Teaching Introduction to Programming as Part of the IS Component of the Business Curriculum

Borislav Roussev
Susquehanna University, Selinsgrove, USA

roussev@susqu.edu

Abstract

Modern software practices call for the active involvement of business people in the software process. Therefore, programming has become an indispensable part of the IS component of the core curriculum at business schools. In this paper, we present a model-based approach to teaching introduction to programming to general business students. The underpinnings of the new approach are modeling, abstraction, and Bloom's classification of cognitive skills. We employ models to introduce the basic programming constructs and their semantics. To this end, we use statecharts to model object's state, the environment model of evaluation as a virtual machine interpreting the programs written in JavaScript, and UML class diagrams to represent the static structure of the designed software systems. The adoption of this approach helps learners build a sound mental model of the notion of computation process. Learners' achievements, student evaluations, and our subjective opinion suggest that the proposed ideas improve the course significantly.

Keywords: introduction to programming, model-based approach, teaching programming

Introduction

Modern software practices call for the active involvement of business people in the software process (Shaw, 2000). Apart from the crucial role of end users in eliciting precise user requirements, business people are intimately involved in all stages of software development (Jacobson, 99). At present, software is commonly adapted, composed from reusable components and frameworks, and even created by business people rather than information systems (IS) developers (Shaw, 2000). Therefore, programming has become an indispensable part of the IS component of the core curriculum at business schools.

The traditional curriculum design offers four IS courses in the following succession: (1) Using databases; (2) Systems analysis and design; (3) E-Business applications development; and (4) Management support systems.

At our school, E-Business applications development (a.k.a. Client-server, Web-based programming, E-commerce) is the only programming course mandatory for all business students. E-Business applications development is the basic hands-on-experience course, where students develop Web-based e-commerce applications. A major constituent of the course is programming in JavaScript. Introduction to programming in JavaScript is the topic of this paper.

The underlying assumption of this research work is that adopting a model-based approach to teaching programming will enhance the students' ability to think and reason formally about programs, develop software rigorously, and program better.

Bloom's taxonomy of cognitive skills is widely recognized as a basis for classifying skills in edu-

cation and ordering material (Bloom, 1956). Bloom's taxonomy divides cognitive skills into six levels where factual knowledge, theory comprehension and theory application come before analysis, synthesis and finally evaluation.

In our approach we strictly adhere to Bloom's hierarchy of cognitive skills. We start with simple, yet precise, facts about (formal model of) the notion of computation. Then, we gradually enrich the basic model with data types and computational objects, thus generalizing the facts into a consistent theory. Next, we repeatedly ask learners to apply the concepts (theory) introduced to predict initially the behavior of snippets of code and later of middle-size programs. After students have acquired the skills to predict the behavior of programs of any size, we proceed with program analysis and synthesis, or what is commonly referred to as programming.

The rest of the paper is structured as follows. Section 2 attends to the presentation of the model-based approach used in teaching introduction to programming. Next, Section 3 discusses preliminary results about the advantages of adopting such an approach. The final section summarizes the experience gained and concludes.

Teaching Programming: Model-Based Approach

The novelty of our approach lies in teaching the lower-order cognitive skills. We use models to introduce the basic programming constructs. To introduce the notion of state and thus help students appreciate the environment model of evaluation, we begin by modeling with labeled transition systems, a subset of statecharts (Harel, 1987), (Booch, 1999). Our experience confirmed the findings of Davis (1988) that the introduction of this model takes an hour on average. Students are given classical examples from the theory of Finite Automata, such as vending machines, flying robots, and multi-user games. The exercises are a prelude to program state and objects with interesting state machines.

Building large software-intensive systems requires a methodology that would allow structuring the systems into modular components that could be independently developed and maintained. To avoid a semantic gap between the domain of interest and the designed system, we base the structure of the latter on the structure of the physical system that is represented in the computing system (Abelson et al., 1996). The real world is populated with different objects, each one possessing a number of characteristic attributes, e.g., a bank account has an owner and interest rate. The attributes alone fail to represent the dynamic nature of objects. It is highly desirable to know the actions in which the objects can get involved, i.e. the behavior the objects can exhibit. For example, a bank account can accumulate interest rate, can be credited or debited. In order to determine the future behavior of an object, we need to know its history. The history of a bank account could be succinctly represented by its balance rather than by recording the complete sequence of past transactions. The abstract representation of an object's history is called state. After students have comprehended object's state as a potential for future actions, the notion of program variable is introduced. The view that a system is composed of objects, each of which has its own state, poses the question of how to represent this state. To store the object's state, one or more attributes could be added to the set of existing ones. For example, an attribute balance can represent the state of a bank account. Since the state of an object changes over time, the values of the attributes representing the state in the computational objects must also change. Thus, students learn that the decision to base our programs on the structure of the real-world objects it represents necessitates computational objects, the state of which changes as the programs run.

The possibility of associating names with values and later retrieving and changing those values in a way dependent upon their types means that the language processor (the JavaScript interpreter) must maintain some sort of memory that keeps track of the values and the values' names and types (name-type-value triplets). To this end, we use the environment model of evaluation (Baber, 1987). An environment de-

fines the context in which an expression or a statement is evaluated. It should be thought of as an abstraction of the underlying hardware (memory, CPU, etc.), operating system and language processor.

Definition 1: An *environment frame*, or environment for short, is a sequence of variables (data items in general).

For example, d_0 = [(s, string, "Jones"), (i, number, 0)], is an environment consisting of two variables. Environment d_0 can be graphically represented as shown in Figure 1.

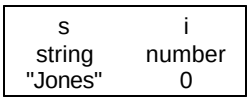

Figure 1: Graphical representation of environment.

Each triplet in the environment is a data item consisting of a name, data type, and value from the domain of the data type, see Figure 2. It is important for the students to internalize the structure of data types, and what can be regarded as a data type, see Table 1. Learners should grasp the idea of binding among name, type and value. Since JavaScript is a loosely-typed language, it hides the notion of data type from the programmer. The latter encourages a hacker style of programming and leads to an increased number of typing errors. That is why taking the data type into consideration is of utmost importance.

The *value* of a data item (variable) x in an environment d is the value of the first item whose name is x. If the environment d does not contain a data item with the name x, then the value of x is undefined. A data declaration creates a variable and prefixes its triplet to the environment, for example, the following declaration evaluated in d_0:

```
var amt = 20;
```

Type	Set of Values	Set of Operations
boolean	True false	and or not
Number	...,1,0,+1,... decimal numbers	+,–,*,/ +,–,*,/
String	'''' ''a'' ''xyz'' ''hello''	concatenation equality characterSelection substringSelection

Table 1: Basic data types and their structure.

results in a new environment d_1 defined as:

d_1 = [(amt, number, 20), (s, string, "Jones"), (i, number, 0)].

Next, we consider computational objects. The following algorithmic procedure is applied to evaluate a statement (or an expression) on an environment.

1. Identify the operator with the highest precedence level.
2. Evaluate its operands
3. Apply the primitive function corresponding to the operator identified in Step 1 to the evaluated operands.

For instance, to evaluate the expression,

```
amt <= i + 10
```

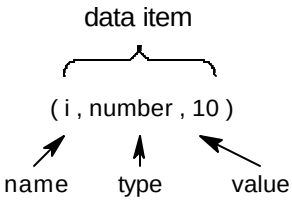

Figure 2: Named data item structure.

on the environment d_1 above, the precedence levels of <= and + are looked up in the operators' precedence table, see Table 2. The precedence of +, level 5, is higher than that of <=, level 4, see Figure 3. Therefore, + is applied first. Next, Step 2 is carried out. To evaluate i, d_1 is scanned from left to right looking for the first occurrence of a variable with name i. Then, the value of this variable is retrieved and substituted for i in expression. The evaluation procedure is applied recursively until the expression is reduced to a literal.

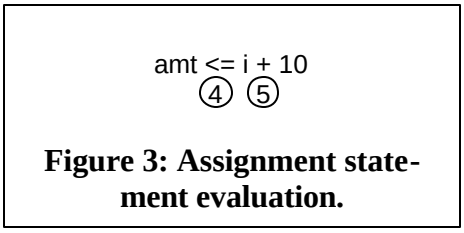

Figure 3: Assignment statement evaluation.

Level	Operators	Description
0	=	assignment
1	||	logical-or
2	&&	logical-and
3	==, !=	comparison
4	<, <=, >, >=	relational
5	+, -	addition, subtraction
6	*, /	multiple, divide
7	!	logical-not
8	<name>()	function call
9	[], .	subscripting, member
10	()	parenthesis

Table 2: A subset of JavaScript operators with precedence levels.

Assuming that the mechanism for applying JavaScript operators is built into the language interpreter, the main issues confronting the learner at this point are precedence and associativity. For practice, a JavaScript interpreter developed by Nombas, called ScriptEase (2002), is used. A sample session with ScriptEase interpreter looks like:

```
Screen.writeln(7 + 3 * 2);
42
```

The response of the interpreter is shown in slanted characters. In reality, it is directed to a separate output window.

The semantics of the assignment statement, the sequence of statements, the if statement, the iterator for/in, and function definition/invocation, are defined operationally by specifying the effect of their execution on an environment. As an example, consider the definition of if.

Definition 2: An *if statement*, if (c) s1 else s2, consists of a predicate (expression that evaluates to true or false) c, called condition, a then-clause s1 and an else-clause s2. The if statement is executed on the environment d by first evaluating the condition in d. Depending upon whether its value is true or false, s s1 or s2, respectively, is executed in d.

A typical exercise for the if statement given to students is as follows.

Let the environment d be

d_2 = [(z, number, 1), (x, number, 9), (y, number, 2)]

Determine the new contents of d for the following if statement:

```
if (x < 9) y = -x; else y = x;
```

The resulting environment is:

d_3 = [(z, number, 1), (x, number, 9), (y, number, 9)]

The basic principle we used in defining compound computations thus far, both at expression and statement level, is combination. We make a point that combination, however useful, does not suppress the detail, and not before long the designed programs become too complex. What we need is a way to deal with the complexity in large systems design, i.e., we need abstraction.

Definition 3: *Function definition* is an abstraction technique by which a block of statements specifying a compound operation is given a name and then referred to by this name.

In JavaScript, the function definition has the following basic form:

```
function name( formal_parameters ) { body }
```

We illustrate the mechanism of applying a function to its arguments with a simple example. The function `inc()`, often called the successor function, increments its argument by one and returns the result to the caller. It is defined as:

```
1| function inc(x) {
2|   return x + 1;
3| }
```

The function can be used as a building block in expressions, respectively statements, for example:

```
4| var z = inc(3);
```

The function call, `inc(3)` is equivalent to the following block statement created at runtime from the function definition of `inc()`:

```
{
   var x = 3;
   return x + 1;  // equivalent to z = 4;
}
```

Next, we point out a major source of inefficiency in utilizing environment space under multiple function calls. What if we call `inc(3)` not once but thousands of times. What if a large number of variables are declared in the body of the function, for example a huge array, and then shadowed forever by subsequent function calls. Both scenarios are typical programming practices. When representing real objects in our programs, we want to model their behavior as closely as possible. For example, disposing of a variable storing an account balance will be disastrous. On the other hand, a variable used to hold a result of an intermediate step in a long computation, e.g. a successive approximation in a numerical integration, is not worth keeping forever. It takes up environment space.

To sum up, we identified a class of variables holding intermediate results and formal parameters that are needed only over a short period of time. To deal with this inefficient environment utilization, we can divide the data environment in two parts, called *global* and *local*, respectively (the local environment is commonly referred to as stack, we purposefully avoid the term stack, since it conotates a specific access protocol). The global environment will take the role of the data environment as we know it. The local environment will be shared by all functions. When a function is called, part of the local environment is allocated as a temporary environment in which variables corresponding to formal parameters and variables declared in the function body are created and in which the function body is evaluated. Upon function termination, the local environment reclaims the temporary environment allocated to the function. Thus, one and the same environment space is reused by allocating it to different functions executing at different times.

Definition 4: Variables, and computational objects in general, declared at the top-most level are said to have *global scope.* The programmer can rely on their existence over the entire lifetime of the program. Variables, and computational objects in general, declared in functions are said to have *local scope*. Their lifetime is restricted to the time span during which the function body is executed.

An alternative way to define a function in JavaScript is to use a variable declaration:

```
var area = function(x) {
   var pi = 3.14; //
   return pi * x * x;
```

```
}
```

The expression to the right of the assignment operator is called a *function literal*. It is used to create an unnamed function (Lisp lambda functions). This form of defining a function shows students that function definition is hardly different from variable declaration. It associates a name with type and value. In this case, the type is function and the value is the code comprising the function definition. Let the environment `d` be `[(z,number,3),(y,number,2)]`. The content of the environment after the definition of function `area()` is shown in Figure 4.

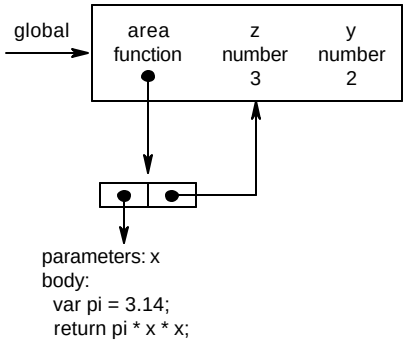

Figure 4: Function definition.

The value associated with the name `area` is a reference to an object consisting of two pointers. The left pointer points to the code of the function definition. The right pointer points to the global environment in which the function definition was read to produce the function. When the function is called, e.g., `z = area(y);`, a new local frame, `e1`, is created as shown in Figure 5. In this frame, the formal parameters are bound to the actual values with which the function has been called. Observe that the enclosing environment of the new frame is the environment to which the pointer in the function definition points, i.e., `global`. After the function terminates, the local frame pointed to by `e1` is removed, i.e., the memory is released and can be allocated to another function.

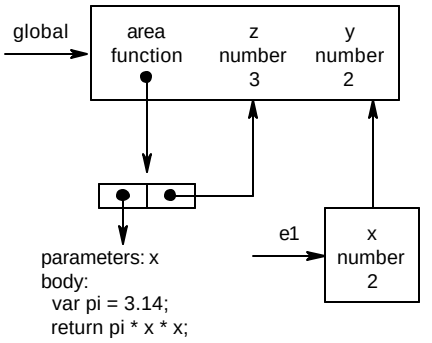

Figure 5: Evaluating the call to area().

We need to revisit the variable evaluation rule in the light of global and local frames. The value of a variable `x` in an environment `d` is the value of the first variable in the first frame in `d` whose name is `x`. If no frame in the environment `d` contains a program variable with the name `x`, then the value of `x` is undefined.

After functions as computational objects, expressing named compound computations, are covered, collections and objects as software analogs of real-world objects are introduced. We try to strike a balance and not to delve into technical details about JavaScript. The exercises given are mainly to test students' skills in applying the material rather than asking students to synthesize programs. Synthesis is a higher-order cognitive skill. We ask the students to do synthesis only in the second half of the course that covers Web applications development. By the end of the first part of the course, students should be able to apply (almost mechanically) the environment model of evaluation to predict the behavior of a program of any size. Tasks on object-oriented problems are restricted to using the `String`, `Array`, `Math` and `Date` objects and their respective methods.

Evaluation and Observations

As indicated before, introduction to programming in JavaScript is part of our E-business applications development course. The ultimate aim of the course is to teach students how e-commerce applications operate and how such applications are built. We do not evaluate directly the results of the model-based introduction to programming. The real test is how students can cope with implementing Web applications' business logic in JavaScript. Due to the almost mechanical application of the environment model of evaluation, students quickly develop skills in solving well-defined problems. The real challenge, and

therefore the ultimate test, is to "integrate the parts into a new more complex whole" (Huba, 2000), as it is required by the application logic of Web-based information systems. We assume that business logic is executed only on an application server (embedded in a Web server). In the second half of the course, we enriched gradually core JavaScript by introducing the ASP server-side extension of JavaScript. This API consists of the following basic classes: `Request`, `Response`, `Session` and `Application`. Students were asked to create applications from all major e-commerce models: business-to-customer (e.g. e-store and ticket booking); customer-to-customer (e.g. auction); peer-to-peer (e.g. chat room); business-to-business (integrate services, e.g. integration of a payment service, like PayPal, with an e-store using a virtual shopping cart).

After adopting the model-based approach, students' performance proved to be better than in previous runs of the course. At this point, we cannot prove in absolute terms the success of the model-based introduction to programming. We will return to this topic in the conclusion, where we discuss our plans for future work. Our observations at this stage of research suggest that the results are very encouraging. Table 3 summarizes the statistics for two classes. The first class had been exposed to a traditional code-based approach to teaching introduction to programming, while the second one had been exposed to the model-based approach presented above. More data will be available at the end of fall'02 semester when two more sections will have completed the new course. It is important to note that the number of students in the new course who achieved B and above (on the scale from A to F, with A being excellent) exceeds by approximately 20% the corresponding number of students in fall'01. The instructor's evaluation using standard IDEA forms (nationally normalized instrument for student evaluations of instruction used across campuses in US) was above average compared to average in previous years. This is a very good result for a new course. Note also the difference in the failing rate of the two classes.

	Fall'01 code-based	Sping'02 model-based
Number of students	41	21
Grade B and above	52%	71%
Failing grades	5%	0%

Table 3: Statistics on E-business applications development.

Conclusion

This paper presented a new model-based approach to teaching introduction to programming in JavaScript to general business students. The basic programming constructs and their semantics are introduced using the environment model of evaluation. Other modeling languages employed in the course are statecharts and UML class diagrams. The course material is structured in compliance with Bloom's model of cognitive skills. The first, by no means conclusive, results from the incorporation of the new approach are positive and very encouraging. The scholastic performance, the student evaluations, and our experiential observations suggest that the course has been significantly improved owing to incorporation of the novel elements presented in this work. Students have acquired a deeper understanding of the notion of computation process. As a result, learners have been able to apply creatively the accumulated knowledge in real-world applications, i.e., the acquired knowledge is active, discriminating and critical.

The teaching approach presented in this paper is part of a larger project aiming at teaching general business students how e-commerce systems operate and how Web-based applications are developed using modeling. The ultimate goal of the E-business applications development course is for learners to realize the importance of the software architecture, to see clearly the relations between user requirements and software architecture, and to understand the interconnectedness between software architecture and program code. The desire is to help learners acquire skills and ability successfully to develop robust software architectures using a subset of UML based on the Web modeler profile (Conallen, 1999).

To prove the advantages of the model-based approach and to generalize our results, we plan a statistical test once we collect and process the data from two classes currently taking E-business applications development. We will test a hypothesis concerning the parameters of a multiple regression model. The independent variables of the model will be verbal SAT (scholastic achievement test), math SAT, GPA (grade point average), and teaching method. We will control by SAT and GPA.

Acknowledgements

We would like to thank the teaching assistants Scott McQuiggan and Theran Mossholder for their invaluable help. Without their dedication the course would have been impossible.

References

Abelson, H. and Sussman, G. J., with Sussman, J. (1996). *Structure and interpretation of computer programs*. MIT Press.

Baber, R. (1987). *The spine of software: Designing provably correct software - theory and practice*. Chichester: John Wiley & Sons.

Bloom, B. S. (ed.) (1956). *Taxonomy of educational objectives: Book 1 Cognitive Domain*. London: Longman.

Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The unified modeling language.* Addison-Wesley.

Conallen, J. (1999). Modeling web application architecture with UML. *Communications of ACM*, 42(10).

Davis, M. (1988). A comparison of techniques for the specification of external system behavior. *Communications of ACM*, 31(9).

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8.

Huba, M. E. & Freed, J. E. (2000). *Learner-centered assessment on college campuses.* Allyn and Bacon.

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The Unified Software Development Process.* Addison-Wesley.

ScriptEase (2003). *JavaScript interpreter*. Retrieved from http://www.nombas.com.

Shaw, M. (2000). Software engineering education: A roadmap. *22nd Int'l Conference on Software Engineering*, Limerick, Ireland.

Biography

Borislav Roussev is an Assistant Professor of Information Systems at Susquehanna University. He was educated in Bulgaria, and previously was a faculty member in higher education in South Africa.