A Declarative Laboratory Approach for Discrete Structures, Logic, and Computability

James L. Hein
Department of Computer Science
Portland State University
Portland, OR 97207-0751
jhein@cs.pdx.edu

Overview

Many students find it hard to grasp and retain the ideas presented in courses covering discrete structures, logic, and computability. These subjects provide a foundation for required upper division courses in computer science. Therefore a major effort must be made to improve the learning environment for students studying these ideas at the lower division level.

Many of us succeeded academically in spite of the way we were taught. But how many people have not succeeded because of the way material was presented to them? Since people learn in different ways, it makes sense to present students with a variety of learning experiences.

We have created a laboratory component for a year long sophomore course in discrete structures, logic, and computability for students majoring in computer science or computer engineering. The labs consist of experiments in declarative programming environments. The experiments are designed to reinforce the learning of material on a daily basis, just like the regular homework assignments. In other words, the lab experiments are short in duration and relevant to the material covered by each lecture.

Short programming labs that correspond to each lecture should be useful learning tools for many traditional courses. The instant feedback that students get from wrong assumptions can give them incentive to try something new - to experiment and see what happens. The lab component can also encourage the use of laboratory partners, interaction of students, team presentations, and lively classes. Programming environments as lab tools can be as important to learning as the traditional tools found in science and engineering labs.

We are not talking about the well defined ideas of Computer Aided Instruction and Intelligent Tutoring Systems, where students interact with a program for a specific domain of knowledge. We are talking about using declarative programming languages as laboratory tools, to be used experimentally by students to learn new ideas. The key is to give students traditional laboratory work, where the experiments are well defined and relevant to the lecture material.

The instant feedback obtained from an interpreter, in response to a declarative solution to a problem, is a powerful incentive to try to get it right. Ideally a student should be able to complete a laboratory experiment during one lab period, just like a traditional science laboratory.

Implementation

Laboratory courses need more credit. The learning that takes place by writing down solutions to problems is not in question [7]. Thus we don't wish to reduce the written homework in these courses. We intend to supplement this learning with a laboratory component. Therefore the credit must be increased to reflect the extra work.

The Department of Computer Science at Portland State University has implemented, as part of a curriculum change, a laboratory component for a new year long sophomore course in discrete

structures, logic, and computability. The course is actually a sequence of three 10 week courses, where each course is four credits consisting of three credits of lecture and one credit of lab. The three courses cover the following material:

Discrete Structures
Introduction to notations and techniques to represent and analyze computational objects. Sets, bags, and tuples. Properties of functions. Construction of sets, languages, and recursively defined functions. Equivalence and order relations. Inductive proof techniques. Elementary combinatorics. Programming problems introduce use of a functional language.

Logical Structures
Introduction to logic from a computational viewpoint. Propositional calculus, first order predicate calculus, formal reasoning. Resolution and natural deduction. Applications to program corrrectness and automatic reasoning. Proof techniques. Programming problems introduce use of a logical language.

Computational Structures
Elementary algebraic structures, Boolean algebra. Regular languages and finite automata. Context-free languages and pushdown automata. Automata as computation devices. Turing machines. Chomsky language hierarchy. Church's thesis, computation models and their equivalence. Solvability and unsolvability. Use of a declarative language.

Lab Environment

The programming language chosen for the laboratory experiments is a very important consideration. A language must be easy to learn so it does not distract from the learning goals of the lab experiments. The environment must be able to provide instant feedback for testing assumptions and correcting mistakes. This will allow students to rapidly complete short experiments that reinforce the daily class material. Declarative programming languages satisfy these requirements. For example, declarative programming languages have been shown to be good tools for learning recursion [4].

From the several declarative languages that are available, we have been using the languages FP, ML, and Prolog. All three have syntax and semantics similar to the mathematical and logical notations used to present the ideas of discrete structures, logic, and computability. So the learning curve can be steep. For example, the FP language is a powerful tool to explore composition of functions. Its non-variable nature can be used as a tool for laboratories that encourage students to think about combining simple functions to create more complicated functions. The ML language reflects the natural way we write definitions of functions. The type inference mechanism in ML can be used as an experimental tool for a laboratory to learn about domains and codomains of functions.

Logic is becoming an important part of the computer science curriculum [6]. Since we have a full term course in logical structures, it makes sense to use Prolog as a learning tool. Since the language can be introduced from a relational viewpoint, lab experiments concentrating on the propositional calculus can be given before students have been introduced to the predicate calculus. Thus when predicate calculus is discussed, students can see Prolog as an automated predicate calculus. Similarly, when resolution is discussed, students already have seen its use in the Prolog computation rule. Information on using these languages can be found in [1], [2], and [3].

Conclusion

Since people learn in different ways, it is essential that we provide them with as many learning paths as we can. We must not restrict a student's learning by the methods we use to teach. The subjects of discrete structures, logic, and computability are normally taught with paper and pencil type homework. The instant feedback provided by declarative programming labs can speed the process of understanding, when compared to the slow feedback caused by the time it takes to grade and return homework papers. For those students who can learn the ideas in the traditional

way, the laboratory will provide new ways to think about those ideas. But for other students the laboratory approach might be the difference between success and failure. Also, paper and pencil homework can take on a new dimension. For example, students will obtain valuable writing experience by writing summaries of the lab experiments [4].

It must be emphasized that the structure and content of the laboratories is the key to success or failure of this learning method. Students have asked for laboratory projects to be easier, shorter, and more relevant to the lecture material, just like the daily homework. In other words, the lab should not be a traditional programming language lab. The languages chosen must be easy to learn and reflect the notation used in the courses. As an added benefit, students will be exposed in a natural way to new programming languages, and not become addicted to one particular language.

We now have a crop of declarative languages that are easy to learn and use because they emulate the natural notations of logic and mathematics [4] and [5]. With carefully constructed lab experiments, these languages could be used in many courses where mathematics and logic are used. For example, a new kind of introductory course may be possible, with the laboratory experiments serving as a first introduction to programming.

Sample Laboratories

Our goal is to build a portfolio of sixty or more labs - two per week for thirty weeks of class. We have constructed 25 labs that satisfy our criteria. But much work is yet to be done, testing and modifying existing labs, and creating new labs. The following samples are representative of the existing labs. Some functions and predicates are locally defined.

Power Set Lab: We can use the FP interpreter to compute the power set. For example, the power set of $\{a, b\}$ is obtained by typing the command

power: <*a, b*>

The interpreter will respond with the answer

<<>, <*a*>, <*b*>, <*a, b*>>

which represents the power set $\{ \varnothing, \{a\}, \{b\}, \{a, b\}\}$.

For each of the following sets compute the power set by hand, and then use FP to check your answer.

a. $\{x, y, z\}$. b. $\varnothing$. c. $\{a, \{a, b\}\}$.
d. $\{\varnothing\}$ e. $\{\{a\}, \varnothing\}$. f. $\{a, \{a\}, \{\{a\}\}\}$.

Bags (Multisets) Lab: The bag $[a, a, b]$ is a subbag of $[a, a, a, b, b]$. This can be verified by typing the FP expression

subbag: <<*a, a, b*>, <*a, a, a, b, b*>>

The interpreter returns the value t. The union of the two bags $[a, a, b]$ and $[a, a, b, b, c]$ is $[a, a, b, b, c]$ and it can be computed by typing the FP expression

bagUnion: <<*a, a, b*>, <*a, a, b, b, c*>>

The interpreter returns <*a a b b c*>. The intersection of the same two bags is $[a, a, b]$ and it can be computed by typing the FP expression

bagIntersect: <<*a, a, b*>, <*a, a, b, b, c*>>

The interpreter returns <*a a b*>. For each pair of bags find whether the first is a subbag of the second, and also find the union and intersection of the pair. First do each problem by hand, and then use FP to check your answer.

a. $[x, y]$ and $[x, y, z]$.
b. $[x, y, x]$ and $[y, x, y, x]$.
c. $[a, a, a, b]$ and $[a, a, b, b, c]$.
d. [1, 2, 2, 3, 3, 4, 4] and [2, 3, 3, 4, 5].
e. $[x, x, [a, a], [a, a]]$ and $[a, a, x, x]$.
f. $[a, a, [b, b], [a, [b]]]$ and $[a, a, [b], [b]]$.

Combining Functions Lab: We want to construct the function f defined by

$f(n)$ = <<0, 0>, <1, 1>, ..., <*n, n*>>.

We can use the existing functions *seq* and *pairs* to define $f(n)$ = pairs(seq(n), seq(n)). This definition can be translated into FP by typing the expression

{*f* pairs @ [seq, seq]}

The interpreter will respond with {*f*}, indicating that it has received the definition.

1. Use FP to test the function *f*.
2. Translate the following function into FP and

test a few values: $g(n)$ = dist(n, seq(n)). Also give an informal description of g.

Recursively Defined Functions Lab: Suppose we have a function g defined in if-then-else fashion as follows:

$g(x)$ = if $a(x)$ then $b(x)$ else $c(x)$.

We can translate this definition into FP by typing the expression

$(g \ \ (a \rightarrow b; c))$

1. Translate each function into FP and test it on a few values. Use the trace command for at least one test.
   a. $f(n)$ = if $n = 0$ then 3 else $n + f(n - 1)$.
   b. back(n) = if $n = 0$ then <0> else cons(n, back(n - 1)).
2. Construct a recursively defined function to add up the numbers in a list. Translate your function into FP and test it.

Type Inference Lab: Figure out the type of each function. Then use ML to check your answer.

a. f(x) = x + 4.0.
b. $f(x) = x$.
c. g(x) = x div 4.
d. $f(x) = [x]$.
e. g(x) = hd(tl(x)).
f. $h(x, y)$ = if $x = 0$ then y else $[x]$.

Modus Ponens Lab. Try out the modus ponens inference rule with the following Prolog experiment. First, ask the question

|-? q.

Then enter the following fact into the Prolog database:

q :- p.

Now ask the question

|-? q.

Finally, add the following fact in the database:

p.

Then ask the question:

|-? q.

Question: Since the lab is about the modus ponens inference rule, what is a reasonable interpretation of a prolog statement like b :- a?

If..Then..Else Lab: We want to experiment with the statement "If c then x else y" where x and y are Boolean values. First, enter the following two facts in the database:

x :- c.
y :- not c.

Try out the following two questions:

|-? x.
|-? y.

Now enter the following fact in the database:

c.

Now ask the two questions again:

|-? x.
|-? y.

Make some observations and conclusions.

Quantification Lab. When we enter a statement containing a variable into the database, the statement is implicitly quantified by Prolog. Try the following experiment to discover the kind of quantification. Insert the following facts in the database:

$r(X)$.
$s(a)$.
$t(Z)$:- $r(Z)$, $s(Z)$.

Try to discover how each line is quantified. What about the scope of the quantifiers? To get an idea about things, try some questions, like,

|?- t(book).
|?- $t(a)$.
|-? r(hello).
|-? s(89).

Equality Axioms Lab: Suppose we have the following two axioms for an equality theory: $\forall x \ (x = x)$ and $t = u \wedge p(\ldots\ t\ \ldots) \rightarrow p(\ldots\ u\ \ldots)$. We can use these axioms to prove the following symmetric and transitive properties for terms:

$t = u \rightarrow u = t$

$t = u \wedge u = v \rightarrow t = v.$

For example, we have the following proof of the symmetric property:

1.	$t = u$	P
2.	$t = t$	EA Axiom
3.	$t = u \wedge t = t \rightarrow u = t$	EE Axiom
4.	$t = u \wedge t = t$	1, 2, Conj
5.	$u = t$	3, 4, MP
6.	$t = u \rightarrow u = t$	1, 5, CP

QED.

To see if Prolog validates this argument, enter the following facts in the database, where $e(a, b)$ means $a = b$.

```
e(X, X).
e(t, u).
e(u, t) :- e(t, u), e(t, t).
```

Now ask the following question:

```
|?- e(u, t).
```

1. Why does this prolog experiment verify the six line proof of symmetry?
2. Consider the following proof of transitivity:

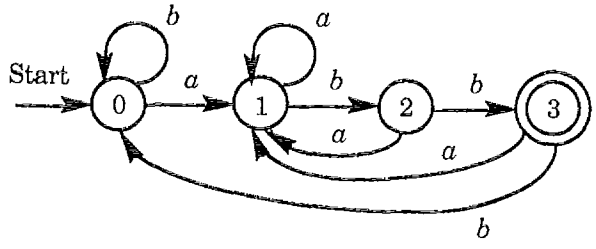
1.	$t = u$	P
2.	$u = v$	P
3.	$u = v \wedge t = u \rightarrow t = v$	EE Axiom
4.	$u = v \wedge t = u$	1, 2, Conj
5.	$t = v$	3, 4, MP
6.	$t = u \wedge u = v \rightarrow t = v$	1, 2, 5, CP

QED.

   a. Construct a Prolog experiment to verify this proof.

   b. Explain how your prolog experiment verifies the transitivity proof.

Finite Automata Lab: Calculate the transition function t for the following DFA:



Enter t in the Prolog database as a collection of facts having the following form:

```
t(state, letter, nextstate).
```

To indicate that state 3 is a final state, enter the fact

```
final(3).
```

To test whether the string *aaba* is accepted by the DFA, we write the string as a list of letters and type the following goal:

```
|?- accept([a, a, b, a]).
```

This action starts the execution of the following simple DFA interpreter:

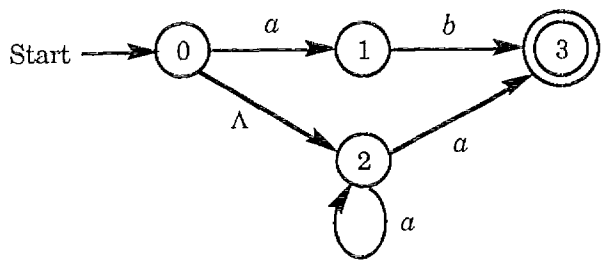
```
accept(S) :- path(0, S).
path(K, []) :- final(K).
path(K, [H|T]) :- t(K, H, N), path(N, T).
```

1. Perform the following experiments for the given DFA:
   a. Use the DFA interpreter to test five strings accepted by the DFA.
   b. Use the DFA interpreter to test five strings rejected by the DFA.
   c. What is the regular expression for the DFA?
2. Find a DFA for the regular expression

$$aa^*b + b(a + b).$$

   Then use the DFA interpreter to test the DFA on five strings that are accepted and five strings that are rejected.

Nondeterministic Finite Automata Lab: Calculate the transition function t for the following NFA:



Enter t in the Prolog database as a collection of facts of the form

t(state, symbol, nextstate).

For example, since $t(2, a) = \{2, 3\}$, there must be two facts corresponding to the non-determinism:

$t(2, a, 2)$.
$t(2, a, 3)$.

When there is a Λ edge like $t(0, \Lambda) = \{2\}$, write Λ as the empty list:

$t(0, [\,], 2)$.

Indicate a final state by a fact of the form

final(state).

To test whether the string *aaba* is accepted by the NFA, we write the string as a list of letters and type the following goal:

| ?- accept([*a*, *a*, *b*, *a*]).

This action starts the execution of the following NFA interpreter:

```
accept(S) :- path(0, S).
path(K, []) :- final(K).
path(K, [H | T]) :- t(K, H, N), path(N, T).
path(K, X) :- t(K, [], N), path(N, X).
```

1. Perform the following experiments for the given NFA:
   a. Use the NFA interpreter to test five strings accepted by the NFA.
   b. Use the NFA interpreter to test five strings rejected by the NFA.
   c. What is the regular expression for the NFA?

2. Find an NFA for the regular expression

   $$ab^* + bc^*.$$

   Then use the NFA interpreter to test the NFA on five strings that are accepted and five strings that are rejected.

Pushdown Automata Lab: Suppose we write a pushdown automaton as a set of Prolog facts of the following form:

t(state, letter, top, operation, nextstate).

final(state).

In order to write a simple interpreter for PDAs, we'll need to make a few assumptions. We will require that every PDA start in state 0. The stack is a list that is initialized with the value [*e*], which means *e* is the top of the "empty" stack. We'll reserve the letters *p* and *n* for the operations pop and nop, and we'll agree to let the push instruction be represented by the symbol that is to be pushed. For example, in the instruction $t(0, a, e, b, 1)$ the letter *b* means push*b*.

For example, a PDA to recognize the language $\{a^n b^n \mid n \geq 0\}$ can be written as the following set of facts:

```
t( 0, a, e, a, 0 ).
t( 0, a, a, a, 0 ).
t( 0, b, a, p, 1 ).
t( 0, [ ], e, n, 2 ).
t( 1, b, a, p, 1 ).
t( 1, [ ], e, n, 2 ).
final(2).
```

To check whether this PDA accepts the string *aabb*, we type the following goal:

| ?- accept([*a*, *a*, *b*, *b*]).

This action starts the execution of the PDA interpreter, which we will describe next.

Since the stack is necessary to the computation, we've added a third variable to the "path" predicate to carry along the stack. The predicate top(OldS, *T*) checks to see if the top of OldS is *T*. The predicate oper(OldS,*Y*, NewS) performs operation *Y* on the stack OldS and returns the new stack NewS. The interpreter can be written as follows:

```
accept(S) :- path(0, S, [e]).
path(K, [ ], Stack) :- final(K).
path(K, [H | T], Stack) :- t(K, H, A, O, M),
                           top(Stack, A),
                           oper(Stack, O, NewStack),
                           path(M, T, NewStack).
```

```
path(K, X, Stack) :- t(K, [ ], A, O, M),
                     top(Stack, A),
                     oper(Stack, O, NewStack),
                     path(M, X, NewStack).

top([H | T], H).

oper([H | T], p, T).
oper(X, n, X).
oper(X, A, [A | X]).
```

1. Test the example PDA by using the PDA interpreter to check acceptance of a few "strings" in the form of lists, such as

   [ ], [*a*, *a*, *a*], [*a*, *a*, *b*, *b*, *b*], [*b*, *b*, *b*], [*b*, *a*, *b*], etc.

2. Find a PDA for the language of all strings over {*a*, *b*} that have the same number of *a*'s and *b*'s. Test your solution with the PDA interpreter.

References

1. S. Baden, *Berkeley FP User's Manual, Rev. 4.1*, 1982.
2. M. Carlsson and J. Widen, *SICStus Prolog User's Manual*, 1990.
3. R. Harper, Introduction to Standard ML, *Technical Report 86-14*, Department of Computer Science, University of Edinburgh, 1986.
4. P. B. Henderson and F. J. Romero, Teaching Recursion as a Problem-Solving Tool Using Standard ML, *ACM SIGCSE Bulletin* 20, 1, 1989, 27-30.
5. R. A. Kowalski, Logic as a Computer Language for Children, in *New Horizons in Educational Computing*, M. Yazdani, Ed. Ellis Horwood, 1984.
6. J. P. Meyers, Jr. The Central Role of Mathematical Logic in Computer Science. *ACM SIGCSE Bulletin* 22, 1, 1990 22-26.
7. G. Polya, *How to solve it*, McGraw-Hill, Princeton, 1954.

*******OBERON Continued From Page 18********

7. CONCLUSION

(i) What is Oberon-2? Oberon-2 is an extension of Oberon which provides the so-called type-bound procedures. A type-bound procedure is very close to what traditionally is called a "method".

(ii) How to find Oberon implementations? The Oberon operating system, including a compiler, has been implemented on different machines, including IBM PCs and Mackintosh II computers. The executable codes and the sources can be obtained free via anonymous internet file transfer from the Institute of Computer Systems, Zurich, Switzerland.

FTP Hostname: neptune.inf.ethz.ch
Internet Address: 129.132.101.33
FTP Directory: Oberon

(iii) What to read on Oberon? Three books have been published in 1991 and 1992. The book [1] provides both a tutorial and a complete reference to the language Oberon. Another book [2] contains everything that is needed to use the Oberon operating system. Finally, the book [3] describes the implementation of the whole operating system, including the Oberon compiler. It contains almost all sources in Oberon except for those that are machine dependent.

An extended version of the present paper can be obtained from its author.

8. REFERENCES

1. Reiser, M., Wirth, N. Programming in Oberon. Steps beyond Pascal and Modula. Wokingham: Addison-Wesley (1992).
2. Reiser, M. The Oberon System: User Guide and Programmers Manual. Wokingham: Addison-Wesley (1991).
3. Wirth, N., Gutknecht, J. Project Oberon. Wokingham: Addison-Wesley (1992).