# A Laboratory Computer Science Course for Liberal Arts Students

John E. Howland
Department of Computer Science
Trinity University
715 Stadium Drive
San Antonio, Texas 78212-7200
Voice: (210) 736-7480
Fax: (210) 736-7477
Internet: jhowland@ariel.cs.trinity.edu

## Subject Areas

Computer Science Education, Scheme, Arts and Humanities, Exposition.

## Abstract

A new laboratory course in computer science for liberal arts students has been developed by the author. This breadth-first course covers 13 topics in computer science in a lecture course, while a co-requisite laboratory course provides lab experience with 13 prepared experiments. The Scheme programming notation is used consistently in lecture and laboratory as a lingua franca for computer science. Students are not taught Scheme programming, but rather, learn just enough Scheme syntax and semantics to be able to read and understand programs written by others. Working Scheme models for each computer science topic are presented in the lecture course and are studied experimentally in the laboratory course. The development of this course and laboratory was funded by the Meadows Foundation and NSF grant DUE 9452050.

**Keywords:** CS laboratory course, Scheme, exposition.

## 1. Introduction

At my university there was a need for new laboratory science courses which students could use to partially satisfy the science requirement in each student's common curriculum. Each student is required to complete two or three science courses. Two courses suffice when one of the two courses is a laboratory science course, so there is some incentive to take a laboratory course because the common curriculum requirement is reduced by one course. Traditionally, laboratory science courses had been selected from the natural science disciplines of biology, geology, physics or chemistry. The author developed a new introductory computer science laboratory course (and co-requisite lecture course) in which students are gathered together in a workstation laboratory at the same time to work in pairs performing a prepared laboratory experiment.

One feature of this course is that it covers a variety of computer science topics at about the rate of one new topic per week. Consequently, only an introduction to each topic can be presented during the three lectures on each topic. As a result, the course emphasizes a breadth of understanding at the expense of depth of understanding of any single topic.

Another feature of this course is that, while it uses the Scheme programming notation extensively to describe and model each topic, the course does not attempt to teach students to become effective programmers. Students are taught just enough Scheme syntax and semantics to be able to read and understand Scheme expressions and programs written by others. Such programs form the basis of models of various computational structures such as computer circuits, arithmetic units, data structures, processors, processes, etc. Since these models are expressed in Scheme, the models can be executed on a workstation and form the basis of laboratory experimentation in the course.

## 2. Scheme as an Expository Language for Computer Science

The philosophical motivation for the choice of Scheme as an expository notation for computer science is derived from its natural language sentence-like syntax [Kon 94], [Rie 93]. Since the majority of the students enrolling in this course are likely to be fine arts and humanities students who are generally less oriented towards technology and science, it was thought to be useful to use a notation which could be easily related to the structure and form of natural language which each student already has mastered.

## 3. Lecture Topics

The lecture course consists of about three lectures on each of the following topics:

1. Introduction to reading the Scheme notation
2. Computer organization
3. Computer arithmetic
4. Computer circuits
5. Algorithms
6. Data structures
7. Programming methodology
8. Software engineering
9. Language translation
10. Program execution time
11. Computer networks
12. Parallel processing
13. Computability
14. Artificial intelligence

Covering this many topics in a three credit hour course requires adhering rather rigidly to a schedule. Fortunately, many of the topics are inter-related so that knowledge gained in one topic is immediately used in a following topic. For example, the logical organization of a computer is expanded on when discussing computer arithmetic. The modeling of computer circuits fills in some detail purposely left out of the computer organization topic.

This fast paced presentation of topics leaves little time for discussion of topics in the lecture setting. Some discussion of each topic is done during the weekly meetings of the laboratory course. Recently, an experiment involving required out of class discussion of topics on a local USENET discussion group for the class, has been started. Each student is required to contribute one new discussion thread each week and respond to two or three threads per week. The news group is not moderated, but is archived. Other members of the university community have been invited to read and/or join the discussion of these topics.

A set of lecture notes has been constructed which use Scheme as an expository notation for computer science topics. Each topic has one or more Scheme based descriptive models which is human readable by the students and executable by the laboratory workstation. These models form the basis of laboratory experimentation as well as providing an interactive working model of the topic under discussion so that students may obtain hands-on experience with each topic.

The lecture notes are available to students via a world wide web server and the instructor can use the www presentation during lecture presentations. A discussion of how this is accomplished is presented in Section 5.

## 4. Laboratory Experiments

The laboratory course is one credit hour which meets once a week for three hours. Students work in pairs to perform a prepared laboratory experiment. Each pair of lab students has its own laboratory workstation where the experimental work is performed. Currently, the laboratory course consists of twelve experiments. All experiments, except the first, require the students to formulate a conjecture to be verified, gather experiment data, analyze the results and write a brief laboratory report which must be turned in at the beginning of the next laboratory class period. With each offering of the course we have attempted to modify and improve the experiments as well as devise new experiments.

The following are representative titles and purpose of experiments used in the course.

1. Getting Started with the Scheme Notation.

   In this lab, students familiarize themselves with the lab workstations, editor and Scheme notation. No lab report is required for this lab session.

2. Using Computer Science Department Laboratory Facilities.

   In this lab, students learn the format of a lab report and perform the first simple experiment which is to determine how fast the workstation can add numbers. Students are introduced to the problem of experimental sampling.

3. How Fast Do Computers Perform Arithmetic.

   The purpose of this laboratory experiment is to determine the relative performance of different arithmetic types on a lab workstation.

4. Designing and Verifying a 4 bit Binary Adder.

   The purpose of this laboratory is to build a working model of a 4 bit binary adder from modeled circuit elements and verify its correct operation.

5. Implementation of an Algorithm.

   This experiment involves the experimental estimation of the time required to evaluate (fibonacci 100) using a recursive implementation of the fibonacci function. This experiment requires an understanding of recursive and iterative implementations of the fibonacci function.

6. Choosing a Data Structure.

   The purpose of this laboratory experiment is to examine and compare two implementations of an abstract data structure for a stack. The first implementation uses functions to implement a constructor, predicates and accessors for stacks. The second uses an object oriented approach to implement stacks. Most students are surprised by the outcome of this experiment since they find their conjecture incorrect.

7. Programming Methodology.

In this laboratory problem students are given three different implementations of a system for performing exact rational arithmetic. The implementation has been carefully designed and layered so that operations are separated from data representations by using abstract constructors and accessors. They are asked to predict relative performance of each system, comment on the quantitative aspects of each implementation, gather experimental data and draw conclusions in a written laboratory report.

8. Software Prototypes.

In this laboratory problem students work with a prototype implementation of rational arithmetic operations which allow rational numbers to be combined with other types of numbers in arithmetic expressions. You are asked to evaluate the performance of the prototype implementation and write laboratory report.

9. Recognizing Syntactic Elements of a Language.

In this laboratory experiment, students are asked to design a recognizer for a syntactic element of the Scheme notation, given the BNF syntax description.

10. Recursive Processes, Iterative Processes and Compiling.

In this laboratory problem students analyze a function which generates recursive process then consider a recursive version of the same function which generates an iterative process and finally examine the compiler output and analyze the performance of both functions.

11. Evaluating Parallel Performance.

The purpose of this laboratory problem is to evaluate the performance of a network computation in which two networked workstations cooperate to perform a calculation in parallel.

12. Expert Systems: A Rule Interpreter.

In this laboratory problem students will try to discover the behavior of a rule system by tracing the execution of that rule based system as it interprets different rule sets.

This course is the first course in our computer science department which attempts to utilize the more traditional contained science laboratory approach to teach computer science concepts. The effort required to design a successful laboratory experiment is much higher than anticipated, but the benefits can be rewarding in that successful experiments seem to convey the concept being taught more efficiently and forcefully to the student than the more traditional kinds of programming problems we have assigned in other introductory courses. Our department plans to introduce contained laboratories for other introductory computer science courses as a result of our experience with this course.

## 5. Laboratory Hardware

The Scheme notation plays a fundamental role both in the exposition of computer science topics and in the laboratory experiments. Because of this, an effort was made to design a laboratory facility which could be used for lecture exposition as well as laboratory experimentation.

A recent Scheme frequently asked questions document for the USENET discussion group comp.lang.scheme lists 23 Scheme systems. Of these, only three Scheme systems are commercial products. All other Scheme systems are available, without charge, from a variety of internet sources. We currently use Aubry Jaffer's scm system which can be built to run on PC's, Mac's and UNIX systems. Recently, scm was selected by the Free Software Foundation to become the standard GNU extension language for all GNU software products. For a number of reasons, including using this laboratory/teaching facility for a variety of other courses, we made a decision to base this lab on the UNIX operating system.

Grant proposals to the Meadows Foundation and the National Science Foundation (Grant # DUE-9452050) were prepared and both groups agreed to fund 50% of the cost of laboratory equipment. A vendor competition involving Apple, IBM, Sun, SGI and HP was designed which involved running certain Scheme based benchmarks and meeting a color graphics requirement of at least 8 bits per pixel on a display of at least 1024 by 768 pixels. Each vendor also had to meet other requirements involving memory capacity, UNIX operating system, disk capacity and networking.

One additional requirement, to be used in the lecture course, was the ability for a lecturer to be able to use a machine for presentations and demonstrations and have these presentations be visible on the screens of each of the student workstations. We proposed that this could be accomplished either by using a video amplifier/switching system fed by the instructor's workstation monitor or a network based software video feed.

HP won the vendor competition with 17 HP 712 machines of varying configuration. There are 15 HP 712/60 16M student workstations, 1 HP 712/60 32M instructor workstation and 1 HP 712/80 32M server machine providing passwords and home directories to each student workstation. HP also supplied an X windows based software package, SharedX, which allows the instructor machine to share any of its windows with any of the student lab workstations. In addition, SharedX also allows the instructor to turn over control of any of its windows to any of the student workstations. This means that students can, at the discretion of the instructor, provide responses to instructor queries which can be seen at all of the other student workstations. We are conducting a variety of experiments on how to use this facility in a lecture and lab environment.

When used either as a lecture room or as a lab room we seat two students in front of each workstation. This limits class size to 30 students per section which is an appropriate maximum size for this kind of course.

## 6. Examples of Scheme in Exposition

A small subset of Scheme suffices when used as an expository notation in a course like this. Students first learn the rule for evaluating function expressions and then learn a few exceptions (special forms) to the general rule. Specifically, `define`, `quote`, `lambda`, `if`, `and`, `or`, `begin` and `let`. In the following sections a few examples of the expository use of Scheme are given.

The circuit elements and, or and negate can be modeled as:

```
(define bit-or
  (lambda (x y)
    (if (or (= x 1) (= y 1))
      1
      0)))

(define bit-and
  (lambda (x y)
    (if (and (= x 1) (= y 1))
      1
      0)))

(define bit-not
  (lambda (x)
    (if (= x 0)
      1
      0)))
```

A half-adder can be modeled using these elements as:

```
  (define bit-half-adder
    (lambda (a b)
      (bit-or
       (bit-and a (bit-not b))
       (bit-and b (bit-not a)))))
```

Next we can build a 1 bit adder using two half-adders as:

```
  (define bit-adder
    (lambda (a b cin)
      (let* ((t (bit-half-adder a b))
             (g (bit-and a b))
             (p (bit-and t cin)))
        (list (bit-or g p)
              (bit-half-adder t cin)))))
```

Notice that `let*` is used in this description. If `let*` is not in the student's current vocabulary of special words, then it would be introduced at this point. The idea is that certain language features are introduced when there is an expository need for them. At this point in the course a student would know that `let` is syntactic sugar for the evaluation of a particular `lambda` expression. Since the arguments to a function are evaluated in some unspecified order, an ordinary `let` cannot be used in `bit-adder`; hence the need for `let*`.

Next we can model a wire for connecting circuit elements as:

```
  (define wire-output
    (lambda (pin-number outputs)
      (list-ref outputs pin-number)))
```

We can use two bit-adders and some wire to build a 2 bit adder as:

```
  (define 2-bit-adder
    (lambda (a1 a0 b1 b0)
      (let* ((t0 (bit-adder a0 b0 0))
             (t1 (bit-adder
                  a1
                  b1
                  (wire-output 0 t0))))
            (list (wire-output 0 t1)
                  (wire-output 1 t1)
                  (wire-output 1 t0)))))
```

It needs to be emphasized that while students find such descriptions readable, they may also be used to provide an interactive working model which can be explored during a lecture using the classroom workstation and display system. These descriptions also constitute part of the prepared laboratory software for simple experiments involving computer circuit elements. In one laboratory experiment, students are asked to design and build a model of a 4-bit adder. The lab suggests that they verify experimentally that this adder performs 2's complement arithmetic and asks them to suggest a modification of their design which would allow the adder to subtract as well as add. The above models for adder circuits can be installed in a model of a simple CPU which is used in the lectures describing the organization of a simple computer.

In the lectures on software design and engineering principles, when discussing techniques of modularization and layering of software, the routines for rational arithmetic found in *Structure and Interpretation of Computer Programs* [Abel 85] have been used. One can demonstrate that it is possible to change the implementation of rational numbers without changing any code in layers above the implementation layer. The benefits of abstraction barriers is explored experimentally in a lab which asks students to form hypotheses concerning three different rational number implementations (don't remove common factors from numerator and denominator, remove common factors at construction and remove common factors at access), gather data, analyze results and draw conclusions. This experimentation may be done with respect to speed or space used.

In the lectures on the topic of programming language translation, we describe the syntax of a tiny language given by the BNF rules:

```
<exp> ::= <var-ref> | (lambda (<var>) <exp>)
      | (<exp> <exp>)
```

Since this language is a subset of Scheme, it is easy to build Scheme functions which model the process of recognizing whether or not an expression is a syntactically correct expression in this language.

```
(define exp?
  (lambda (exp)
    (if (symbol? exp)
      #t
      (if (and
           (pair? exp)
           (= 2 (length exp))
           (exp? (car exp))
           (exp? (cadr exp)))
        #t
        (if (and
             (pair? exp)
             (= 3 (length exp))
             (eq? 'lambda (car exp))
             (= 1 (length (cadr exp)))
             (symbol? (caadr exp))
             (exp? (caddr exp)))
          #t
          #f)))))
```

In the discussion of semantics for this language we see that this language supports variable reference, definition of functions of one argument and function call of functions of one argument. It is also easy to build Scheme functions which identify free and bound variables in this language. A predicate for free variables might be written as:

```
(define free?
  (lambda (var exp)
    (if (and
         (symbol? exp)
         (eq? var exp))
      #t
      (if (and
           (pair? exp)
           (= 2 (length exp))
           (exp? (car exp))
           (exp? (cadr exp))
           (or
            (free? var (car exp))
            (free? var (cadr exp))))
        #t
        (if (and
             (pair? exp)
             (= 3 (length exp))
             (eq? 'lambda (car exp))
             (= 1 (length (cadr exp)))
             (symbol? (caadr exp))
             (exp? (caddr exp))
             (not (eq?
                    var
                    (caadr exp)))
             (free? var (caddr exp)))
          #t
          #f)))))
```

Here, again, the point of such descriptions, as given above, is that they are readable by humans and each such description is also a working model of what is being discussed which can be explored in the laboratory.

## 7. Student Response to the Course

Generally, student response to this course has been enthusiastic. They are relieved that they can learn computer science concepts and have hands on experience conducting experiments of various types without learning to write their own programs. Even though learning to program is not one of the course goals, many of the students find that they can write their own simple programs and seem to be pleased with this knowledge even though most of them will never find an occasion to program computers later in life.

The first few offerings of this course occurred before the acquisition of the laboratory equipment described in Section 5. The course was run in a makeshift lab of borrowed, out of date, Sun and Apple UNIX workstations. The lab workstations were under powered for some of the experiments and presented different user interfaces at student workstations. Student response to laboratory experiments is much more positive with the new HP workstations.

## 8. Distribution of Course Materials

Preliminary versions of the course notes and laboratory experiments are available via the department's web server, `http://www.cs.trinity.edu/About/The_Courses`. Students can use a web browser such as NCSA Mosaic or NetScape to view these materials. The instructor can share a Mosaic window on his machine with each student workstation and use Mosaic as a presentation program. Expository Scheme can be copied from the web browser presentation window and pasted into and instructor's XTerm window running a Scheme interpreter which is also shared with the student workstations to provide interactive demonstrations of a working Scheme model.

The course materials are also freely available to anyone else via the internet. Other distribution of course materials are available by contacting the author.

## 9. References

[Abel 85] Abelson, Harold and Sussman, Gerald with Sussman, Julie. *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

[Frie 92] Friedman, Daniel, Wand, Mitchell and Haynes, Christopher. *Essentials of Programming Languages*, MIT Press, 1992.

[Kon 94] Konstam, Aaron and Howland, John, "Teaching Computer Science Principles to Liberal Arts Students Using Scheme", SIGCSE Bulletin 26 (4), December 1994.

[Rie 93] Riehl, Arthur, moderator, "Using Scheme in the Introductory Computer Science Curriculum", Panel, SIGCSE Bulletin 25 (1), March 1993.

[Spri 89] Springer, George and Friedman, Daniel. *Scheme and the Art of Programming*, MIT Press, 1989.