included simpler comments.

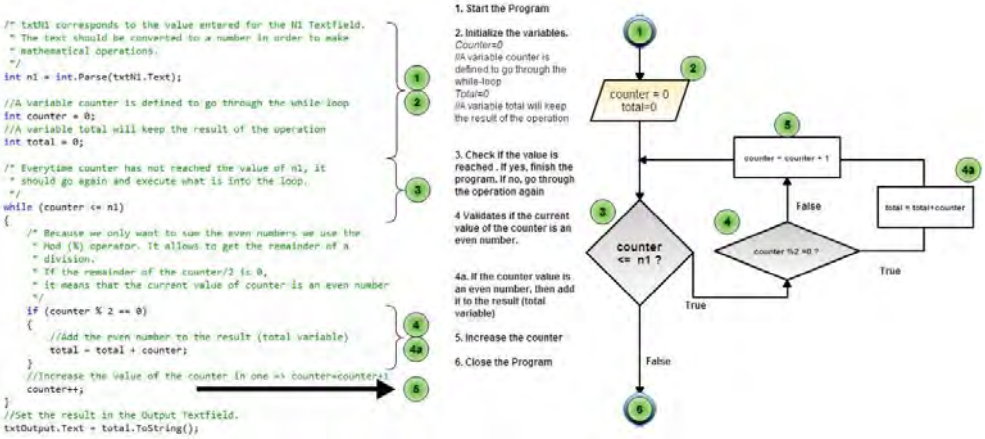


Figure 2. Distribution of a worked example including multiple representations of the solution (i.e. computational, textual, and graphical)

3.3 Data Collection

3.3.1 Learning

Pretest and posttest as well as lab assignments were employed to assess learning gains during the lab sessions. Table 3 summarizes how these instruments were prepared and implemented. The exercises for the pretest and posttest were slightly modified (i.e. changing values or sizes) to minimize the testing effect. Written comments within the code were also used as an additional source of qualitative data. Students were required to do this as a strategy to self-explain one of the provided examples.

Table 3. Description of the learning instruments employed in each laboratory session

Session	Description	Pretest and Posttest	Lab Assignment

8th	The student will create an algorithm that contains for-loop and while-loop structures to solve summations or display a list of values	Four exercises to calculate the output after certain loop structure. For example:
		`int count = 1;`
		`int sum = 0;`
		`while (count <= 10)`
		`{`
		`  sum = sum + 5;`
		`  count++;`
		`}`
		`MessageBox.Show(sum.ToString());`
			Complete the implementation for the following list of functions:
			(1) Sum of evens between 0 and a given value N
			(2) Sum of a range of numbers
			(3) Calculate the factorial of a given value N
			(4) Calculate $1 + 2^2 + 3^2 + \ldots N^2$

9th	The student will create an algorithm that initializes an array, add some values, and display the stored values	Four exercises to complete the code or write the output of certain algorithm. For example:
		`_____ mNumbers = _____;`
		`int k;`
		`int m = 1;`
		`for(k = 0;  k <= _____; k++)`
		`{`
		`  _____;`
		`  _____;`
		`}`
			Write the code to store and display numerical and textual values in an array.

10th	The student will create an algorithm to perform a sequential search and switch array elements	Four exercises to write/complete the code to find an element within an array, reverse an array or switch two values within an array. For example:
		`int [] mNum = new int[5] {3, 9, 2, 7, 6};`
		`int position = 0;`
		`for (k = 0; _____; k++)`
		`{`
		`  if (_____)`
		`  {`
		`  _____`
		`  }`
		`}`
			Write the code to complete the following methods for a given array:
			(1) Add
			(2) Find
			(3) Switch
			(4) Merge
			(5) Reverse

3.3.2 Perceptions

At the end of the lab sessions, students were given a survey where their responses were recorded using a seven-level Likert Scale with scores ranging from 0 to 6 from strongly disagree to strongly agree. The results were normalized from 0% to 100%. Values between 0% and 40% were considered negative perceptions. Values from 40% to 70% were considered undecided perceptions and values higher than 70% were considered positive perceptions.

The following questions were asked in the survey: (1) I feel I have the ability to accurately evaluate and construct a <concept>; (2) I feel I have the ability to describe a <concept>; (3) I have the ability to create a program that includes a <concept>. The three questions were posed to assess perceived ability to complete the given tasks. Two additional open-ended questions were asked to students to analyze their perceptions about the examples and the laboratory session: (1) What would you improve for the examples; and (2) What suggestions do you have for the laboratory sessions?

3.4 Data Scoring and Analysis

3.4.1 Learning

Pretest and posttest assessments were scored by two different graders to assure reliability. Whenever the graders got different scores, they discussed the scores until they agreed on a certain value. The lab assignments were scored by the teaching assistant. The comments written by the students were analyzed qualitatively to identify different categories in which the comments could fit. These categories were assigned a descriptive code that was used later to identify students' commenting styles.

3.4.2 Perceptions

Descriptive and inferential statistics were used to analyze the learning and perception measures. Whenever the data did not satisfy the normality assumption, a logarithmic transformation was used to be able to run the inferential tests. The open-ended questions were first analyzed using open-coding by one of the researchers. Then, to assure reliability, another researcher re-analyzed students' responses using his codes. The percentage of agreement was 80%. These codes were then grouped by themes.

4. RESULTS

Three iterations of data collection are reported in this section. At the end of each iteration, quantitative and qualitative results were used to improve the learning materials and the instruments for the following iteration.

4.1 Session #8: The student will create an algorithm that contains for-loop and while-loop structures to solve summations or display a list of values

4.1.1 Quantitative Data

Pretest and posttest scores were compared to evaluate learning gains. No significant differences were found for the complete group of students t (54) = -0.702, p = 0.4857 nor for the subgroups (i.e., students with/without programming experience). Table 4 depicts the descriptive statistics for the learning measures from session #8.

Differences between groups were assessed by comparing lab score and time to complete the assignment. Significant differences were only found in the "time to complete" variable, and these differences were found when comparing students who had programming experience and those who had not $F(26,1)=23.86$, $p<0.001$. however, although non-significant differences were found, students without programming experience increased their score from pretest to posttest more than those with some experience. They also received a higher lab score as compared to students with prior programming experience.

Overall, perception measures fell in the positive perception category for the ability construct (Mean = 79.49%; Standard Deviation –SD– = 16.61%). The measure was also compared between groups. Significant differences were found for the ability construct ($t(24)=3.204$, $p<0.01$) when compared by programming experience. The results suggest that students with previous programming experience (Mean = 89.90%; SD = 15.48%) perceived a higher ability to deal with loops than those without previous experience (Mean = 71.85%; SD = 13.19%).

Table 4. Descriptive statistics of student learning scores in Lab Session #8

Test		Overall (N=28)	Programming Experience	
			Yes (N=12)	No (N=16)
Pretest (%)	Mean	50.71	64.58	40.31
	SD	30.81	24.90	31.38
Posttest (%)	Mean	56.25	65.42	49.38
	SD	28.14	29.81	25.62
Lab Score (%)	Mean	95.71	95	96.25
	SD	8.36	9.05	8.06
Time to Complete (min)	Mean	86.32	68.83	99.44
	SD	22.29	17.66	15.42

4.1.2 Qualitative Data

The two open-ended questions were completed by twenty-five students. The questions were: (1) What would you improve for the examples?; and (2) What suggestions do you have for the laboratory sessions? Table 5 and Table 6 depict the results of the qualitative analysis to students' responses. A group of students suggested getting rid of some of the comments (24%) or better aligning the examples with the assignments (16%).

Table 5. Categorical analysis for the student responses to the strategies to improve examples in Lab Session #8

Theme	Code	Definition	%	Representative Quote
Students struggled with specific elements within the examples	Nothing to Improve	The student thinks that the examples are fine the way they are presented	56	"*nothing to improve*" / *"Nothing I can think of. As long as they are related to the problems and the comments are descriptive, they are fine"*
	Less Comments	The student highlights the need to get rid of some of the comments since they have an impact on the code readability	24	*"Less comments, too hard to find place among a sea of comments"/ "I feel like the comments chitter the code and makes it difficult to read"*
	Math Expression	The student feels the use of unknown mathematical expression constrains her/his understanding of the example	16	*"Explain N! - What that means?"* / *"…the '^' syntax issue was confusing for me"*
	Explicit relation example / assignment	The student requests that the examples be more detailed so that they guide the student through the problem solving process of the assignment	16	*"Better descriptions for what we are supposed to do" / "Make it so the examples demonstrate most of the common types of loops people mess up on."*
Students suggested integrating more hands-on activities as part of the classroom approach	Better with Examples	The student thinks working with examples is a better approach than working from scratch	20	"*I wanna spend more time with examples*" / "*It helped a lot but I feel like the book could've helped explain writing the math problems more in depth*"
	In-class activities	The student thinks that the class activities should be focused on practical activities (design and programming activities)	12	"*Maybe more hands on in class and allow us to program it on the computers"* / *"Make students answer questions in algorithmic form*"
	Better without	helped her/him to solve the assignment	4	*"Unsure, did not use them"*

Table 6. Categorical analysis for the student responses to the strategies to improve the laboratory sessions in Lab Session #8

Theme	Code	Definition	%	Representative Quote
Students' suggestions about laboratory sessions	No suggestion	The student thinks the laboratory sessions are fine the way they are carried out	64	*"They are going good" / "Nothing so far"*
	Logistic Improvements	The student feels that the laboratory session could be improved by either having more time, different levels of difficulty, or more teaching assistants	32	*"More TAs for more help" / "More optional assignments" / "More TA to speed things up."*
	Exploring examples	The student thinks that exploring examples would help them to better understand the concepts before starting to build a program from scratch	20	*"To continue to experiment with these types of ideas on presenting programming in an easier to understand format" / "Explore and try various examples".*
	Better without examples	The student does not consider the examples as having helped her/him to solve the assignment	12	*"I personally like the old method better" and "Keep them from Scratch".*

Table 7. Categorical analysis for the student comments within the second example in Lab Session #8

Category	Definition	%	Representative Quote
1. Detailed Comments	The student wrote a detailed description in every step of the code	21	**radSumOfNumbers is the name of the Radiobutton* **related to the sum of numbers* **it allows to identify whether the user wants to perform* **this operation when checked or when the radio button is not checked. */* */*txtN1.Text is what the user enters* ** The text should be converted into a number froms string to do mathematical operations* **/* *//txtN2.Text will correspond to n2, because that is what the user enters* *// the parse method converts the string into numbers* *in the for loop here, the variable i is defined as n1,* ** i <= n2, will make sure the loop will continue until the number reaches the* ** the number the user entered for n2* ** i++ will make sure the count will increase 1 in every loop.* *//Add the sum to the total result* *//shows the result in the output textbox*
2. Basic Comments	The student used the first example to write the comments for the second one. The comments were very simple.	32	*"*radSumofNumber is the name of the radio button* **if this radio button is checked, the loop/calculation are executed* *//declare N1 and parse* *//declare N2 and parse* *//declare the initial value for total* *//create the loop with the variables* *//calculation from the loop* *//display the calculation*
3. No Clear Comments	The student did not write any comments at all or the comments were too incomplete to be understood.	18	*// adds together all the inputed values*
4. Relevant Conditions	The student only focused on relevant sections of the code (e.g. loop conditions) with rich descriptions.	29	*// * the total is initialized to zero* *// * i equals n1 in the beginning of the code then as long as i is smaller than n2 than* *// the program will operate and it will add 1 to n1 after every time.* *// i is added to the total every time that the program is run.* *// The output is displayed through by using the tostring method.*

Regarding the laboratory sessions, students' perceptions were divided between those who preferred working with examples (20%) and those who preferred solving problems from scratch (12%). The other source of qualitative data was students' comments in the code for one of the provided examples. Four categories were identified for the commenting styles from students. The categories, descriptions, and examples are presented in Table 7. Most of the students either used the first example as a model to comment the other one with simple comments, or focused on describing the most relevant section of the code.

4.1.3 Quan + Qual

In addition to the qualitative analysis of the comments, we wanted to evaluate if there was a quantitative difference among students with different commenting styles. Table 8 shows descriptive statistics for the learning and perception measures grouped by commenting style.

Non-significant differences were found from pretest to posttest for all of these groups; however, the highest scores in both posttest and lab scores were from students with either detailed comments or those who highlighted relevant conditions with their comments. These students also spent more time completing the assignment on average compared to the rest of the students. We speculate that these non-significant difference may be due to a large standard deviation and the small sample size, which resulted from dividing the students into four groups.

Significant differences were found for the Ability Construct between the commenting styles "Basic" and "Unclear." The results suggest that students who did not write comments or who wrote unclear comments felt very confident in their abilities. On the other hand, those with basic comments may have felt unsure of their abilities; therefore, their comments were as simple as possible.

Table 8. Descriptive statistics of learning and perception scores grouped by commenting styles in Lab Session #8

Commenting Style	Pretest (%)		Posttest (%)		Lab Score (%)		Time to Complete (min)		Ability (%)	
	Mean	SD	Mean	Mean	Mean	SD	Mean	SD	Mean	SD
1. Detailed (N=5)	58.33	37.64	75	27.39	96.67	8.16	91.83	26.44	82.22	14.91
2. Basic (N=9)	38.33	33.16	45.55	30.46	95.56	8.81	77.33	18.49	67.90	18.59
3. Unclear (N=5)	55	20.92	49	26.32	92	10.95	77.20	26.37	92.22	10.83
4. Relevant (N=7)	56.25	29.12	58.75	23.87	97.5	7.07	98	16.86	83.33	10.14

4.1.4 Evaluation of the Iteration

As part of the results, two elements were called to our attention from this first iteration: (1) there were no significant differences from pretest to posttest; (2) students requested improvement of the examples by removing detail in the comments but increasing explanations.

After analyzing the results in the pretest and posttest measures, it was identified that some students were able to understand how a loop worked, but they failed to calculate the resulting value that was asked for in the test. Another identified aspect from the test was that students were struggling with mathematical expressions that are common in pseudo-code but might not be that common for them (e.g., "^" to indicate potentiation). Therefore, the following tests were more focused on building/completing code and all the potentially confusing terms were removed. Besides, the comments in the examples were organized in such a way that only the main portion of the code had a rich description of the solution.

4.2 Session #9: The student will create an algorithm that initializes an array, add some values, and display the stored values

4.2.1 Quantitative Data

During this session, the two groups were exposed to different approaches. One of the groups used examples (Experimental, N=18), while the other group used their traditional problem solving approach (Control, N=14). Table 9 shows descriptive statistics for the learning measures of these groups. The programming experience values were only calculated for the experimental group since that is the only group where these may have an impact for assessment.

Non-significant differences were found between groups or between pretest to posttest. In spite of this, it is interesting to see that students without programming experience performed better - and with a smaller standard deviation- in the lab score than students with programming experience. This follows the trend from lab session #8.

Table 9. Descriptive statistics of student learning scores in Lab Session #9

Test		Group		Programming Experience	
		Control (N=14)	Exper. (N=18)	Yes (N=10)	Yes (N=18)
Pretest (%)	Mean	63.09	55.56	59.17	51.04
	SD	29.55	36.04	37.98	35.47
Posttest (%)	Mean	67.86	63.43	70	55.21
	SD	30.29	35.14	33.38	37.78
Lab Score (%)	Mean	79.14	91.67	88.5	95.63
	SD	35.36	23.45	31.27	6.78
Time (min)					
	SD	26.65	10.70	11.69	9.78

For the ability construct, students in both control group (Mean = 83.33%; SD = 15.71%) and experimental group (Mean = 70.37%; SD = 26.61%) showed a positive perception. Non-significant differences were found between groups. For the experimental group, contrary to lab session #8, differences in ability were not

found between experienced (Mean = 76.11%; SD = 24.71%) and non-experienced (Mean = 63.19%; SD = 28.78%) programmers.

4.2.2 Qualitative Data

At the end of the session, students responded to two open-ended questions: (1) what would you improve for the examples?; and (2) what suggestions do you have for the laboratory sessions? This time only one student suggested that the examples would benefit from having still less comments while another commented: "This was much better without all the comments." In addition, more than sixty-percent of the students thought the examples were complete and useful. Table 10 and Table 11 summarize the results of the qualitative analysis to students' responses.

Regarding the suggestions for the lab session, more than sixty percent of students thought the examples were fine the way they were implemented. As in lab session #8, results suggest that there are differences regarding the preference of using examples. While there is a broad acceptance concerning the way the examples are presented and some of the students really enjoy using this scaffolding, there is another group of students who preferred building their code from scratch.

The self-explanation process of writing comments on the code was only required for the experimental group. The same categories were found for the commenting styles compared to the lab session #8. The distribution of students' comments was: detailed comments (33.33%), basic comments (33.33%), no clear comments (16.17%), and relevant conditions (16.17%).

4.2.3 Quan + Qual

Table 12 shows the results of the comparison of the learning and perception measures grouped by commenting style. The reduced sample size due to the separation between experimental group and control group makes it difficult to use inferential statistics. As in lab session #8, in lab session #9, non-significant differences were found for all of the learning measures of these groups; however, once again, the highest scores were for students who had detailed comments or highlighted relevant conditions.

Non-significant difference was found between the groups for the perception measures. We see, however, that the students without comments or with unclear comments are those who feel more confident about their ability. This result is similar to lab session #8. Students with basic comments present the lowest scores for the perception construct.

4.2.4 Evaluation of the Iteration

For lab session #9, students' suggestions about the examples changed significantly in terms of the number of comments. Still, a couple of students considered the amount of comments could be reduced. Therefore, even simpler but explanatory comments were included in the following example. In addition, students suggested adding more complexity to the examples and programming challenges. Since lab session #9 was the first one focused on the array concept, it dealt with creating and listing arrays. For the following lab session (#10) the level of difficulty was increased by dealing with swap and sequential search array operations.

Table 10. Categorical analysis for the student responses to the strategies to improve examples in Lab Session #9

Theme	Code	Definition	%	Representative Quote
Students struggled with specific elements within the examples	Nothing to Improve	The student thinks that the examples are fine the way they are presented	69	"*The examples given was perfect. I don't find any improvements needed.*" / "*Nothing, the examples were good*"
	Less Comments	The student highlights the need to get rid of some of the comments since they have an impact on the code readability	6	*"Less Comments"*
	Complexity and Quantity	The student feels that it would be better to have more and more complex examples	15	*"Maybe harder ones"* / *"Not much, just detail and complex examples would help"*
Students suggested integrating more hands-on activities as part of the classroom approach	Better with Examples	The student thinks working with examples is a better approach than working from scratch	27	"*More examples*" / *"Nothing really, already enough material to help a novice like me"*
	In-class activities	The student thinks that the class activities should be focused on practical activities (design and programming activities)	6	*"More of class time is necessary to fully understand this language"* / *"Know how to build array"*
	Better without Examples	The student does not consider the examples as having helped her/him to solve the assignment	3	*"In order to remember how to write the code, I feel we should practice writing code (not typing), i.e., the methods, etc. until we know them."*

Table 11. Categorical analysis for the student responses to the strategies to improve the laboratory sessions in Lab Session #9

Theme	Code	Definition	%	Representative Quote
Students' suggestions to laboratory sessions	No suggestion	The student thinks that the laboratory sessions are fine the way they were carried out	67	*"No suggestions" / "I enjoy these labs immensely. I have no suggestions"*
	Logistic Improvements	The student feels that the laboratory session could be improved by having more time, different levels of difficulty, or more teaching assistants.	9	*"Maybe the instructor could walk us through the code that is already provided so that we have a better understanding of what we are going into. " / "Pretest and posttest during a lab adds stress to an inherently stressful situation"*
	Exploring examples	The student believes that exploring examples would help her/him to better understand the concepts before starting to build a program from scratch	12%	*"I like the way it was taught this week and last week" / "Perhaps more code demonstrations."*
	Better without examples	The student does not consider the examples as having help	12%	*"I prefer building programs from scratch, as I understand my own code better."/ "Writing code myself is the best way to improve my skill, at least for me".*

Table 12. Descriptive statistics of learning and perception scores grouped by commenting styles in lab session #9

Commenting Style	Pretest (%)		Posttest (%)		Lab Score (%)		Time to Complete (min)		Ability (%)	
	Mean	SD	Mean	Mean	Mean	SD	Mean	SD	Mean	SD
1. Detailed (N=6)	61.11	32.35	66.67	36.89	100	0	74	8.07	72.22	30.63
2. Basic (N=6)	45.83	42.41	54.16	36.04	93.33	7.53	80.33	10.60	53.70	25.98
3. Unclear (N=3)	33.33	28.87	61.11	41.94	65	56.34	72	8.66	85.19	16.97
4. Relevant (N=3)	86.11	24.06	77.78	38.49	98.33	2.89	89	12.49	85.19	13.98

4.3 Session #10: The student will create an algorithm to perform a sequential search and switch array elements

4.3.1 Quantitative Data

For this last session, the experimental and control groups were switched after lab session #9's configuration. Thus, the experimental group became the control group (N=16), while the control group became the experimental one (N=14). Table 13 shows descriptive statistics for the learning measures of these groups. Significant differences were found between pretest and posttest measures for the non-experienced students $t(12)=-2.14$, $p=0.053$ (one tailed t-test). With an average increment of 25%, students in the experimental group showed a significant change in the posttest learning measure as compared to the pretest. The result suggests that students in the experimental condition, with no previous programming experience, took advantage of the examples to increase their understanding about sequential search in arrays.

Regarding the perception measures, students in the experimental condition showed a positive perceived ability (Mean = 80.74%; SD = 16.38%) as compared to the neutral perceived ability presented by the control group (Mean = 65.93%; SD = 20.67%).

Table 13. Descriptive statistics of student learning scores in Lab Session #10

Test		Group		Programming Experience	
		Control (N=16)	Exper. (N=14)	Yes (N=8)	Yes (N=7)
Pretest (%)	Mean	56.64	66.67	75.78	56.25
	SD	25.36	15.25	11.29	12.50
Posttest (%)	Mean	64.06	77.50	73.21	81.25
	SD	25.87	22.76	16.81	27.55
Lab Score (%)	Mean	78.13	80	86.25	72.86
	SD	40.04	35.25	35.03	36.84
Time (min)	Mean	110.31	101.40	91.50	112.71
	SD	8.55	22.03	14.91	23.25

4.3.2 Qualitative Data

The open-ended questions asked at the end of the lab session #10 were analyzed following the codes and themes found on the previous lab sessions. On this iteration, fewer suggestions

concerning changes were made. Students highlighted that "These examples were clearer than in the past." Moreover, none of the students suggested that there was a need to reduce the comments or to change the quantity/complexity of the examples. Results are summarized in Table 14.

Regarding the lab sessions, the different perspectives about the preference of using/not using worked examples continued. Four students (13%) mentioned that they wanted to continue with examples, while two students (7%) preferred working from scratch. Three students (10%) talked about logistics, such as more time for the lab sessions or more lab sessions for specific topics. Seventeen students (57%) made no suggestions.

Finally, some students seemed worn out by the research process and complained about the time the pretest and posttest took from the session: "I don't have a problem with it but these in lab quizzes take away time from the overall lab and if they are a little sooner, then they might have trouble finishing lab in time." In fact, the complexity of this lab as well as the time taken to solve the tests made this lab session the longest in terms of the time to be completed. Therefore, only four students wrote the comments in the code. The distribution of these students for commenting styles was: (1) Detailed (two students); (2) Basic (one student); (3) Unclear –no comments- (ten students); (4) Relevant (one student).

4.3.3 Quan + Qual

Since the sample size became too small in this lab session, descriptive or inferential statistics were not calculated; however, to identify whether the trend that came from lab sessions #8 and #9 continued, the lab score for the three students in the detailed and relevant commenting styles were checked. All three students got a score of 100% , thereby confirming the trend.

5. DISCUSSION

This study explored the use of worked examples to support programming activities as part of an introductory course. Specifically, this study explored two questions and findings are discussed below.

5.1 How can worked examples be effectively designed to introduce programming concepts to novice learners?

Three laboratory sessions were used to introduce programming concepts using worked examples. The design and implementation of the worked examples were iteratively improved using students' suggestions and validated through learning assessments. The structure of the examples followed the principles suggested by Atkinson [5] that included: a problem statement, a procedure for solving the problem, and auxiliary representations of the problem and solution.

Two examples were used to scaffold the learning process in each session. The problem statement consisted of a single programming task aligned to the learning objective of the lab session and embedded within the problem set. The solution was represented in multiple forms including textual, graphical, and computational representations. All the representations were aligned with each other. A self-explanation task was also included as part of the assignment using written comments within the code to engage the students in the process.

The feedback from open-ended questions was useful for improving the examples. The main component of these edits was the elimination of complex explanations within the code that could generate additional cognitive load to students. In fact, the examples with the simplest comments (lab session #10) were the ones that showed significant differences. Some other changes were included such as: (1) avoiding the use of complex mathematical symbols; (2) increasing the complexity of the examples; and (3) aligning them to the problem assignments.

Only the last laboratory session (#10) presented significant differences in learning gains for students with non-programming experience. This result is aligned to what is suggested by Atkinson et al. [5] in that the worked examples approach may be useful for novices in an initial skill-acquisition stage such as analogy or abstract rules of learning [4].

Table 14. Categorical analysis for the student responses to the strategies to improve examples Lab Session #9

Theme	Code	Definition	%	Representative Quote
Students struggled with specific elements within the examples	Nothing to Improve	The student thinks that the examples are fine the way they are presented	61	*"They seem fine"/"Examples are fine"*
	More Detail	The student suggests increasing the level of detail in the examples or exercises.	6	*"More descriptions on how to reverse the array" / "Describe in more detail what the questions asking"*
Students suggested integrating more hands-on activities as part of the classroom approach	Better with Examples	The student thinks working with examples is a better approach than working from scratch	27	"*More examples*" / *"Nothing really, already enough material to help a novice like me"*
	In-class activities	The student thinks that the class activities should be focused on practical activities (design and programming activities)	6	*"More of class time is necessary to fully understand this language" / "Know how to build array"*
	Better without Examples	The student does not consider the examples as having helped her/him to solve the assignment	3	*"In order to remember how to write the code, I feel we should practice writing code (not typing), i.e., the methods, etc. until we know them."*

On the other hand, expert students with prior programming experience did not benefit from the examples, perhaps because they may have already developed a mental model [19]. For the rest of the sessions (#8 and #9), we speculate that the examples were unclear because they had too many comments included within the code. After students' suggestions, the examples were refined with simpler comments. Another possible explanation can be related to the time students need to get used to this new pedagogical approach. The worked examples approach was only introduced starting on lab session #8. Hence, the students were already used to a different problem solving approach. Moura [13] experienced this phenomenon and highlighted that students needed some time to get used to the tool she used for the worked examples. After that time, students performed better. Finally, the small sample size also made it difficult to find significant differences.

Regarding the perception constructs, novice students perceived their ability to solve various computing-related tasks to be significantly higher than those students with programming experience (in lab session #8). This, however, changed over time and a non-significant difference was found between experienced and non-experienced programmers for the rest of the iterations. The result suggests that, as the examples were improved, students with no previous experience were better able to take advantage of them. This is also suggested by the perceived ability of the students from the experimental group in the last session (80.74%), which was higher than the control group.

The worked examples approach generated a separation between those students who enjoyed exploring and learning from them and those who preferred to build the whole program themselves. From the students' responses from any of the sessions, of those who mentioned that they preferred coding from scratch, 75% identified themselves as experienced programmers. This is aligned with the rest of the findings and the literature suggesting that worked examples are more useful for novice learners than for expert ones [4, 5].

5.2 How do students self-explain worked examples when approaching a solution to a programming assignment?

Commenting on the code was used to encourage students' self-explaining process for the examples. These comments were grouped as four commenting styles: (1) Detailed; (2) Basic; (3) Unclear; and (4) Relevant (see Table 8 for a full description). Although non-significant differences were found between the groups, valuable insights were identified. First, as suggested by Chi et.al. (1989), students with a deeper self-explaining process (either (1) Detailed or (4) Relevant) performed better in all the lab sessions. Students with an incomplete self-explanation process appear to not fully understand the problem solving approach and, therefore, are unable to solve similar problems by analogy. Chi and collaborators [6] called this effect the self-explanation effect and enumerated four differences between students who were able to take better advantage of the examples than students who passively explored the examples. Trends identified in [6] were (1) high performers presented more self-explanations while studying examples; (2) "Poor" performers did not perform enough self-monitoring activities such as "I can see now how they did it"; (3) High performers referenced less to the examples when solving another problem than "poor" performers; (4) The "poor" performers self-explained more during the problem solving than the high performers who preferred to do it during the example exploration.

The second insight is that students who did not include any comments reported a higher perceived positive ability than those students who wrote very simple comments. We speculate that these students felt confident about their abilities and, therefore, did not want to spend time understanding another approach; however, they did not perform as well as students who wrote thorough comments.

The main limitation of the study is the small sample size constrained by the course size. Therefore, the significance of the differences found in this study lies in the qualitative data regarding students' recommendations, perceptions, and commenting styles. Another limitation is that the worked examples approach began in lab session #8. This means that the students had been exposed to seven previous sessions with a different approach. This may have generated a negative reaction in some students who preferred to work in a more familiar way.

6. IMPLICATIONS

6.1 Implications for Teaching

The use of Atkinson's instructional principles to design worked examples has been identified as useful in situations where novice learners seem to take more advantage of this technique. Expert learners may have already acquired mental models in the thematic area that provided them with the necessary tools for problem solving.

The identification of intra-example, inter-example, and interacting-with-the-learning-environment features of worked examples can provide a framework for instructors to effectively design their worked examples. Specifically, the intra-example features used in this study presented several requests by the students to keep simple explanations, especially when they are integrated into the code. Students often mentioned that many comments within the example code decreased readability. Thus, the use of at least two different examples with a good alignment with the assignments is the main inter-example feature that should be considered.

Finally, for programming activities, requiring students to write comments within the code can be useful as a self-explanation process; however, to take full advantage of this process, it is important to encourage students to write detailed comments or to highlight relevant conditions by describing boundaries and the consequences of their solutions.

6.2 Implications for Learning

Results from this study suggest that students who described relevant conditions along the code, as well as details in the way the code worked, performed better than those students who commented on the code superficially or did not self-explained it at all. Several studies have demonstrated that a passive approach to studying worked examples has no impact on learning as compared to problem-solving instruction (Chi et. al., 1989; Atkinson et al., 2000). The reason for this could be a lack of understanding resulting from not actively engaging with the examples.

Chi and colleagues [6] suggested that the examples are not always completely clear, so the students have to engage in a self-explanation process allowing them to identify the relevant aspects of the solution. Thus, a self-explanation should contain four aspects that depict an understanding: (1) the conditions of

application of the actions; (2) the consequences of actions; (3) the relationship of actions to goals; and (4) the relationship of goals and actions to natural laws and other principles. In this study, the "Detailed" and "Relevant" commenting styles contained all these characteristics while "Basic" or "Unclear" commenting styles contained only one of these features, (e.g. the conditions of application of the actions) if any of them at all. Furthermore, a good understanding of the example can lead to more proficient problem-solving skills, while poor understanding may lead to a continuous reference to the example while trying to solve another problem.

7. CONCLUSION

The use of worked examples to scaffold programming and algorithm design learning has been evaluated. Different instructional design elements were assessed in order to identify effective design characteristics for worked examples. Multiple representations of the solution, including textual, graphical and computational representations, were employed. Writing in-code explanations as simple sentences enhanced code readability and improved students' perceptions about the examples. Moreover, encouraging students' self-explanation process by asking them to comment within the code helped the students to actively engage with the examples. Specific suggestions include encouraging students to write detailed comments as opposed to superficial ones in order to take advantage of the examples. This approach seems to be useful for novice students who did not have previous experience in programming.

The contribution of the study is the detailed description of the implementation of worked examples in a programming context. It includes the use of multiple representations as well as the use of comments within the code as a self-explanation process.

8. LIMITATIONS AND FUTURE WORK

The main limitation of this study is that the learning outcomes for each iteration were different. Thus, the changes implemented based on the results were not evaluated in exactly the same context. Therefore, future work will explore the effect of these recommendations for these three lab sessions.

Next steps also include the design of additional examples using instructional principles of worked examples [5] as well as students' suggestions in this process. Future instruction should also encourage students to carry out a thorough self-explaining process that may lead them to an understanding of the examples. This can be accomplished either through incentives or by means of extended training.

9. ACKNOWLEDGMENTS

This research was supported in part by the U.S. National Science Foundation under the award #EEC1329262.

Authors would also like to thank Guity Ravai for her assistance in reviewing the assessment instruments and facilitating the implementation of the learning materials.

10. REFERENCES

[1] [NRC]. 1999. Being fluent with Information Technology: National Academy Press.

[2] [PITAC]. 2005. "Computational science: ensuring America's competitiveness," President's Information Technology Advisory Committee (PITAC), vol. 27

[3] [WTEC]. 2009. "International assessment of research and development in simulation-based engineering and science" World Technology Evaluation Center, Inc., Baltimore, Maryland.

[4] Anderson, J. R., Fincham, J. M., and Douglass, S. 1997. The role of examples and rules in the acquisition of a cognitive skill. Journal of Experimental Psychology: Learning, Memory, and Cognition, 23, 932-945.

[5] Atkinson, R.K. Derry, S. J., Renkl, A., and Wortham, D. 2000. Learning from Examples: Instructional Principles from the Worked Examples Research/ Review of Educational Research. Summer 2000, Vol. 70, No. 2, pp. 181-214.

[6] Chi, M. T. H., Bassok, M., Lewis, M. W., Reimann, P., and Glaser, R. 1989. Self- explanations: How students study and use examples in learning to solve problems. Cognitive Science, 13, 145-182.

[7] Cuny, J., Snyder, L., and Wing, J. M. 2010. Demystifying Computational Thinking for Non-Computer Scientists. Work in progress.

[8] College Board. 2011. CS Principles. http://www.collegeboard.com/prod_downloads/computerscience/Learning_CSPrinciples.pdf

[9] du Boulay, B. 1989. Some difficulties of learning to program. In E. Soloway & J.C. Spohrer (Eds.), (pp. 283–299). Hillsdale, NJ: Lawrence Erlbaum.

[10] Gray, S., St. Clair, C., James, R., and Mead, J. 2007. Suggestions for graduated exposure to programming concepts using fading worked examples. Proceedings of the third international workshop on Computing education research - ICER '07, 99.

[11] Guzdial, M. and Robertson, J. 2010. Too much programming too soon? . Communications of the ACM, Volume 53 Issue 3, March, 2010.

[12] Kirschner, P. A., Sweller, J., and Clark, R.E. 2006. Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching, Educational Psychologist, Vol. 41, Iss. 2.

[13] Moura, I. C. 2013. Visualizing the Execution of Programming Worked-out Examples with Portugol. Proceedings of the World Congress on Engineering 2013 Vol I, WCE 2013, July 3 - 5, 2013, London, U.K.

[14] Paas, F., Renkl, A., and Sweller, J. 2004. Cognitive load theory: Instructional implications of the interaction between information structures and cognitive architecture. Instructional science 32, 1-8.

[15] Paas, F., Renkl, A., and Sweller, J. 2003. Cognitive Load Theory and Instructional Design: Recent Developments. Educational Psychologist, 38(1), 1–4.

[16] Pirolli, P. and Recker, M. 1994. Learning strategies and transfer in the domain of programming. Cognition and Instruction, 12, 235–275.

[17] Renkl, A., Atkinson, R. K., and Große, C. S. 2004. How Fading Worked Solution Steps Works – A Cognitive Load Perspective. Instructional Science, 59–82.

[23] Teddlie, C., and Tashakkori, A. 2006. A general typology of research designs featuring mixed methods. Research in the Schools, 13(1), 12-28.

[24] Wing, J. 2006. Computational Thinking. Communications of the ACM, 49, 3, 33-35

Picky: A New Introductory Programming Language

Francisco J. Ballesteros
Universidad Rey Juan Carlos
C/ Camino del Molino SN
E29843, Fuenlabrada, Madrid, Spain
nemo@lsub.org

Gorka Guardiola Múzquiz
Universidad Rey Juan Carlos
C/ Camino del Molino SN
E29843, Fuenlabrada, Madrid, Spain
paurea@lsub.org

Enrique Soriano Salvador
Universidad Rey Juan Carlos
C/ Camino del Molino SN
E29843, Fuenlabrada, Madrid, Spain
esoriano@lsub.org

ABSTRACT

In the authors' experience the languages available for teaching introductory computer programming courses are lacking. In practice, they violate some of the fundamentals taught in an introductory course. This is often the case, for example, with I/O. Picky is a new open source programming language created specifically for education that enables the students to program according to the principles laid down in class. It solves a number of issues the authors had to face while teaching introductory courses for several years in other languages. The language is small, simple and very strict regarding what is a legal program. It has a terse syntax and it is strongly typed and very restrictive. Both the compiler and the runtime include extra checks to provide safety features. The compiler generates byte-code for compatibility and the programming tools are freely available for Linux, MacOSX, Plan 9 from Bell Labs and Windows. This paper describes the language and discusses the motivation to implement it and its main educational features.

Categories and Subject Descriptors

D.3.0 [**Programming Languages**]: General
; D.3.3 [**Programming Languages**]: Language Constructs and Features
; K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms

Programming Languages, CS1

Keywords

Programming Languages, CS1

1. INTRODUCTION

The authors are in charge of teaching an introductory computer science course (CS1 from now on). The curriculum is focused on imperative, statically typed procedural programming. Nevertheless, it is not the usual imperative programming curriculum. It strongly emphasizes the *top-down* approach and the definition of subprograms. Proof of this is that the course starts as a functional programming course. The students learn how to build expressions and functions before learning how to declare variables and build sequences of statements. However, only one (imperative) programming language is used for the whole course. Our approach is similar in spirit to [8], but with a different implementation. Where Decker et al focus on object organization, we focus on the strategy for attacking the problem by breaking it into subproblems.

The course follows a twofold pedagogy methodology. First, at every point the student is required to write code to test her understanding of the matter at hand. Second, every line of code the student writes must be comprehensible at that point of the course. Of course, the second part needs to be relaxed somewhat at the start of the course, but it is an important principle we adhere to, whenever possible.

After teaching this course several years using Ada as the main language, the authors decided to look for an alternative for several reasons. Ada, despite being a Pascal descendant, is multiparadigm. Moreover, its syntax is too verbose, and it has other issues that are discussed next.

Selecting a new language for CS1 is a complex and delicate task. The related bibliography is extensive [18] and there are many open discussions about the different approaches, for instance [14, 22, 21, 15, 5]. Although other authors defend the use of object oriented languages for introductory courses (see for example [17]), there is no consensus about which approach is better (*objects early* vs *procedures early*) [22, 21, 15, 5]. In the authors' experience, object oriented languages are too complex to be used as a first language. As other authors state [12], the student should be instructed before delving into an understanding of object oriented programming concepts, which are more abstract (inheritance, delegation, polymorphism, etc.) than other basic prerequisites (variables, parameters). Object oriented languages may be popular, but they are not simple enough to be understandable for a primer and look like magic to most novice students.

After some research, the authors were not satisfied with the existing alternatives to replace Ada. Although there are

many programming languages available, and some of them are specifically created for education, none satisfied all our needs, described in section 2. In the end, the authors decided to design and implement Picky and write a text book (in Spanish) for the course [3]. Picky is a new imperative programming language that meets all these requirements. This paper presents the main features of the language for teaching the CS1 course and the experience after using it for nine CS1 courses with more than six hundred students.

2. REQUIREMENTS

2.1 High level

It is widely accepted [16] that low level languages, such as C, are not suitable for CS1 courses. Even the defenders of such languages acknowledge their shortcomings [20].

Thus, the candidate language to replace Ada for this purpose must completely abstract the details of the machine and the underlying operating system.

2.2 Single-paradigm

Some prestigious institutions, for example CalTech and MIT, use multi-paradigm languages in introductory courses [14]. As stated before, some of the issues that the authors found while teaching CS1 in Ada are related to its multi-paradigm features.

Although multi-paradigm languages can be suitable for long courses that start with imperative programming and then continue with object oriented programming, they are not suitable for single-paradigm courses (i.e. imperative programming). It is confusing for students to consult bibliography that mixes the paradigms or references focused on a paradigm that is out of the scope of the course.

Also, as part of the learning process, the student, by mistake, may write programs that wander off the subset taught in class. When the language has many heterogeneous constructions, like Ada, it is highly probable that the student may come across one of them by mistake. Another issue is the compiler returning an error related to one of these off-course constructions. Frustration and confusion ensues.

Using a pure object oriented language (e.g. Java) to teach imperative programming, like some institutions do, is even worse. For a significant part of the course, the student gets used to writing code which is incomprehensible at that point in time (public, static, class, etc.). This violates one of the cores of the twofold approach detailed above.

2.3 Restrictive

The candidate language must provide strong typing and range checking. These features are very convenient when learning how to program for the first time. With them, the compiler and the runtime act as a safety net which prevents the students from wandering off too much. This is another reason for not using languages such as C, where the plasticity of the language makes it easy to write obscure code. In addition, it is desirable to use a language that includes extra restrictions. For example, global variables are very harmful in an introductory course and it is convenient to use a language that forbids them. This forces the student to get into the habit of structuring the code properly.

Some kinds of syntactic sugar and language features make it unclear for students what the code actually does. They also make difficult to consolidate some important concepts, such as data typing. For example, transparent dereferencing of pointers in Ada prevents students from understanding the difference between a record and pointer to a record. Another example is automatic declaration of variables (i.e. dynamic typing) in Python. The lack of variable declaration complicates the comprehension and identification of data types and variables. Furthermore, it also muddles the concept of static scoping.

While all these features may enhance the expressiveness of a language later on, the basic concepts need to be established first in the mind of the student.

2.4 Terse and simple syntax

In the authors' opinion, the perfect candidate is a language as simple as Pascal (or even simpler), with terse syntax like C.

Pascal has been widely recognized as a good language for CS1 courses. However, its control syntax is too verbose. Also, the use of brackets and parenthesis in constructions emphasizes the formal character of the language, one source of confusion for new programmers.

In addition, Pascal syntax is more complex than needed. For example, the use of semicolons as separators instead of terminators for sentences is a problem for students. They end up guessing when to add a semicolon and when not to add one.

There is also a practical problem with Pascal. It is difficult to find an implementation of Pascal which works well in all the operating systems the students may use at home and the lab.

Ada is quite verbose and utterly complex. This makes things hard for students in introductory courses, because there are many different constructs to master and the possibility of wandering off by mistake, as explained before. Also, control structures requiring *exit when* constructs are easily misused. At the same time, this construction cannot be forbidden because it is necessary for do-while (in fact do-until) loops.

Using white space characters and tabs as part of the syntax is a double-edged sword. On one hand, it is useful to force a valid indentation (e.g. Python). On the other hand, it leads to syntax errors that are hard to solve for a first course student. For example, mixing white space characters and tabs in the same program causes errors, and it is difficult to locate them manually. Even worse, the correctness of the program depends on the text editor. Some editors hide white space characters or translate tabs to them or vice versa. A common pitfall when programming in Python is to use two different editors to write the same program (e.g., the editor installed in the laboratory and the editor installed in your personal computer). The authors consider that, in general, using these characters as part of the syntax is not desirable

in a introductory programming language.

2.5 Explicit management and debug facilities

One of the aims of a first programming course is to teach students how to debug programs.

To make memory allocation errors explicit and introduce the concept of dynamic memory (de)allocation, the language must support manual memory deallocation instead of automatic garbage collection.

In addition, it must be easy to detect dynamic memory failures and leaks. It would be also desirable to be able to inspect the program stack in a novice-friendly format without using complex tools (e.g. gdb).

2.6 Text based

The language must be suitable for a first year University course. There are several visual programming languages for education at different levels [19, 4, 11, 6, 7]. Nevertheless, the authors need a classic text-based language, closer to real world programming languages, to ease the way into the other languages taught later in the curriculum (C, Ada, Java, Python, etc.).

2.7 Editor/IDE independent

Another important requisite is that the language must be independent of IDEs. Some authors are especially critical of commercially available IDEs [11]. For the authors of Picky, it is a must to be able to compile and execute programs in the command line and from shell scripts.

In some IDEs (like Eclipse), it is very difficult for a novice to understand when a program is being compiled, when it is being ran and what version is being used. The authors experience in more advance courses on Java and Android using Eclipse is that the facilities provided for managing projects cause problems even to last year students. For example, it is quite complex for them to export and import a project, even on machines running the same operating system.

In addition, it is paramount to allow the expert users (i.e. the teachers) to select the text editor of their choice so that the class can be taught fluidly. It is very common for the authors to program something on demand as part of an explanation. If the environment is cumbersome, the students will get bored and distracted.

While syntax highlighting (or any other feature that make plain text look like formatted text) may be useful for more advanced courses, it is utterly harmful for novices. On one hand, the students are told that the compiler only accepts source code in plain text and that plain text does not have any format, it is just a sequence of character codes. On the other hand, the IDE or editor magically shows bold and italic colored fonts. The authors want to avoid this kind of *magical effects* to improve the comprehension of the tools.

2.8 Realistic I/O managing preserving referential transparency

File I/O is important not just to perform I/O, but also to teach the students how to use control structures to guide data consumption without violating file I/O rules imposed by the file abstraction.

File handling in Ada is clumsy, to put it mildly. Calling *End_Of_File* may block a program when reading from the terminal, and students will not know why. Furthermore, we teach that functions should be *referentially transparent*. Nevertheless, many Ada file I/O subprograms (that is, non-deterministic subprograms) are functions, not procedures. This violates the referential transparency.

2.9 Portable

In order to study at home and complete the assignments, students must be able to use at home the same tools that they use at the laboratory. The tools must be available for the systems they use, namely, Windows, Linux and Mac OS X. Of course, the tools must be easy to install for all these systems.

In addition, the executable files generated by the students should also be portable. The first option is to use an interpreted language. Nevertheless, interpreted languages make it difficult to consolidate concepts like compiling, linking, and executing. For the student, it is hard to distinguish between source code files and executable files. Another option is to use a compiled language that generates machine independent code to run on a virtual machine. In this case, it is also difficult to distinguish between the virtual machine and the compiled program.

The solution we have taken is to follow the latter approach and keep the illusion that the compiler generates an actual native binary file that can be executed in the system like a native executable (i.e. without invoking another program like the virtual machine).

2.10 Open source

Last, the authors need to be able to modify the tools if necessary. Thus, the language selected for the course must have open source tools available for all the systems enumerated in the previous point.

3. PICKY IS REALLY PICKY

Before providing a description of the language, we would like to summarize its main features regarding safety. As the name of the languages suggest, Picky is very restrictive. The aim is to forbid students any practice that can be harmful if it becomes a habit.

When a kid learns how to ride a bicycle it is convenient to use side-wheels for a while. Only after such artifact is under control, a new bicycle (one without side-wheels, and perhaps with an engine) is more convenient. In the same way, Picky is highly restrictive regarding what can be done and what can not in a program. It has side-wheels attached.

Apart from the desired features described before (strong typing, avoidance of *automatic* features such as dynamic declaration or automatic deferences of pointers, no global variables, and so on), both the compiler and the run time include extra checks and waste memory and time to provide additional safety features.

- If the student forgets to initialize a variable, it will not be zeroed. Moreover, the variable will not have the corresponding value left in the stack by a previous activation register.

  In Picky, all variables are implicitly initialized with a random value. Thus, if there are uninitialized variables, every execution will be different.

- The compiler does not provide *warnings*. Any error is a fatal error and the program does not compile.
- The runtime tracks dynamic memory usage and provides informative diagnostics regarding accidental use of dangling pointers.
- A program fails if there are dynamic memory leaks, i.e. if there is memory allocated and not freed before the program terminates.
- Functions do not accept parameters passed by reference.
- It is required that *return* is the last statement in the function body.
- Procedures cannot use *return*.

In addition, some constructions are forbidden. For example, the authors have detected that the following erroneous construction is very common:

```
if(condition){
       dosomething();
}else{
       ;
}
```

For the above construction, the Picky compiler gives a compilation error. This code should be rewritten as:

```
if(condition){
       dosomething();
}
```

4. THE LANGUAGE

The language is very simple. To get a full description of the language, see [2]. What follows is a discussion of the most relevant details from a pedagogical point of view, following the requirements stated in section 2. There are further pedagogical omitted here for the sake of brevity.

4.1 Programs

Picky has control structures reminiscent of C and data declarations in the style of Pascal. A source program is made of a single file. A simple hello world example:

```
1 /* Hello world  */
2
3 program Hello;
4
5 procedure main()
6 {
7       writeln("hello, world");
8 }
```

Comment syntax is taken from C. A program is introduced by a *program* clause (line 3) that assigns an identifier to the program. A procedure named *main* must be included, like in C. The program starts executing its body and terminates when returning from it.

All declarations and statements are terminated by a semicolon, but note that procedure and function definitions are not terminated by a semicolon. Constants, types, procedures, and functions may not be declared within the scope of a procedure or function. That is, subprograms may not be nested and constants and types must be declared in the global scope.

The language is case-sensitive. An identifier must start with an alphabetic character followed by zero or more alphanumeric characters. Picky only has 26 keywords and a total of 81 language defined names, including keywords, builtins and predefined constants.

A program may also include one or more constant declaration blocks, one or more type declaration blocks, one or more variable declaration blocks, and procedure and function definitions. The scope for a declaration goes from the point where it happens in the source to the end of file. Global variable declaration sections are forbidden by the compiler unless a flag is supplied.

Constant, type, and variable declaration blocks start with the keyword *consts*, *types*, and *vars* (respectively) followed by declarations. The following program is an small, correct, albeit useless, example:

```
1  program Xample;
2
3  consts:
4        Npts  = 11;
5        Greet = "hi";
6
7  types:
8        Tmonth = (Jan, Feb, Mar);
9        Tpt = record{
10             x: int;
11             y: int;
12       };
13       Tpts = array[0..Npts-1] of Tpt;
14
15 consts:
16       Zmonth = Jan;
17
18 vars:
19       a: month;
20
21 procedure incptx(ref pt: Tpt)
22 {
23       pt.x = pt.x + 1;
24 }
25
26 function addpty(p1: Tpt, p2: Tpt): Tpt
27 {
28       p1.y = p1.y + p2.y;
29       return p1;
30 }
31
32 procedure main()
33         pts: Tpts;
34         i: int;
35 {
```

```
36       for(i = 0, i < Npts){
37             pts[i] = Tpt(2, 4);
38             incptx(pts[i]);
39             pts[i] = addpty(pts[i], pts[0]);
40       }
41       writeln(pts[Npts-1].x);
42       writeln(Greet);
43 }
```

4.2 Basic data types

Per requirement 2.3, Picky is strongly typed. The basic types are *bool*, *char*, *int*, *float*, and *file*. They correspond to booleans, characters, integers, real numbers in floating point, and external (text) files.

Two types are compatible (for assignment and other operators) only if they have the same name. Predefined types also obey the same rule. Constants and literals are an exception, they belong to *universal* types that are assumed to be compatible with any basic data type of the same kind. This is reasonable, for example, to permit using integer literals in expressions that belong to a user defined integer type. Another exception are subranges. Subranges do not introduce a new type; they declare a restriction defining a subset of an existing type.

A type definition defines a new type and declares its name. For example:

```
types:
      Apples = int;
      Oranges = int;
```

This code defines two new types: *Apples* and *Oranges*. It is not legal to mix apples with oranges, and it is not legal to mix any of them with *int* values. However, integer constants and literals may be mixed with any of them.

In general, the language does not permit type casts. However, type casts are permitted to convert ordinals to the integer representing their position in the type and vice versa. Also, integers may be converted to floating point numbers and vice versa.

To convert a value to a type use the target type name as a function. For example, these are legal expressions:

```
char(int('A') + 1)
float(3)
int(4.2)
```

4.3 Explicit dynamic memory and resource management

Resources in Picky are managed explicitly as stated in section 2.5. Memory allocations and deallocations are explicit and there is no garbage collection.

A pointer data type refers to another type and permits using *new* and *dispose* to handle dynamic variables of the pointed-to type. Type definition uses the ^ notation, taken from Pascal:

```
types:
      Arry = array[1..10] of int;
      Iptr =  ^int;
      Aptr = ^Arry;
```

The second line declares an array data type used in the last line, to declare a *pointer to Array* data type. The third line declares a pointer to integer. It is legal to declare a pointer to a type that is not yet defined in the program, but the target type must de defined later. This permits declaring circular data types, like linked lists. In no other case may a type be defined in terms of not yet defined types.

Syntax to dereference a pointer value is also taken from Pascal, and also uses the ^ sign:

```
iptr^ = 2;
aptr^[1] = iptr^;
```

L-values of pointer types may use the following procedures to allocate and deallocate memory: $new(ptr)$ (set ptr to point to newly allocated memory) and $dispose(ptr)$ (frees the memory referenced by ptr). All memory allocated with *new* must be released by calling *dispose* before completion of the program, or the program will abort and report memory leaks. The interpreter makes sure that dereferencing a dangling pointer (i.e. a pointer pointing to freed memory) will abort the execution, providing the corresponding error to the user.

File descriptors are also managed explicitly. Files need to be opened and closed using the appropriate builtins, $open(file)$ and $close(file)$. Any error related to accessing a file is fatal for the program.

4.4 Input/Output

Some languages use I/O primitives that are predictable but too low-level. Others provide high-level, but unpredictable facilities. Among other things, it is impossible, in general, to know if there is an *end of file* before trying to read. On the other hand, it is not reasonable to read without checking the *end of file* condition.

As we explain to our students in the CS1 course, when programming, side effects must be contained. Checking for the *end of file* should be a function without side effects. The *read* operation should be a procedure with side effects.

In Picky the I/O primitives follow the requirements stated in section 2.8. They are both practical and clean from a theoretical point of view. A *peek* procedure scans the input to check for *end of file* or *end of line* conditions. Part of the *peek* specification is that it may read internally from the file. The *eof* operation is a function and has no side-effects (i.e. it never reads). Before any attempt to call *read* or *peek*, *eof* returns false as it should.

The language forbids to read *end of line* marks, they must be skipped. The runtime includes checks to trigger errors if a program tries to read them directly instead of using a *readeol* primitive.

4.5 Debugging facilities

Following the requirement in section 2.5, built-in procedures are provided for user friendly debugging, and abnormal termination: *fatal(text)* (print text and abort execution), *stack()* (dump the stack in a friendly format for debugging) and *data()* (dump global data in a friendly format for debugging). For example:

```
stack trace at:
dowork() pid 0 pc 0x000008 xample.p:9
arguments:
x = 3
local variables:
z = 8

called from:
main() pid 1 pc 0x000016 xample.p:16
local variables:
x = 3
```

In other development environments, students tend to debug by using step-by-step execution on debuggers instead of thinking. In this language, it is natural for them to dump the program state and think about the cause of their problems. Later, when they are less prone to misuse them, they will learn more advanced debugging techniques such as step-by-step execution, breakpoints, etc.

4.6 Procedures and functions

There is a clear separation in Picky between procedures and functions to follow the principle described in 2.8. The principle is that functions should have no lateral effects and should preserve referential transparency. This principle is also followed by builtin functions and procedures. Procedures are named actions, so can have lateral effects, and do not return values. Argument passing is by value (by default) or by reference (using the keyword *ref* before an argument name). See lines 21-24 in the *Xample* program. Functions are declared similarly, see lines 26-30 in the program.

4.7 Global and local variables

Picky does not permit global variables by default. They can be enabled with a compiler flag. The flag is in place so that the concept of global variables can be explained in the corresponding class.

It is not allowed to declare a type on the fly in the variable declaration, unlike in Pascal. A type identifier is required after the colon. This forces the students to define types first and assign them meaningful names before using them.

Variables are initialized to random values. This feature makes programs fail when using uninitialized variables instead of making them work intermittently. Therefore, students learn quickly that uninitialized variables are dangerous.

4.8 Control structures

Picky has the usual control structures. The *if*, *while*, *do-while*, and *switch* statements borrow their syntax from C and semantics from Pascal (there is no *break*). Statements used for *then* and *else* arms must always be blocks. Students face no *dangling else* in Picky.

The *for* loop (see lines 36-40 in *Xample*) has a header with only two expressions, an initialization and a condition. The initialization must be an assignment for a variable of an ordinal type. The condition must use any of these operators: "<", "<=", ">" or ">=". The first two ones make the variable increase automatically after each iteration. The last two make the variable decrease automatically after each iteration.

After the *for* loop, the control variable is equal to the value on the right of the condition. This implies that there is no out of range condition for the control variable even when using "<=" or ">=" with the first or last valid value of an ordinal type. In *Xample*, *i* value is *Npts* when the loop is done.

The only way to exit a loop is to satisfy the condition of the loop; there is no *break* or *goto* statement. This way the postconditions are clear and the student is forced to structure the program.

5. COMPILATION AND EXECUTION

The Picky compiler, *pick*, is implemented in C. The compiler is implemented using *yacc* [13] and should be easy to understand.

The compiler does not emit warnings. All diagnostics correspond to compile time errors. In many cases, when an error is detected, a symbol or node in the syntax tree is still built, for safety; other parts of the compiler still get a data structure as expected, and it's less likely that an invalid value causes a bug.

Picky compiles to a virtual machine (PAM [2]), invoked transparently by the compiled output file. Thus, students are not surprised by "binaries" behaving differently on different platforms. Code generation is straightforward. The machine is stack based. Most operations take arguments from the stack and replace them with a result, pushed also on the stack. There is a single flow of control, guided by a loop switching on the instruction type.

PAM wastes memory and time to detect mistakes like out of range conditions, the use of already disposed data structures, etc. This way, it issues very descriptive diagnostics and not just "*segmentation violation*".

As already stated, variables (from the data, stack or heap) are initialized with random values, to let the user discover early that variable initialization is missing. Such random values are always odd, to recognize uninitialized pointer values and issue a descriptive diagnostic for that case at run time, instead of a *segmentation violation* or producing a heisenbug.

The abstract machine construction makes it possible to dump the state at any point in a user friendly format. The *stack* and *data* builtins (explained in section 4.5) rely on this feature.

Picky "binaries" are just text files that are interpreted by PAM. They start with the Unix *hash bang* syntax to call PAM on their own. In Windows, to the same end, the file

extension *pam* is associated in the registry to the application *pam* as part of the installation. Thus, students have "binary" files that, at the same time, are portable and can be used for pedagogical purposes. Students compile and then run the resulting file:

```
prompt$  pick hello.p
prompt$  out.pam
 hello picky!
prompt$
```

The "binary" generated includes portions of the source code in comments, and can be used during lectures to teach how the code written by students maps to machine instructions:

```
#!/bin/pam
entry 0
...
# x: int = 3
0000a push 0x00000003 # 3;
0000c lvar 0x00000000 # x;
0000e sto 0x00000002
# dowork(x: int)
00010 lvar 0x00000000 # x;
00012 ind 0x00000004
00014 call 0x00000000 # dowork();
```

This way students do not perceive the machine as a *magical device*.

6. EXPERIENCE

The authors are quite happy with the results of using Picky in CS1 courses. They have used the language to teach nine CS1 courses that are part of three different degrees of the Telecommunications Engineering School of the Rey Juan Carlos University of Madrid. The number of students that have actually used the language is greater than six hundred. The first generation of students that used Picky for CS1 is currently programming in Java, C, Ada, Python and shell scripting in 3rd-year courses.

It is difficult to evaluate fairly and accurately the effectiveness of the language for teaching CS1 courses. Since the authors are in charge of teaching and evaluating the students, any evidence related to grades of tests and assignments could be unintentionally biased. In addition, given the continuous turmoil of secondary education in Spain, which creates a high heterogeneity of students at different points in time it is difficult to quantify any approach.

In order to assess some feedback from the students, we passed a survey in a 3rd-year course class. Of course, this survey should not be considered an indisputable evidence, but it points in the right direction. We polled 3rd-year students because they have learnt other programming languages and have a wider vision. On the other hand, there is an implicit bias because many students abandon the degree (for many reasons, but the common case is the difficulty of the degree, not necessarily CS1). The results of the survey are shown in Figure 1. The questions were:

(A) *How did you like Picky as your first language programming language?*

(B) *Did using a simple language in CS1 helped you to learn more than a complex but powerful language?*

(C) *Was it difficult to learn the Picky syntax in CS1?*

(D) *Was it difficult to learn the Ada syntax in CS1?*

Questions A to C were given to students that used Picky as a first language. Question D was given to students that used Ada instead.

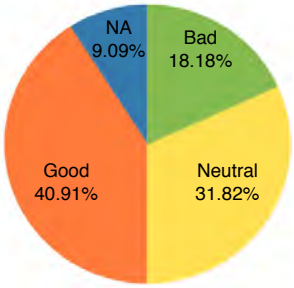
The experience with the language is positive. We do see the students less engaged in nitpicking with the unimportant details of the language and more focused on the learning task. In our opinion, Picky has made teaching simpler and the students learn more compared to other introductory courses the authors have taught in Ada and C. Before using Picky, the authors had to explain to students how things in practice departed from what was taught in theory. This was an imposition of the language being used (e.g. the *eof* function with side effects in Ada). In addition, the students had problems regarding dynamic memory, uninitialized memory, and all the other issues enumerated early in this paper. Picky has alleviated most of these problems.

One disadvantage of creating a custom language for the course, is the absence of ready-made materials for teaching the subject and for student consultation. In order to cover this gap, we wrote an introductory programming book (in Spanish) using Picky [3] for the course. This book covers the course following the same approach and in the same order we cover it in class. It serves two purposes. On the one hand, it is a reference material for the students, with some extra content for the more advanced students. On the other hand it serves as a guide for the teachers, helping to provide a detailed guideline of what should be taught in class and in what order.

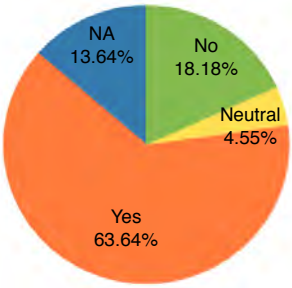
The absence of ready-made code snippets to copy from the network helps make the students work more in their assignments and spend less time forcing code copied from a random web page into them.

Another unanticipated benefit of using a language built by ourselves, is, of course, that we understand it thoroughly. With more complicated languages, it is always possible to have a dark corner of the language appear in code written by students which puzzles the teacher, sometimes momentarily, sometimes longer. While the response to the student is simple: "rewrite that mess", more advanced students may want to understand what exactly is going on. For instance, one of the authors remembers fondly trying to understand an accidental and obscure variation on the Duff device [9] to be able to explain to a good student why his code worked. With Picky, these days are over.

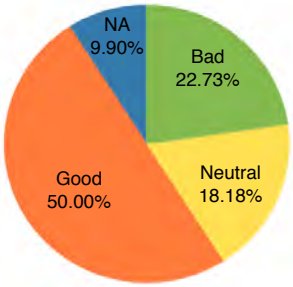
As every teacher knows, plagiarism detection is an important issue whenever students are given assignments. While we were concerned when we started that we would have to write our own tools for this purpose, we found that the already existing tool Moss [1] works very well with Picky and we use it routinely on the assignments.



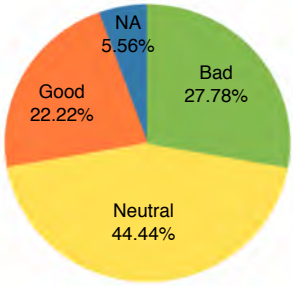
Survey A



Survey B



Survey C



Survey D

International Conference, pages 317–322, 1996.

[18] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. *SIGCSE Bull.*, 39(4):204–223, Dec. 2007.

[19] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[20] E. S. Roberts. Using c in cs1: evaluating the stanford experience. In *ACM SIGCSE Bulletin*, volume 25, pages 117–121. ACM, 1993.

[21] R. M. Siegfried, D. Chays, and K. Herbert. Will there ever be consensus on cs1? In H. R. Arabnia, V. A. Clincy, and N. Tadayon, editors, *Proceedings of the 2008 International Conference on Frontiers in Education, FECS 2008, July 14-17, 2008, Las Vegas, Nevada, USA*, pages 18–23. CSREA Press, 2008.

[22] M. Vujoševic-Janicic and D. Tošic. The role of programming paradigms in the first programming courses. *The Teaching of Mathematics*, 11(2):63–83, 2008.

Identification of Inhibitors of Fatty Acid Synthesis Enzymes in Mycobacterium Tuberculosis

Alexander Priest[1], E. Davis Oldham, Lynn Lewis
University of Mary Washington
1301 College Avenue
Fredericksburg, VA 22401
apriest,eoldham,llewis@umw.edu

David Toth[2]
Centre College
600 West Walnut Street
Danville, KY 40422
david.toth@centre.edu

ABSTRACT
Antibiotic-resistant strains of *Mycobacterium tuberculosis* have rendered some of the current treatments for tuberculosis ineffective, creating a need for new treatments. Today, the most efficient way to find new drugs to treat tuberculosis and other diseases is to use virtual screening to quickly consider millions of potential drug candidates and filter out all but the ones most likely to inhibit the disease. These top hits can then be tested in a traditional wet lab to determine their potential effectiveness. Using supercomputers, we screened over 4 million potential drug molecules against each of two enzymes that are critical to the survival of *Mycobacterium tuberculosis*. During this process, we determined the top candidate molecules to test in the wet lab.

Categories and Subject Descriptors
J.3 [**Life and Medical Sciences**]: Biology and genetics.

General Terms
Experimentation.

Keywords
Computational science education, drug discovery, virtual screening, parallel computing education.

1. INTRODUCTION
Since their discovery in the early 20th century, antibiotics have seen exponential growth in usage due to their unparalleled efficacy for the treatment of bacterial diseases [1]. Unfortunately, because this method of treatment is relatively new, we are only just now observing the ramifications of their ubiquity; widespread use and misuse of antibiotics has become a force of natural selection for bacteria, and as a result, these pathogens are evolving to resist them [2]. Antibiotic-resistant strains of many disease-causing bacteria have been observed, and among these is the causative agent of tuberculosis, *Mycobacterium tuberculosis* [2, 3]. Tuberculosis affects millions of people worldwide to this day, and a variety of reasons that have contributed to resistant strains of the disease have resulted in a critical need to search for novel drug treatments [3, 4]. In the past, this has been accomplished by taking soil samples and plating them to look for naturally occurring antibiotic producers, but as this research has gone on, it is more difficult to find novel antibiotic producers [5]. With this in mind, it is easy to see a need for new methodologies to come into play. In the age of technology, there has been an increase in the use of computers to ease research processes like this. For example, there are several molecular docking programs which exist now that are designed to simulate the binding interactions of molecules with protein targets, including AutoDock Vina, DOCK, GOLD, and Glide [6, 7, 8, 9]. Screening molecules with a molecular docking program is much faster and more convenient than testing for inhibitors with *in vitro* methods. We can use this technology to investigate novel mechanisms for antibacterial compounds. Rather than waiting a week or more for a panel of bacterial plates to respond to exposure to potential drug candidates, these programs can give us an idea of how strong the interaction would be in a matter of minutes of compute time per compound. In this study, we used an *in silico* virtual drug screening process to comb through approximately 4.2 million ligands as potential drugs to target a critical enzyme in *M. tuberculosis*. To deal with the logistical issues of the sheer compute time this required, we decided to run the virtual screen on a supercomputer capable of running thousands of simulations at the same time, achieving a throughput unmatched by any *in vitro* assay method. However, while the results of a virtual screen indicate which molecules are likely to bind to a target protein, it does not necessarily mean the molecules will actually bind to the protein and even more importantly, inhibit the protein [10]. Because of this shortcoming, the virtual screening process is used as a first phase in the drug discovery process, filtering out the vast majority of molecules which likely will not bind to the protein [11]. After the virtual screening is completed, the top hits are screened with biological assays to test which molecules will actually work as treatments [11].

2. RELATED WORK
Using virtual screening to narrow down the list of compounds to test in a wet lab with biological assays has become accepted over the last number of years, and people from various research groups are using this method [12]. The corresponding author has worked with teams using virtual screening on several projects [13]. In one such study, the target was an essential enzyme found in *Plasmodium* sp., the causative agent of malaria [14]. The open-source docking simulation program AutoDock Vina, designed at the Scripps Research Institute, was used to screen the full_nci_ALL_TAUTOMERS_2011 library of about 320,000 chemical compounds from the ZINC database against the enzyme PfUCHL3 [6, 15]. The top scoring compounds were then re-screened against the human analog for this enzyme to determine which would be safest for human use; these were then screened in vitro in the lab to confirm their efficacy against *Plasmodium*. As a

[1] Undergraduate Student

[2] Corresponding Author

result, the authors determined two compounds with very high promise as novel malaria treatments which would be effective without causing side effects due to binding with the human orthologous protein. Franco et al. also screened DrpE1 in an attempt to find a novel cure for tuberculosis [16].

3. METHODS

The first step in our drug discovery process was to pick a target. We first decided on targeting *M. tuberculosis* because of its recent trend in antibacterial resistance; novel drugs for tuberculosis would be especially sought after as a result [3]. We chose two enzymes (called Target 1 and Target 2 in this paper) that are critical to the survival of *Mycobacterium tuberculosis*, and we understand the mechanisms of their action. Next, we found the structures for these proteins from the Protein Data Bank (PDB) and prepared them for docking using AutoDock Tools [17, 18]. The structures were derived from X-ray diffraction, and were specific to *Mycobacterium tuberculosis* [17]. Of the different structures available, we selected the wild-type structures including a ligand in the binding pocket. We also used AutoDock Tools to locate the coordinates of the binding site and noted these down. We then uploaded the prepared molecules and coordinates to the Blue Waters and Stampede supercomputers, along with the molecular docking program AutoDock Vina. We obtained about 4.2 million ligand files from the ZINC database and downloaded these as well, and created shell scripts to break the work into pieces. We created another shell script to run the program for each compound, and another to collect and package the results for download and analysis. Once we downloaded the data, we uploaded it to an SQL database and searched for the top hits.

4. RESULTS

The results of the virtual screens were grouped into bins based on the binding affinities of the compounds. These bins allow us to separate the most promising compounds from the rest and determine which compounds should be tested with assays. Table 1 and Table 2 show the number of compounds in each binding affinity range. For Target 1, 4,182,163 compounds were screened and for Target 2, 4,182,137 compounds were screened. Figure 1 and Figure 2 show the binding affinities for the compounds screened against Target 1 and Target 2, respectively. Figure 3 and Figure 4 break down the top hits for Target 1 and Target 2 into bins of narrower width. It is important to note that the best binding energies are the ones with the most negative values, so a compound with a binding energy of -13 is more likely to bind to the target than a compound with a binding energy of -12. The top hits for Targets 1 and 2 are given in Table 3 and Table 4.

Table 1 - Summary of Binding Affinities of Virtual Screen against Target 1

Binding Affinity Range (kcal/mol)	Number of Compounds in Range
-13 ≥x > -14	8
-12 ≥x > -13	139
-11 ≥x > -12	3,576
-10 ≥x > -11	55,866
-9 ≥x > -10	413,115
-8 ≥x > -9	1,377,570
-7 ≥x > -8	1,607,582
-6 ≥x > -7	606,245
-5 ≥x > -6	94,722
-4 ≥x > -5	20,151
-3 ≥x > -4	3,082
-2 ≥x > -3	94
-1 ≥x > -2	6
0 ≥x > -1	2
x > 0	5

Table 2 - Summary of Binding Affinities of Virtual Screen against Target 2

Binding Affinity Range (kcal/mol)	Number of Compounds in Range
-13 ≥x > -14	4
-12 ≥x > -13	91
-11 ≥x > -12	3,756
-10 ≥x > -11	71,393
-9 ≥x > -10	571,938
-8 ≥x > -9	1,453,342
-7 ≥x > -8	1,101,984
-6 ≥x > -7	443,499
-5 ≥x > -6	192,355
-4 ≥x > -5	108,482
-3 ≥x > -4	68,534
-2 ≥x > -3	47,881
-1 ≥x > -2	33,959
0 ≥x > -1	25,098
x > 0	59,821

5. CONCLUSIONS

Using the Blue Waters and Stampede supercomputers, we have screened over 4.1 million compounds against two enzymes that are critical to *Mycobacterium tuberculosis* surviving. The virtual screens have indicated 12 compounds with a binding affinity of < -13 that are likely to bind to *Mycobacterium tuberculosis*. If those compounds can indeed bind to the target enzymes in tuberculosis and inhibit the functioning of those enzymes, then the compounds may be useful in treating tuberculosis.

6. FUTURE WORK

For future work, we will test as many of the top hits as we can in the wet lab. The compounds that scored in the -13.0 to -13.9 range will be prioritized. High scoring compounds with different structures will also be prioritized to give a wide range of coverage of different types of compounds. We note that an entity with

Table 3 - Top Hits for Target 1 from the ZINC Database Libraries Screened

Score	Library	Folder	Compound
-13.5	full_nci_ALL_TAUTOMERS_2011	SetOf10k_0004	ZINC01588230.pdbqt
-13.3	ChemBridge_FullLibrary2011	SetOf10k_0037	ZINC02880067.pdbqt
-13.2	asinex_newMay2011_fixedForVinaInDec	SetOf10k_0000	ZINC06281466.pdbqt
-13.2	asinex_newMay2011_fixedForVinaInDec	SetOf10k_0014	ZINC13564997.pdbqt
-13.1	asinex_newMay2011_fixedForVinaInDec	SetOf10k_0019	ZINC13565797.pdbqt
-13.1	asinex_newMay2011_fixedForVinaInDec	SetOf10k_0014	ZINC13564995.pdbqt
-13.0	asinex_newMay2011_fixedForVinaInDec	SetOf10k_0014	ZINC13564992.pdbqt
-13.0	asinex_newMay2011_fixedForVinaInDec	SetOf10k_0018	ZINC13084337.pdbqt
-12.9	ChemBridge_FullLibrary2011	SetOf10k_0000	ZINC02833848.pdbqt
-12.9	asinex_newMay2011_fixedForVinaInDec	SetOf10k_0014	ZINC13565000.pdbqt
-12.9	asinex_newMay2011_fixedForVinaInDec	SetOf10k_0042	ZINC13564941.pdbqt

Table 4 - Top Hits for Target 2 from the ZINC Database Libraries Screened

Score	Library	Folder	Compound
-13.9	full_nci_ALL_TAUTOMERS_2011	SetOf10k_0016	ZINC04824645.pdbqt
-13.1	asinex_newMay2011_fixedForVinaInDec	SetOf10k_0008	ZINC04838539.pdbqt
-13.1	ChemBridge_FullLibrary2011	SetOf10k_0069	ZINC19634897.pdbqt
-13.0	asinex_newMay2011_fixedForVinaInDec	SetOf10k_0025	ZINC06475337.pdbqt
-12.9	ChemBridge_FullLibrary2011	SetOf10k_0007	ZINC04980431.pdbqt
-12.8	full_nci_ALL_TAUTOMERS_2011	SetOf10k_0015	ZINC04428442.pdbqt
-12.7	ChemBridge_FullLibrary2011	SetOf10k_0087	ZINC16662786.pdbqt
-12.7	ChemBridge_FullLibrary2011	SetOf10k_0029	ZINC02893797.pdbqt
-12.6	ChemBridge_FullLibrary2011	SetOf10k_0074	ZINC19634255.pdbqt
-12.6	ChemBridge_FullLibrary2011	SetOf10k_0049	ZINC19632616.pdbqt
-12.6	ChemBridge_FullLibrary2011	SetOf10k_0074	ZINC23281397.pdbqt

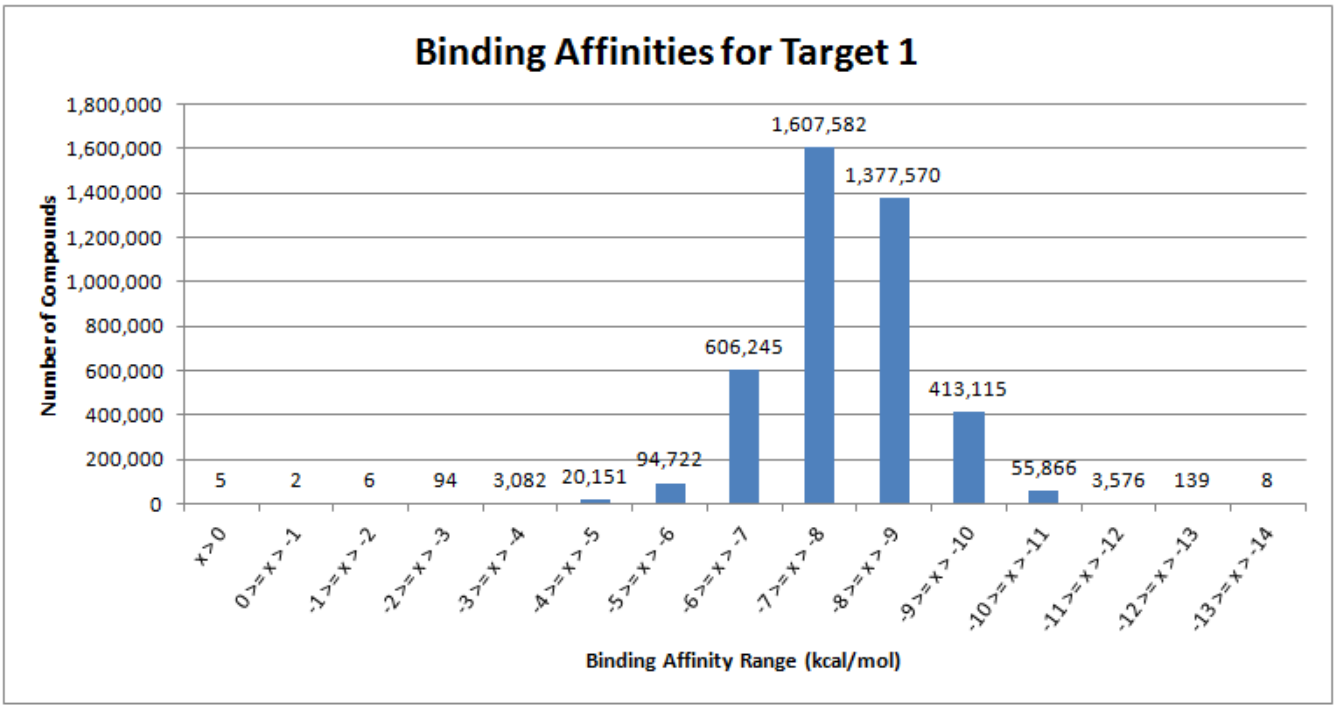


Figure 1 - Results of Virtual Screen of Compounds against Target 1 with Binding Affinities Grouped in Ranges

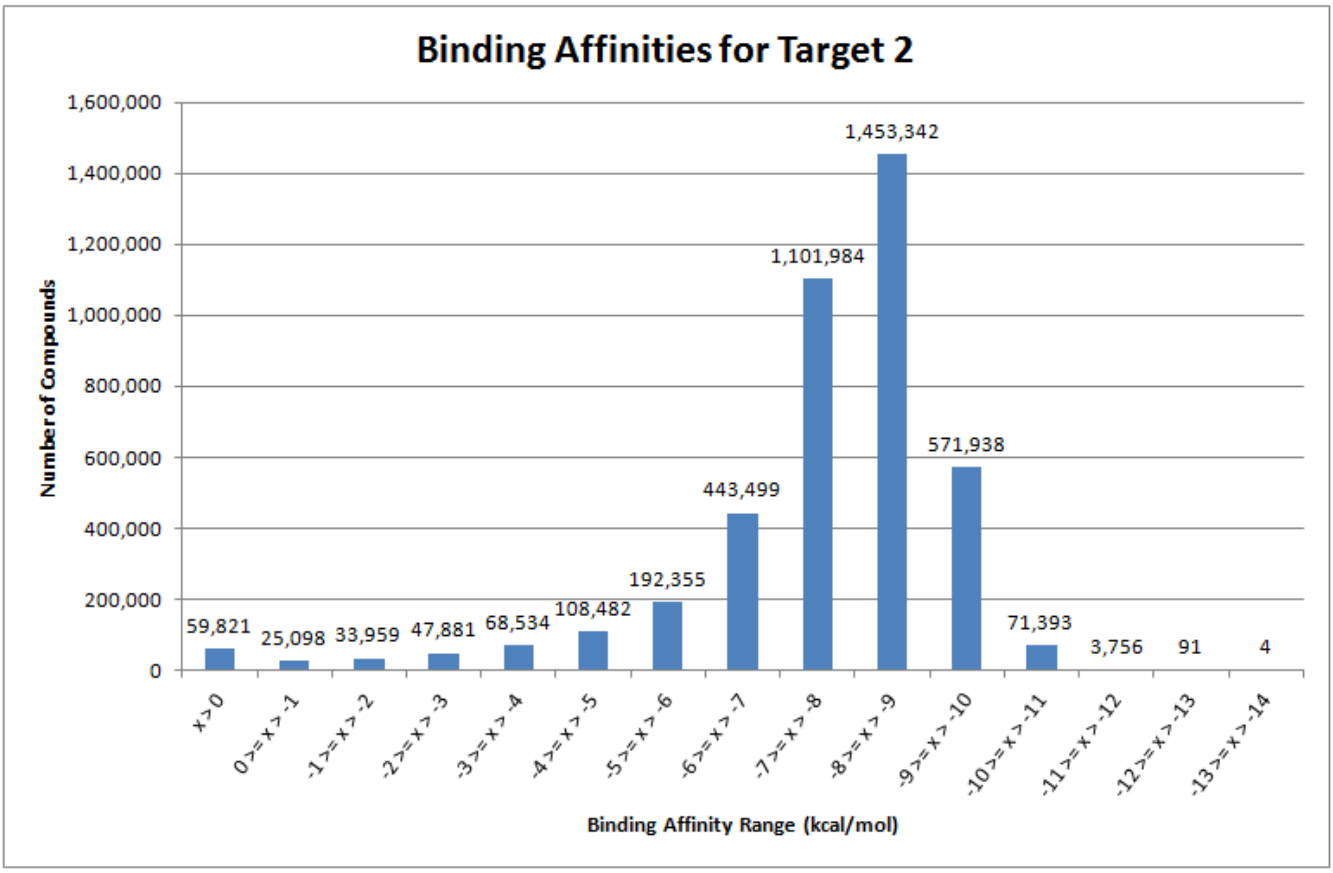


Figure 2 - Results of Virtual Screen of Compounds against Target 2 with Binding Affinities Grouped in Ranges

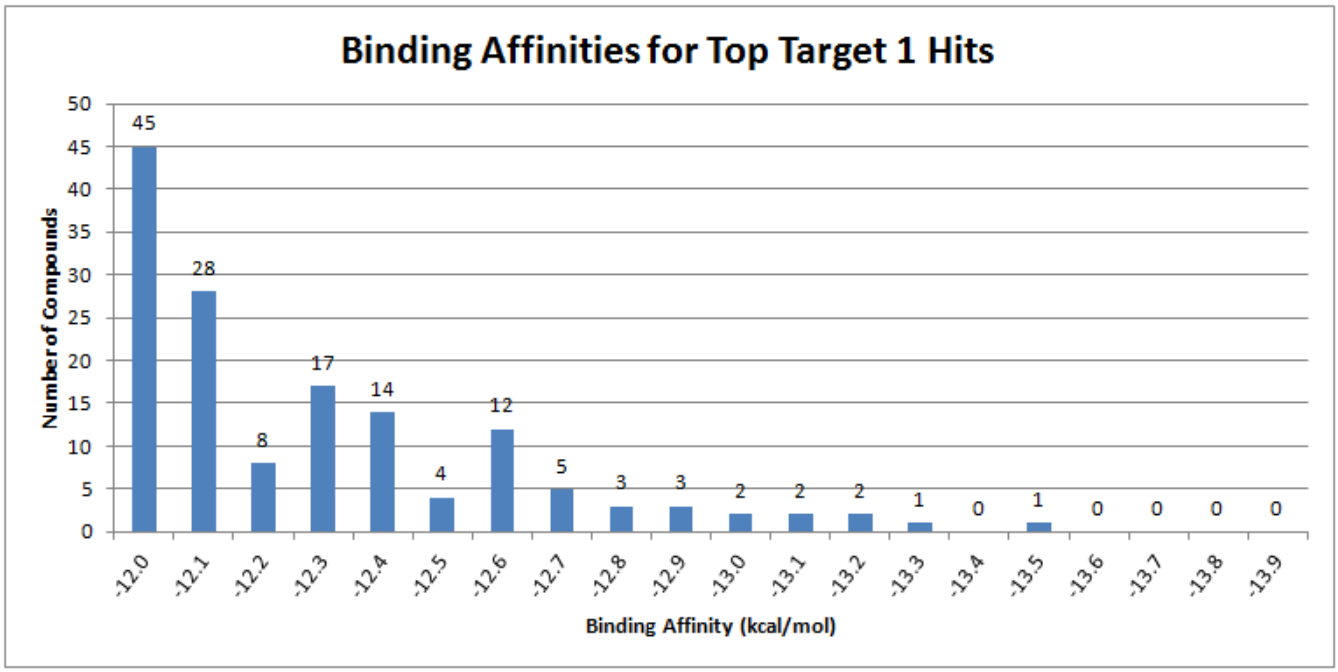


Figure 3 - The Top Binding Affinities for Virtual Screen of Compounds against Target 1 Grouped in Ranges

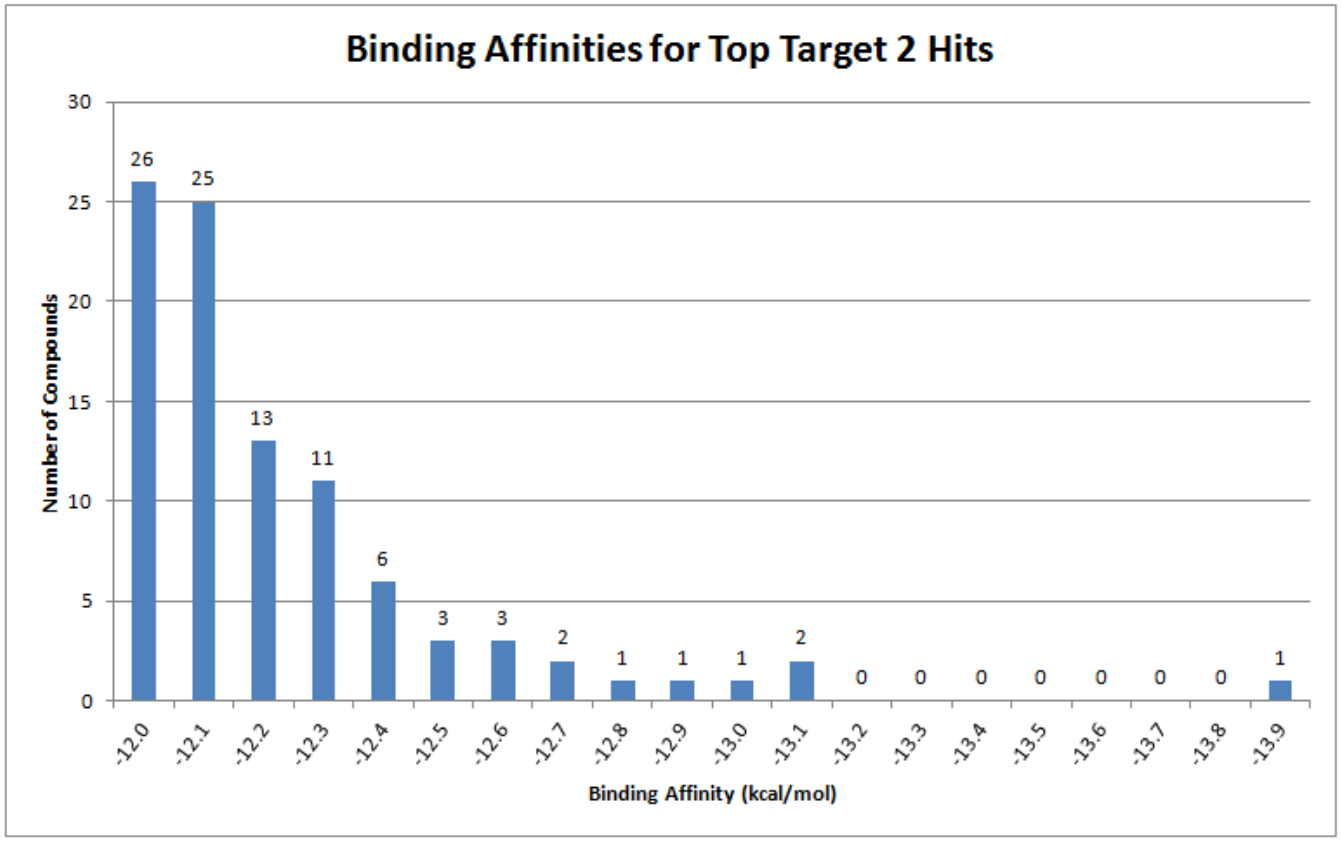


Figure 4 - The Top Binding Affinities for Virtual Screen of Compounds against Target 2 Grouped in Ranges

significant financial resources might test all the compounds scoring -12 or better.

We will apply the top scoring compounds to cultures of a model organism, *Mycobacterium bovis*, and determine if any of these compounds has an inhibitory effect on the growth of the bacteria [19]. We will use *Mycobacterium bovis* because unlike *Mycobacterium tuberculosis*, *Mycobacterium bovis* is not pathogenic [19]. Should one or more compounds prove effective at inhibiting the bacterial growth, the next step requires the resources of a larger organization. Further testing of the successful compounds would be necessary to confirm their action on the cells. Following this, research would be done to determine whether the targeted proteins have any human orthologs, or similar proteins which occur in the human body which may also be affected by the compound, resulting in unwanted side effects. *In vivo* testing with a model host would be the next step, as the compound would need to be proven safe for the consumption of the host organism.

7. REFLECTIONS

The project described in this paper was the first author's Blue Waters Student Internship project where he learned to incorporate computation and high-performance computing into his research. This section details the first author's reflections about his internship and the impact that it has had on his current and future academic endeavors: When I took my first course in computer science, I did not anticipate that it would give me the power to make a difference like this. At the time of beginning this project, I was a Biology major, with minors in Chemistry and Neuroscience. I could tell you a lot about how diseases like tuberculosis can ravage the human body. I could tell you how the increase in the prevalence of antibacterial soaps may have actually led to the rise of hyper-resistant superbugs. However, I could never have explained to you any way in which I could make a difference as an undergraduate student in any of these areas. Before I got involved with computer science, university was simply a place for learning, not for doing. I started with a single course on modeling and simulation which required no formal coding skill (we used drag-and-drop programming environments like Scratch), and grew into learning the basics of C++ in a week before attending the 2-week intensive high performance computing workshop for Blue Waters interns. At the workshop, I learned parts of the C and FORTRAN programming languages in order to learn the basics of the parallel computing libraries OpenMP, CUDA, MPI, and OpenACC. Having only taken a single introductory course in computer science before attending the workshop, I am proud of how much I was able to learn. Now, I am confident using a Linux command prompt and I can write some basic shell scripts. Having learned these skills, I am capable of using supercomputers for my research, which spans biology and chemistry. One of the most lasting impacts that this incredible experience has left me with, however, is my recent decision to stay an extra year at UMW in order to pursue a double major in Computer Science alongside my Biology major, and to add a Data Science minor. I am planning on finding a graduate school that will have the same zeal for interdisciplinary projects that I have now, and I believe that these experiences will make me very competitive in the application process. Having this Blue Waters Internship was genuinely a turning point in my college career and my life in general, and having this opportunity to do real research with real-world implications is a once-in-a-lifetime experience. This project represents the mixing of the disciplines that needs to happen if science as a whole is going to make new discoveries this century to rival the marvels of the past. Computer modeling for scientific applications is certainly the way research will be conducted in the future, and the future is not so far away after all.

8. ACKNOWLEDGMENTS

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. We thank the Blue Waters Student Internship Program for providing Alexander with this opportunity. We also wish to thank XSEDE for providing the vast majority of the compute time for Alexander to conduct the research through grants MCB140189 and MCB140209 and we thank the Texas Advanced Computing Center (TACC), where Stampede resides. Finally, we thank the University of Mary Washington, which provided Alexander with room and board for the summer through their Summer Science Institute and funding for the wet-lab studies.

9. REFERENCES

[1] Davies, Julian, and Davies, Dorothy. Origins and Evolution of Antibiotic Resistance. *Microbiol. Mol. Biol. Rev.* September 2010 vol. 74 no. 3 417-433. doi: 10.1128/MMBR.00016-10.

[2] Alanis, Alfonso. Resistance to Antibiotics: Are We in the Post-Antibiotic Era?, Archives of Medical Research, 36, (Nov.-Dec., 2005), 697-705, doi:10.1016/j.arcmed.2005.06.009.

[3] Rivers, Emma C. and Mancera, Ricardo L. New anti-tuberculosis drugs in clinical trials with novel mechanisms of action, Drug Discovery Today, 13, (Dec. 2008), 1090-1098, doi:10.1016/j.drudis.2008.09.004.

[4] World Health Organization, http://www.who.int/mediacentre/factsheets/fs104/en/.

[5] Ling et al. A new antibiotic kills pathogens without detectable resistance, *Nature* 517, 455–459 (Jan. 2015) doi:10.1038/nature14098.

[6] Trott, O. and Olson, A.J. AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization and multithreading. *Computational Chemistry*, 31, (Jan. 2010), 455-461, DOI: 10.1002/jcc.21334.

[7] P. T. Lang, S. R. Brozell, S. Mukherjee, E. F. Petterson, E. C. Meng, V. Thomas, R. C. Rizzo, D. A. Case, T. L. James, and I. D. Kuntz, DOCK 6: Combining techniques to model RNA–small molecule complexes. RNA 2009, 15, 1219-1230.

[8] Verdonk ML, Cole JC, Hartshorn MJ, Murray CW. et al. Improved protein-ligand docking using GOLD. Proteins. 2003;52:609–623. doi: 10.1002/prot.10465.

[9] Friesner RA, Banks JL, Murphy RB, Halgren TA. et al. Glide: A new approach for rapid, accurate docking and scoring. 1. Method and assessment of docking accuracy. J Med Chem. 2004;47:1739–1749. doi: 10.1021/jm0306430.

[10] Shoichet, Brian. Virtual screening of chemical libraries. *Nature*. 2004 Dec 16; 432(7019): 862–865.

doi: 10.1038/nature03197

[11] Cheng, Tiejung, Li, Qingliang, Zhou, Zhigang, Wang, Yanli, and Bryant, Stephen H. Structure-Based Virtual Screening for Drug Discovery: a Problem-Centric Review. *AAPS J*. 2012 Mar; 14(1): 133–141. doi: 10.1208/s12248-012-9322-0

[12] Ellingson, Sally R., Smith, Jeremy C., and Baudry, Jerome. 2014. Polypharmacology and supercomputer-based docking: opportunities and challenges, *Molecular Simulation*, DOI: http://dx.doi.org/10.1080/08927022.2014.899699.

[13] Toth, David, Franco, Jimmy, and Berkes, Charlotte. Attacking HIV, Tuberculosis and Histoplasmosis with XSEDE Resources. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery* (San Diego, CA, USA), July 22-25, 2013). ACM, New York, NY. DOI: http://dx.doi.org/10.1145/2484762.2484766.

[14] Franco, Jimmy, Blackie, Margaret A.L., Toth, David, Smith, Peter J., Capuano, Joseph, Fastnacht, Kurt, Berkes, Charlotte. A structural comparative approach to identifying novel antimalarial inhibitors. *Computational Biology and Chemistry*, 45, (Aug. 2013), 42-47. DOI: http://dx.doi.org/10.1016/j.compbiolchem.2013.04.002.

[15] Irwin, Sterling, Mysinger, Bolstad and Coleman, ZINC – A Free Database of Commercially Available Compounds for Virtual Screening. *Chem. Inf. Model*. 2012 DOI: 10.1021/ci3001277

[16] Wilsey, Claire, Gurka, Jessica, Toth, David, and Franco, Jimmy. A large scale virtual screen of DprE1. *Computational Biology and Chemistry*, 47, (Dec. 2013), 121-125, DOI: http://dx.doi.org/10.1016/j.compbiolchem.2013.08.006.

[17] RCSB Protein Data Bank. http://pdb.org/pdb/home/home.do.

[18] Morris, G. M., Huey, R., Lindstrom, W., Sanner, M. F., Belew, R. K., Goodsell, D. S. and Olson, A. J. Autodock4 and AutoDockTools4: automated docking with selective receptor flexiblity. *Computational Chemistry* 16 (2009), 2785-91.

[19] Altaf, Mudassar, Miller, Christopher H., Bellows, David S., and O'Toole, Ronan. Evaluation of the *Mycobacterium smegmatis* and BCG models for the discovery of Mycobacterium tuberculosis inhibitors. Tuberculosis, 90, (Nov. 2010), 333-337. http://dx.doi.org/10.1016/j.tube.2010.09.002

July 2015

Volume 6 Issue 1

TABLE OF CONTENTS

ISSN 2153-4136 (online)