Exploiting Logic Program Schemata to Teach Recursive Programming

Timothy S. Gegg-Harrison

Department of Computer Science
Winona State University
Winona, MN 55987, USA
tsg@vax2.winona.msus.edu

Abstract. Recursion is a complex concept that most novice logic programmers have difficulty grasping. Problems associated with recursion are avoided in imperative languages where iteration is provided as an alternative to recursion. Although difficult to learn, recursion is very easy to use once it is understood. In fact, many problems that have straightforward recursive solutions have very sophisticated iterative solutions. Many of the difficulties associated with learning recursion can be overcome by incorporating *conditional recursion* (i.e., a structured recursive equivalent to the WHILE loop) into logic programming languages. Two popular instructional techniques are collaborative learning and situated learning. The underlying claim of the situated cognition movement is the desire to enculturate the student into the domain of the teacher by involving the student in a series of authentic activities which are designed to incrementally improve the skills of the student. In the domain of computer programming, collaboration is an authentic activity. In this paper, we present logic program templates and schemata which add conditional recursion to logic programming languages and enable collaborative logic programming. Conditional recursion also provides a bridge to higher-order programming. Higher-order programming is the essence of abstraction in problem solving. Thus, in addition to aiding its students in acquiring the knowledge of recursion, conditional recursion also promotes abstract problem solving skills. We have successfully employed this schema-based approach to teaching recursion in several declarative programming languages with much success over the past four years.

1 Introduction

The strength of logic programming languages is their duality of semantics. The declarative semantics of a logic programming language is based on logic, while the procedural semantics of a logic programming language is based on execution mechanisms (e.g., SLD-resolution + computation rule + search rule). Because the declarative and procedural semantics can be shown to be equivalent (at least for some classes of programs), it is possible for logic programmers to think less in terms of what processes the computer must go through and much more in terms of the logic of the

problem itself and its possible solution. Logic programming promotes thinking about *what* the problem is rather than *how* to solve the problem. Problem solving becomes a process that is independent of the particular machine. Programs become easier to write and easier to debug because their structures more closely resemble the problem that is being solved rather some representation of the procedure required to solve it.

Since logic programming languages embody a declarative programming style and their syntax is extremely simple, it would seem that they should be easy to learn and easy to use. However, logic programming languages require their programmers to define their programs recursively. Recursion is a very difficult concept at first, but once it is learned it becomes a very straightforward (and natural) problem solving technique. The trick is finding a method for representing common recursive control flow patterns. The solution is program schemata and programming techniques. Program schemata enable the creation of *conditional recursion* (i.e., a structured recursive equivalent to the WHILE loop). Conditional iteration imposes structure on iterative programming languages, abstracting the essence of conditional repetition. Conditional recursion can serve the same role for logic programming languages.

Logic program schemata have proven useful in teaching recursive logic programming to novices [22], debugging logic programs [25], transforming logic programs [18,20,27,37], and synthesizing logic programs [17]. Furthermore, using program schemata to teach programming facilitates instruction that is tailored to the student's capabilities [23]. Program schemata enable improved instruction for novice programmers while at the same time promoting a structured programming style and the acquisition of abstract problem solving skills. In addition to being helpful for novice programmers, program schemata are essential to expert programmers. The key difference between experts and novices is not the size of their memory span, but rather their ability to chunk information together into meaningful units. Schemata provide a method of organizing meaningful information about complex domains. Experts have more and better problem schemata than novices. Novice programmers tend to categorize problems based on surface syntax-based features of the problem statement, while experts categorized problems with respect to a more abstract hierarchical organization of algorithms [1,36]. Thus, program schemata are essential to expert programmers.

By abstracting out common recursive control flow patterns, program schemata capture large classes of logic programs. Programming techniques represent common program components. By instantiating portions of program schemata with programming techniques, it is possible to generate arbitrary logic programs. In order to represent program schemata for any programming language, it is desirable to use a higher-order programming language as the representation language. Functional programming languages (e.g., Lisp, ML, Miranda, etc.) support higher-order functions. Most logic programming languages, on the other hand, do not have full support of higher-order predicates. Prolog supports first-order Horn clauses with only limited higher-order features. λProlog is a higher-order logic programming language that extends Prolog by incorporating higher-order unification and λ-terms [30]. Because of its support of higher-order Horn clauses [31], λProlog makes an excellent logic programming language

for representing logic program schemata [26].

In this paper, we present a set of logic program schemata and show how to use them to promote a structured approach to teaching logic programming. Our approach to teaching recursive programming is to use logic program *templates* to help guide the student to produce a correct program by providing her with the basic structure for the program. Using templates as an instructional aid has an important side effect of promoting the development of abstract problem solving skills which are born out as logic program *schemata*. Thus, program templates help novices learn to program while at the same time encourage the development and acquisition of program schemata which are the basis of abstract problem solving.

2 The λProlog Language

The basic syntactic conventions of λProlog are the same as those of Prolog: all legal statements must be in clausal form where `:-` represents implication, the comma represents conjunction, the semicolon represents disjunction, cut is represented by the exclamation mark, and identifiers that begin with an uppercase letter represent variables while identifiers that begin with a lowercase letter represent constants. The same set of built-in predicates for unifying terms and evaluating arithmetic expressions exist in λProlog. In addition to λProlog's support of predicate variables and λ-terms, the most notable difference between the syntax of the Prolog and λProlog is that λProlog uses a curried notation. Thus, the Prolog program `length/3`:

```
length([],0).
length([H|T],Result) :-
    length(T,X), Result is X + 1.
```

which finds the summation of all the elements in its input list would be written in λProlog's curried form as:

```
length [] 0.
length [H|T] Result :-
    length T X, Result is X + 1.
```

We can rewrite `length/2` as a single clause using disjunction:

```
length List Result :-
        (List = [], Result = 0);
        (List = [H|T], length T X, Result is X + 1).
```

which enables us to write `length/2` in λProlog is as a λ-term:

```
length List Result :-
       (X\Y\(sigma H\(sigma T\(sigma R\(
        (X = [], Y = 0);
        (X = [H|T], length T R, Y is R + 1)
     ))))) List Result.
```

λ-terms are used in λProlog to represent predicate application and anonymous predicates (i.e., predicates that have no name associated with them). Predicate application is denoted in λProlog by juxtaposition. Anonymous predicates are denoted with λ-abstractions which have the form λ`x.`ρ`(x)` in λ-calculus and the form `(X\(`ρ`(X))` in λProlog and represents an anonymous predicate that has a single argument `X` which succeeds if ρ`(X)` succeeds where ρ`(X)` is an arbitrary set of λProlog subgoals. In addition to supporting λ-terms, λProlog also permits existential quantifiers. λProlog uses the keyword `sigma` to represent the existential quantifier ∃ so the λ-term λ`x.`λ`y.`∃`z.(p x y z)` would be coded in λProlog as `(X\Y\(sigma Z\(p X Y Z)))` and represents an anonymous predicate that has two arguments `X` and `Y` which succeeds if `p X Y Z` succeeds for some `Z`.

Another important difference between Prolog and λProlog is that λProlog is a typed language. λProlog has several built-in types, including types for *int*, *bool*, *list*, and *o* (the type of propositions). If τ_1 and τ_2 are types then $(\tau_1 \rightarrow \tau_2)$ is a type corresponding to the set of functions whose domain and range are given by τ_1 and τ_2, respectively. The application of T_1 to T_2 is represented as $(T_1\ T_2)$ and has the type τ_1 if T_1 is a term of type $(\tau_2 \rightarrow \tau_1)$ and T_2 is a term of type τ_2. If X is a variable and T is a term of type τ', then the abstraction $(X : \tau \setminus T)$ is a term of type $\tau \rightarrow \tau'$.

λProlog has a built-in type inference mechanism which gives its programmers the illusion that they are programming in a typeless language. Thus, the type system of λProlog serves as an aid to the programmer rather than an added layer of syntax. Lists and integers are handled the same way in λProlog as they are in Prolog. Unlike Prolog, however, λProlog supports separate types for propositions and booleans. The type *o* captures propositions and has the values `true` and `fail` and operations for conjunction, disjunction, and implication of propositions. The type *bool* captures boolean expressions and has the values `truth` and `false` and operations for conjunction and disjunction of booleans, and relationship comparisons (<, =<, >, >=). Note that because booleans are distinct from propositions, it is necessary to have the λProlog subgoal `truth is X < Y` in place of the Prolog subgoal `X < Y`.

It is possible to define a set of λProlog program schemata which enable the incorporation of conditional recursion into logic programming. Conditional iteration in the form of WHILE loops imposes structure on imperative languages, abstracting the essence of conditional repetition. WHILE loops are basic program schemata which capture commonly occurring imperative programming techniques. Conditional recursion serves the same role for logic programming languages.

3 Logic Program Templates

Computers have been used in various aspects of instruction for several years. The first use of computers in an educational setting was computer-aided instruction (CAI) where instructional materials (e.g., textbooks or workbooks) were simply stored in computer files and students were able to use them in a variety of structured ways. Intelligent tutoring systems (ITS) emerged in the 1970s with the goal of enhancing the instruction available on traditional CAI systems by facilitating instruction that was tailored to its individual students. Not only did this shift in focus enable an enhancement over traditional CAI, it also provided a style of teaching that was not possible in traditional classrooms with one teacher per thirty students. One of the original goals of ITSs was to extend the power and accuracy of the adaptive instruction available in traditional CAI systems by examining more than just the student's answers to the problems she was assigned [35].

By equipping them with a student model, ITSs are able to dynamically adapt their instruction through instructional planning. Instructional planning has the goal of configuring the most efficient sequence of instructional operations to communicate a body of knowledge to the student. There are two major aspects of instructional planning: presentation determination and subject matter selection. Determining how the subject material should be presented relies on accurate modeling of the student's preferences and learning styles. Subject matter selection requires knowledge of the student's abilities (i.e., prerequisite or background knowledge) and her capabilities (i.e., her readiness to learn the new material). This adaptive tutoring is made possible by a precise understanding and modeling of both the student and the domain being taught. Obtaining a precise model of the student, however, is a formidable (if not intractable) task. Self [34] has outlined the shortcomings of current approaches to student modeling and proposed that ITSs take on a more collaborative role.

Even if it was possible to obtain precise student models, ITSs still fall short of their goal of truly adaptive instruction because they plan instruction based on their students' ability to perform with known concepts rather than their students' readiness to learn new concepts. Based on Vygotsky's developmental theory and his concept of the "zone of proximal development" [38], the schema-based instructional technique employed by our Prolog tutor [24] provides an ideal framework for implementing the guided learning environment necessary to measure the student's knowledge zone.

Traditionally, mental development level has been determined by an individual's performance on a test requiring some sort of individual problem solving. Lev Vygotsky, an influential Russian developmental psychologist in the 1930s, was bothered by the use of this form of testing to determine mental development level since it "oriented learning towards yesterday's development, toward developmental stages already completed" [38, p. 89]. As an alternative, he proposed an extension to the traditional testing paradigm which included both independent problem solving and guided (or assisted) problem solving. While independent problem solving provides a good indicator of actual developmental level, he proposed the use of guided problem solving as an indicator of

potential developmental level.

Guided problem solving provided a mechanism for measuring what he labelled the "zone of proximal development" which is "the difference between the actual developmental level as determined by independent problem solving and the level of potential development as determined through problem solving under adult guidance or in collaboration with more capable peers" [38, p. 86]. The "zone of proximal development" provides a means for distinguishing fully developed (or mature) concepts from developing (or immature) concepts. Empirical evidence [7,8] has shown the effectiveness of guided problem solving as an instructional technique.

An extension to Vygotsky's basic framework is the method of collaborative problem solving (or peer collaboration) in which two or more individuals of essentially equal abilities work together to solve problems [14,15,19,32]. It has also been argued that collaborative problem solving promotes mental development by creating an environment which produces critical cognitive conflicts [12]. Piaget [33] claimed that cognitive conflicts arise when a child's beliefs are in contradiction with actual experiences. When a child finds herself in disagreement with a collaborating peer, she is forced to examine her point of view and reassess its validity. Thus, collaborative problem solving provides an effective mechanism for motivating children to reassess the validity of their views which in turn forces them to reformulate their beliefs.

Collaborative problem solving has been incorporated into intelligent tutoring systems for a number of domains, including mathematics [9] and political science [13]. Each of these collaborative learning systems has replaced the expert and student models with an automated collaborating peer which learns along with the human student. These systems avoid the difficult problems associated with expert and student modeling, while providing the student with a learning environment that promotes the acquisition of planning and problem solving skills [3]. Recently, some researchers have suggested that the traditional approach to instruction be replaced by a form of cognitive apprenticeship [11] or anchored instruction [10], arguing for instruction that is situated [6]. The underlying claim of the situated cognition movement is the desire to enculturate the student into the domain of the teacher by involving the student in a series of authentic activities which are designed to incrementally improve the skills of the student.

Large programming projects are seldom completed by a single programmer. Rather they are designed, implemented, and tested by larger teams of programmers working together to achieve the common goal of producing a useable final software product. This collaborative effort enables the creation of sophisticated systems that would otherwise be impossible to produce. Thus, in the case of computer programming, collaborative learning *is* situated learning. For such domains, the creation of a collaborative learning environment exploits the advantages of both of these approaches. In order to facilitate collaborative programming, a programming language must support structured programming constructs. Unfortunately, logic programming languages do not have explicit support of any structured programming constructs. It is possible to incorporate a structured style of programming into logic programming languages by exploiting program schemata and programming techniques.

In order to successfully enculturate the novice into the domain of the expert programmer, it is not possible to simply thrust them into the world of abstract problem solving. While the expert talks about programs at an abstract level, novices necessarily are focussed at the concrete level. In order for the expert (teacher) to situate the learning towards a more abstract level, she must first "compromise" with the novice (student) and talk at very concrete level using carefully chosen examples. By selecting similar problems and using templates to both guide the student and highlight the common programming patterns or techniques, it is possible to facilitate the learning of abstract problem solving skills. The idea is to use *program templates* to help the novice understand concrete programming examples while at the same time promoting the concept of abstract *program schemata*. Thus, we propose using an abstract meta-language as a crutch to a more useful higher-order programming language.

A number of researchers have looked into various approaches to meta-languages which support templates, including our work on basic Prolog schemata [21,24], the work by Brna and his colleagues on Prolog programming techniques [4,5], the work by Barker-Plummer on Prolog clichés [2], Flener's work on logic algorithm schemata [16], and the work by Hamfelt and Nilsson on metalogic programming techniques [28]. An alternative approach to using templates in a meta-language is have a set of general Prolog programs which can be extended by adding arguments and subgoals to produce other programs within the class. This is the approach taken by Kirschenbaum and Sterling with their Prolog skeletons and programming techniques [29]. Although appropriate for the construction of logic programs by experienced programmers, we believe that novices find completing templates rather than extending programs easier to understand.

Like most logic programming languages, lists are a basic type in Prolog and λProlog. Recall the `length/2` program from the previous section:

```
length [] 0.
length [H|T] Result :-
    length T X, Result is X + 1.
```

λProlog program which finds the summation of all the elements of an arbitrary list of integers (e.g., the summation of the list [2,4,1,9,12,3] is 31) would look like:

```
sum [] 0.
sum [H|T] Result :-
    sum T X, Result is X + H.
```

A pattern seems to be arising here. The only difference between `sum/2` and `length/2` is that `sum/2` adds the first element to the sum of the remainder of the list whereas `length/2` merely increments the length of the remainder of the list by one. Indeed, this pattern is quite common among list processing tasks. Consider, for example, the task of finding the product of a list of numbers (e.g., the product of the list [2,4,1,9,12,3] is 2592). Thus, we merely change the `Result is X + H` subgoal to

`Result is X * H` in the body of the clause and we have a `product/2` program:

```
product [] 1.
product [H|T] Result :-
    product T X, Result is X * H.
```

Note, however, that `product/2` has an additional difference. The base case value was changed from 0 to 1 since 1 is the identity for multiplication. These programs are not the only ones that show this common pattern. Other programs which perform quite different tasks on lists are also members of this class of programs. Consider the task of enqueueing an element to the end of a list. λProlog program for this task would look like:

```
enqueue [] E [E].
enqueue [H|T] E Result :-
    enqueue T E X, Result = [H|X].
```

Now consider appending two lists together. A λProlog program for this task would look like:

```
append [] L L.
append [H|T] L Result :-
    append T L X, Result = [H|X].
```

The more common implementation of `append/3` has the `Result = [H|X]` subgoal unfolded into the head of the clause:

```
append [] L L.
append [H|T] L [H|X] :- append T L X.
```

One of the most commonly used program examples for recursive list processing is list reversal. Although there are other ways to write a list reversal program in λProlog, the one produced by most novice programmers is the following naive `reverse/2`:

```
reverse [] [].
reverse [H|T] Result :-
    reverse T X, append X [H] Result.
```

All of these programs are examples of *global list processing* tasks and can be captured by a common logic program schema and realized in a couple of general templates. It is possible to represent a two argument global list processing template by adapting our meta-language [21,24] to λProlog:

```
template2 [] ϑ1.
template2 [H|T] ϑ2 :-
    «θ1 ϑ3 «ϑ4»j,» template2 T ϑ5  «,θ2 ϑ6 «ϑ7»k».
```

and a three argument global list processing template:

```
template3 [] ϑ1 ϑ8.
template3 [H|T] ϑ2 ϑ9 :-
    «θ1 ϑ3 «ϑ4»j,» template3 T ϑ5 ϑ10 «,θ2 ϑ6 «ϑ7»k».
```

One can further instantiate the two argument global list processing template into `tally/2`:

```
tally [] ϑ1.
tally [H|T] X :-
    «θ1 ϑ3 «ϑ4»j,» tally T Y, X is F(H,Y).
```

where `F(H,Y)` represents an arbitrary arithmetic expression. The two argument global list processing schema can also be instantiated into `alter/2`:

```
alter [] ϑ1.
alter [H|T] X :-
    «θ1 ϑ3 «ϑ4»j,» alter T Y, P H Y X.
```

where `P/3` is an arbitrary predicate which has three arguments. It is possible to further instantiate `tally/2`, `alter/2`, and the three argument global list processing template into programs for `length/2`, `sum/2`, `product/2`, `reverse/2`, `insertion-sort/2`, `append/3`, and `enqueue/3`.

The underlying philosophy of our Prolog tutor is to teach programming through program templates and schemata. Rather than teaching recursion as a general technique by presenting the mathematical foundations of recursive function theory whereby the student is given a universal recursive schemata (e.g., [39]) or using an example-based method whereby the student is forced to generalize a general understanding of recursion from specific examples, schema-based instruction teaches recursion through a set of general programming techniques. Thus, the student is relieved of the task of inducing the basic concept of recursion while being provided with a "toolbox" of general programming techniques.

Because of the hierarchical nature of our domain, it is possible to measure a student's capability with respect to her understanding of the schemata (or programming techniques). When a program is assigned, a schema is selected to serve as a template for the student to complete. The template serves as a λProlog microworld. These λProlog microworlds limit the number of ways of implementing a program. If the student is unable to complete the template to produce the desired program (or if she incorrectly completes the template) then hints are given. Hints include explaining the components of the templates (i.e., describing the microworld) and providing the student with partial or complete solutions to the assigned programs (i.e., simplifying the microworld).

The student's performance on the assigned program determines her knowledge zone with respect to the assigned schema. If the student is able to solve the problem (i.e., produce the desired program) without any hints then the schema (i.e., programming

technique) is within her ability. If she can solve the problem after being given some of the above hints then the programming technique is within her capability, but is not a mature concept within her ability yet. Finally, if the student is still unable to solve the problem after she has been given hints then the programming technique is considered outside her "zone of proximal development." Once the knowledge zone has been identified, the tutor can use it to adapt the lesson plan to the student's cognitive potential. The lesson plan can be modified in a number of ways, including the selection of the next problem and template to be assigned. The basic lesson plan is the same for each student as shown in Figure 1. Each of the programs of the lessons have corresponding entries in the schema hierarchy of Figure 2. The following notation has been used:

- The prefix `tr_` has been used to represent the tail recursive version of the program. Note that the absence of the `tr_` prefix implies the more general (declarative or non-accumulator) version of the program.

- The prefix `g_` has been used to represent a guarded version of the schema. For programs, the actual guard (e.g., `odd`) is part of the predicate name. A guarded version of a program is one which contains multiple recursive clauses.

Lesson	Problem	Template	Example
1	tr_sum/2	tr_tally/2	tr_length/2
2	tr_product/2	tr_template/2	-
3	tr_odd_sum/2	gtr_tally/2	tr_count/2
4	tr_even_sum/2	gtr_template/2	-
5	tr_pos_product/2	gtr_template/2	-
6	sum/2	template/2	length/2
7	count/2	g_template/2	tr_count/2
8	append/3	template/3	enqueue/3
9	reverse/2	-	-

Figure 1. Initial Lesson Plan

For example, `tr_odd_sum/2` (which is the tail recursive version of the `sum/2` program using the `odd` guard) has the following normal form implementation:

```
tr_odd_sum L S :- tr_odd_sum L 0 S.
tr_odd_sum [] S S.
tr_odd_sum [H|T] X S :-
    odd H, !, Y is X + H, tr_odd_sum T Y S.
tr_odd_sum [H|T] X S :- tr_odd_sum T X S.
```

Each lesson consists of three components: problem, template, and example. The general form of each lesson is as follows. If a new concept is being introduced then the tutor gives the student a brief overview of the new concept in order to provide an appropriate context. If the lesson includes an example then the example is shown and explained. The problem is then described and the student is asked to complete the template in order to produce a correct program.

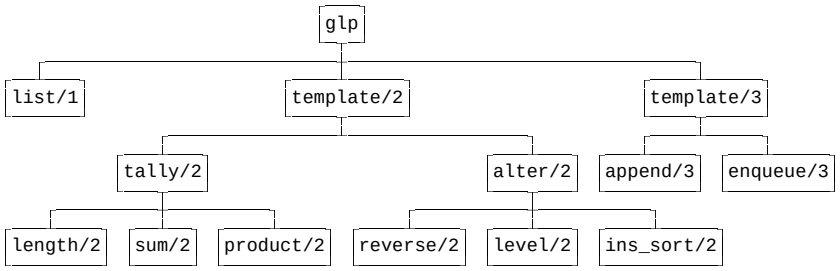

Figure 2. Global List Processing Schema Hierarchy

A tutoring session is initiated by the student entering `tutor`. The following scenario gives the interaction between a student who has mastered tail recursion and is now ready to be introduced to the more general forms of recursion with lesson 6:

Student: `tutor`

Tutor:
```
Welcome to the Schema-Based Prolog Tutor!
What is your name?
```

Student: `Barb`

Tutor:
```
OK, Barb, let's look at more general forms of
recursion today.  We have already looked at tail-
recursive versions of tr_length/3, tr_sum/3, and
tr_product/3.  Now let's look at alternative, more
declarative, approaches to solving these problems.
For example, the tail-recursive program for tr_
length/3 looks like:

    length List Length :-
        tr_length List 0 Length.
```

```
tr_length [] Acc Acc.
tr_length [H|T] N L :-
    M is N + 1, tr_length T M L.
```

and the corresponding declarative version of length/2 looks like:

The "length" of the empty list is 0 (Base Case):

```
    length [] 0.
```

The "length" of a non-empty list is equal to 1 more than the "length" of its tail (Recursive Step):

```
    length [H|T] Length :-
        length T L, Length is L + 1.
```

Your task is to write a Prolog program which takes as input a list and returns the sum of the list of elements using this declarative approach. You've done this before with the tail-recursive tr_sum/3:

```
    sum List Sum :- tr_sum List 0 Sum.
    tr_sum [] Acc Acc.
    tr_sum [H|T] N S :-
        M is N + H, tr_sum T M S.
```

Now try writing the declarative version of sum/2 by completing the following template:

```
    sum [] [    ].

    sum [H|T] Sum :- sum T S, [        ].
```

At this point, the student has two options. She can either attempt to write the program or she can request assistance. If she attempts to write the program and produces a correct program then the tutor assigns her the next problem in the original lesson plan which is `count/2`. Because she was able to solve the problem without any assistance from the tutor, it is assumed that the student has a good grasp of the programming construct being taught (i.e., the construct is well within her "zone of proximal development") so the lesson is tailored to the student by omitting the remedial problems of `product/2` and `square_sum/2` and generalizing the template (i.e., instead of assigning template `g_tally/2` in lesson 7 the tutor assigns the more general template `g_template/2`):

Student:
```
sum [] 0.
sum [H|T] Sum :- sum T S, Sum is S + H.
```

Tutor: Very good, Barb! OK, now let's try a similar problem. Let's assume that our list now contains arbitrary elements (i.e., some of the elements are integers and some of them are non-integers) and we want to "count" the number of integers in the list. You've already given a tail-recursive solution for this problem:

```
count List Count :- tr_count List 0 Count.
tr_count [] Acc Acc.
tr_count [H|T] N C :-
    integer H, !, M is N + 1, tr_count T N C.
tr_count [H|T] N C :- tr_count T N C.
```

Now try writing this program again using the declarative approach you've just learned by completing the following template:

```
count [] [____].

count [H|T] [____] :- [__________],!,

    [__________],count T [____],[__________].

count [H|T] [____] :- [__________],

    count T [____],[__________].
```

If the student is unable to solve the problem then she can request assistance from the tutor by typing `help`. The first level of assistance that the tutor provides is to explain the components of the template. There are two ways to explain the components of the template. The first approach is to simply break the template down into its recursive function components, base case and recursive step, and explain the role of each component:

Student: `help`

Tutor: There are 2 major components to this template:

- base case
- recursive step

For the base case, the basic idea is to stop processing when the list becomes empty and return 0 for the sum.

For the recursive step, the basic idea is to remove the first element from the input list and recursively invoke the main predicate with the tail of the list and then add the value of the head to the sum of the tail of the list.

Which component do you want to attempt first:

1. base case
2. recursive step
3. need more assistance

After explaining each of these components, the tutor gives the student a chance to complete them one at a time or the student can request further assistance. If a student requests further assistance then the tutor provides her with a more detailed template "explanation" by providing her with a more specific template to complete:

Student: `help`

Tutor: `Try writing the program by completing the following template:`

```
sum [] [    ].

sum [H|T] Sum :- sum T S, [        ].
```

By providing a more specific template for the student to complete, the tutor has given the student a partial solution to the problem. This type of "hint" may be sufficient for some students to complete the program. For such students, the problem is within their "zone of proximal development" but is not a mature concept yet. It is now the goal of the tutor to ensure that this concept is more fully developed. This can be done by assigning a remedial problem (e.g., `product/2`) which belongs to the same class of programs as the problem that she just completed. At this point, the tutor is fairly confident that the student could solve the new problem given the more-specific template, but must ensure that the student can solve this problem given the original template:

Student:
```
sum [] 0.
sum [H|T] Sum :- sum T S, Sum is S + H.
```

Tutor: Very good, Barb! OK, now let's try a similar problem. Write a program which takes as input a list and returns the product of the list of elements by completing the following template:

```
product [] [    ].

product [H|T] Prod :- product T P, [        ].
```

Upon successful completion of this remedial lesson, the tutor returns to the 7th lesson of the student's lesson plan and assigns `count/2` with template `g_tally/2`:

Student:
```
product [] 1.
product [H|T] Prod :- product T P, Prod is P * H.
```

Tutor: Very good, Barb! OK, now let's try a similar problem. Let's assume that our list now contains arbitrary elements (i.e., some of the elements are integers and some of them are non-integers) and we want to "count" the number of integers in the list. You've already given a tail-recursive solution for this problem:

```
count List Count :- tr_count List 0 Count.
tr_count [] Acc Acc.
tr_count [H|T] N C :-
    integer H, !, M is N + 1, tr_count T N C.
tr_count [H|T] N C :- tr_count T N C.
```

Now try writing this program again using the declarative approach you've just learned by completing the following template:

```
count [] [    ].

count [H|T] [    ] :- integer H, !,

    count T [    ], [        ].

count [H|T] [    ] :-

    count T [    ], [        ].
```

If the student is still unable to solve the problem after being given a partial solution (i.e., a more-specific template) then the complete solution is given and explained. In this case, the problem is considered outside the student's "zone of proximal development" so the tutor must alter its original instructional goal and focus on a smaller class of problems. Note that in this case, it is not clear whether or not the student can handle this smaller class of problems. This can be tested by assigning a remedial problem (e.g., `product/2` and asking the student to complete the more-specific template. Even if she is able to handle this problem then another remedial problem (e.g., `square_sum/2` which is the same as `sum/2` except the addition operator is replaced with a "new" operator #) should be assigned with the original template to ensure that the original class of problems is within the student's capabilities.

Student: `help`

Tutor: Since you seem to be having so much trouble, let me give you my solution to the problem:

The "sum" of the empty list is 0 (Base Case):

```
sum [] 0.
```

The "sum" of a non-empty list is equal to the "sum" of its tail plus the head of the list (Recursive Step):

```
sum [H|T] Sum :- sum T S, Sum is S + H.
```

OK, Barb, let's continue with another problem. Write a program which takes as input a list and returns the product of the list of elements by completing the following template:

```
product [] [____].
product [H|T] Prod :-
            product T P,Prod is [____].
```

Student:
```
product [] 1.
product [H|T] Prod :- product T P, Prod is P * H.
```

Tutor: Very good, Barb! OK, now let's try a similar problem. Write a program which takes as input a list and returns the "square_sum" (i.e., the sum of the squares of the elements) of the list of elements (where "X is Y#Z" binds X to the expression $Y+Z^2$) by completing the following template:

```
square_sum [] [    ].
square_sum [H|T] SSum :-
    square_sum T S, [          ].
```

If the student is able to complete this second remedial problem given the original template then the tutor returns to the 7th lesson of the student's lesson plan and assigns `count/2` with template `g_tally/2`:

Student:
```
square_sum [] 0.
square_sum [H|T] SSum :- square_sum T S,
    Ssum is S # H.
```

Tutor: Very good, Barb! OK, now let's try a similar problem. Let's assume that our list now contains arbitrary elements (i.e., some of the elements are integers and some of them are non-integers) and we want to "count" the number of integers in the list. You've already given a tail-recursive solution for this problem:

```
count List Count :- tr_count List 0 Count.
tr_count [] Acc Acc.
tr_count [H|T] N C :-
    integer H, !, M is N + 1, tr_count T N C.
tr_count [H|T] N C :- tr_count T N C.
```

Now try writing this program again using the declarative approach you've just learned by completing the following template:

```
count [] [    ].

count [H|T] [    ] :- integer H, !,

                count T [    ], [          ].

count [H|T] [    ] :- count T [    ],

                    [          ].
```

We have presented schema-based instruction as an alternative to the approaches advocated in most introductory and intermediate logic programming texts. By explicitly presenting logic programming templates and presenting programs in the same class as a coherent unit, schema-based instruction stresses the importance of classes of programs and promotes the acquisition of basic logic programming constructs while at the same time providing a bridge to higher-order programming.

4 Logic Program Schemata

In addition to enhancing instruction, the use of logic program templates also promotes a structured approach to logic programming. Structured logic programming is made possible with logic program schemata. High-level programming languages were developed to make programming easier by abstracting out the essence of programming from the physical architecture of the machine on which the programs are executed. The progression of high-level programming languages over time has shown higher levels of abstraction. For example, control structures like WHILE loops were included in imperative programming languages because it was discovered that looping structures were used throughout programs. A similar abstraction is possible for recursive programs in the form of higher-order program schemata. We have identified several logic program schemata that serve as prototype logic programs for list processing [26] and a corresponding set of schema-based logic program transformations that enable a programmer to transform a given prototype logic program schema into the desired logic program [27].

Each of the logic program schemata has two arguments, an input list and a result. Although many logic programs can be used with various modes, we assume a given mode for each of our logic programs. In addition to recursive list processing schemata, it is also possible to define a set of recursive numeric programs which also have two arguments. One of the largest classes of list processing programs is the class of global list processing programs which includes all those list processing programs that process all elements of the input list (i.e., the entire input list is reduced). Global list processing programs are captured by the `reduceList/2` schema:

```
reduceList [] Result :-
    Base Result.
reduceList [H|T] Result :-
    reduceList T R, Constructor H R Result.
```

Global integer processing programs are captured by the `reduceNumber/2` schema:

```
reduceNumber 1 Result :-
    Base Result.
reduceNumber N Result :-
    M is N - 1, reduceNumber M R,
    Constructor N R Result.
```

These logic program schemata can be converted to higher-order logic programs by adding the predicate variables to the set of arguments. For example, the following higher-order `reduceList/4` program can be produced from the `reduceList/2` schema:

```
reduceList [] Result Constructor Base :-
    Base Result.
reduceList [A|B] Result Constructor Base :-
    reduceList B C Constructor Base,
    Constructor A C Result.
```

which can be written using a single disjunctive clause:

```
reduceList List Result Constructor Base :-
    (List = [], Base Result);
    (List = [A|B],
     reduceList B C Constructor Base,
     Constructor A C Result).
```

Likewise, it is possible to produce a higher-order global integer processing program `reduceNumber/4` by adding the predicate arguments to the `reduceNumber/2` schema:

```
reduceNumber 1 Result Constructor Base :-
    Base Result.
reduceNumber N Result Constructor Base:-
    M is N - 1, reduceNumber M R Constructor Base,
    Constructor N R Result.
```

which can be written using a single disjunctive clause:

```
reduceNumber N Result Constructor Base :-
    (N = 1, Base Result);
    (M is N - 1, reduceNumber M R Constructor Base,
     Constructor N R Result).
```

The `reduceList/2` and `reduceNumber/2` schemata can be generalized to include all singly-recursive reduction programs by incorporating the termination (or stopping) condition with the base case value computation and permitting an arbitrary destructor:

```
reduce List Result :-
    Base List Result.
reduce List Result :-
    Destructor List H T, reduce T R,
    Constructor H R Result.
```

which corresponds to the higher-order `reduce/5` program:

```
reduce List Result Destructor Constructor Base:-
    Base List Result.
reduce List Result Destructor Constructor Base :-
    Destructor List H T,
    reduce T R Destructor Constructor Base,
    Constructor H R Result.
```

which can be written using a single disjunctive clause:

```
reduce List Result Destructor Constructor Base:-
    (Base List Result);
    (Destructor List H T,
     reduce T R Destructor Constructor Base,
     Constructor H R Result).
```

Some explanation of the `reduce/2` schema (and corresponding higher-order `reduce/5` program) is in order. It contains two arguments and has three predicate variables. The first argument is the primary input and the second argument is the primary output. The primary input and output can be either simple or structured terms, but they are both first-order terms. The three predicate variables represent arbitrary λProlog predicates. The predicate variable `Destructor` defines the process for destructing the input. The predicate variable `Constructor` defines the process for constructing the output. The other predicate variable, `Base`, is used to define the terminating condition, defining both the process to identify the terminating condition and the process which defines how to construct the output for the terminating condition. An example should help clarify `reduce/2`.

Consider the `factorial/2` program. For an arbitrary query `factorial A B`, the primary input is `A` and primary output is `B`. The destructor predicate decrements the input by one. This process can be defined with the anonymous predicate `(X\Y\Z\( Z is Y - 1, Y = X))`. The constructor predicate for `factorial/2` multiplies the current input by the factorial of one less than the current input and can be defined with the anonymous predicate `(X\Y\Z\(Z is X * Y))`. As can be seen in the base case clause of the definition of `factorial/2`, the terminating condition occurs whenever the input becomes one and the terminating output value should be 1. This process can be defined with the anonymous predicate `(X\Y\(X = 1, Y = 1))`. Combining all this together, one can produce a program for `factorial/2` by instantiating the predicate variables in `reduce/2`:

```
factorial N Result :-
    (X\Y\(X = 1, Y = 1)) N Result.
factorial N Result :-
    (X\Y\Z\(Z is X - 1, Y = X)) N C M,
    factorial M R,
    (X\Y\Z\(Z is X * Y)) C R Result.
```

Furthermore, since `factorial/2` is a global integer processing program, it is also possible to produce a program for it by instantiating the predicate variables in `reduceNumber/2`:

```
factorial 1 Result :-
    (X\(X = 1)) Result.
factorial N Result :-
    M is N - 1, factorial M R,
    (X\Y\Z\(Z is X * Y)) N R Result.
```

Now consider `sum/2` again. For an arbitrary query `sum A B`, the primary input is `A` and primary output is `B`. The destructor predicate decomposes the input into the head element and the tail of the list. This process can be defined with the anonymous predicate `(X\Y\Z\(X = [Y|Z]))`. The constructor predicate for `sum/2` computes the summation by adding the current element to the sum of the rest of the list and can be defined with the anonymous predicate `(X\Y\Z\(Z is X + Y))`. As can be seen in the base case clause of the definition of `sum/2`, the terminating condition occurs whenever the input list becomes empty and the terminating output value should be `0`. This process can be defined with the anonymous predicate `(X\Y\(X = [], Y = 0))`. A program for `sum/2` can be produced by instantiating the predicate variables in `reduce/2`:

```
sum List Result :-
    (X\Y\(X = [], Y = 0)) List Result.
sum List Result :-
    (X\Y\Z\(X = [Y|Z])) List H T, sum T R,
    (X\Y\Z\(Z is X + Y)) H R Result.
```

Furthermore, since `sum/2` is a global list processing program, it is also possible to produce a program for it by instantiating the predicate variables in `reduceList/2`:

```
sum [] Result :-
    (X\(X = 0)) Result.
sum [H|T] Result :-
    sum T R, (X\Y\Z\(Z is X + Y)) H R Result.
```

It is also possible to write each of the programs that have been presented so far as one line programs by simply invoking the `reduceList/4` program. For example, `sum/2` can be written more concisely as follows:

```
sum A B :- reduceList A B
    (X\Y\Z\(Z is X + Y))
    (X\(X = 0)).
```

One can write `append/3` more concisely as follows:

```
append A B C :- reduceList A C
    (X\Y\Z\(Z = [X|Y]))
    (X\(X = B)).
```

The following λProlog implementation of `reverse/2` using only `reduceList/4` can be written:

```
reverse A B :- reduceList A B
    (F\G\H\(reduceList G H (X\Y\Z\(Z = [X|Y])) [F])
    (F\(F = [])).
```

This implementation of `reverse/2` shows the notion of *nested recursion*, which is synonymous to nested WHILE loops in imperative languages. An alternative implementation of `reverse/2` takes advantage of a pre-existing procedure definition for `append/3` and invokes it directly rather than redefining it:

```
reverse A B :- reduceList A B
    (X\Y\Z\(append Y [X] Z))
    (X\(X = [])).
```

Consider `append/3` again. For an arbitrary query `append A B C`, the primary input is `A` and primary output is `C`. The destructor predicate decomposes the input into the head element and the tail of the list. This process can be defined with the anonymous predicate `(X\Y\Z\(X = [Y|Z]))`. Likewise, the constructor predicate for `append/3` composes a new list and can be defined with the anonymous predicate `(X\Y\Z\(Z = [X|Y]))`. The terminating condition occurs whenever the input list becomes empty and the terminating output value should be assigned to `B`. This can be defined with the anonymous predicate `(X\Y\(X = [], Y = B))`. Combining all this together produces the following definition for `append/3`:

```
append A B C :- reduce A C
    (X\Y\Z\(X = [Y|Z]))
    (X\Y\Z\(Z = [X|Y]))
    (X\Y\(X = [], Y = B)).
```

In a similar fashion, `reverse/2` (which reverses the elements of a list) can be defined using `reduce/5`:

```
reverse A B :- reduce A B
    (X\Y\Z\(X = [Y|Z]))
    (X\Y\Z\(append Y [X] Z))
    (X\Y\(X = [], Y = [])).
```

The higher-order `reduceList/4` program (or any of the other programs presented so far) can also be defined using `reduce/5`:

```
reduceList A B C D :- reduce A B
    (X\Y\Z\(X = [Y|Z]))
    C
    (X\Y\(X = [], D Y)).
```

All of the programs that we have seen so far fall in the class of *global list processing* programs since they process the entire input list. Some programs only process part of the input list. For example, `insert/3` takes a sorted list and an element and inserts the element in its correct position in the list, stopping whenever it finds an element in the input list that is larger than the element being inserted or the input list becomes empty. A λProlog program for `insert/3` can be written using `reduce/5`:

```
insert A B C :- reduce A C
    (X\Y\Z\(X = [Y|Z]))
    (X\Y\Z\(Z = [X|Y]))
    (X\Y\(sigma V\(sigma W\(
      ((X = []);
      (X = [V|W], truth is B<V)), Y = [B|X]
    )))).
```

The main difference between `insert/3` and all of the previous programs is the terminating condition. There are actually two terminating conditions. The correct position for insertion of a given element into a sorted list is either immediately in front of the first element in the list which is larger than the given element or at the end of the list if every element in the list is smaller than or equal to the given element. The use of `sigma` identifies the existence of variables which satisfy the terminating condition. Specifically, the λ-term `(X\Y\(sigma V\(sigma W\(((X = []); (X = [V|W], truth is B < V)), Y = [B|X]))))` represents an anonymous two-argument predicate which succeeds if either its first argument is an empty list (i.e., all of the elements in the original list are smaller than or equal to the given element) or if there exist variables `V` and `W` such that the predicate's first argument is unifiable with `[V|W]` where `V` is larger than the given element (i.e., `V` is the smallest element in the original list which is larger than the given element).

Other classic *partial list processing* programs include `member/2` and `position/3`. Each of these predicates can be written using `reduce/5`. The predicate `member/2` is a predicate that produces no outputs, it merely succeeds if the given element is a member of the input list or fails if the given element is not a member of the input list. Thus, the λProlog implementation of `member/2` has a dummy variable in place of the output argument and has the subgoal `true` in place of the recursive and base case constructors in its invocation of `reduce/5`:

```
member A B :- reduce A Dummy
    (X\Y\Z\(X = [Y|Z]))
    (X\Y\Z\(true))
    (X\Y\(sigma W\(X = [B|W]))).
```

The predicate `position/3` takes an input list and an element and returns the position of the element in the input list:

```
position A B P :- reduce A P
    (X\Y\Z\(X = [Y|Z]))
    (X\Y\Z\(Z is Y+1))
    (X\Y\(sigma W\(X = [B|W], Y = 1))).
```

It is also possible to capture programs that construct more than one output or have more than one input list that is being decomposed. Consider the task of partitioning an input list into two output lists which contain all the elements that are less than or equal to a given partitioning element and one which contains all elements that are strictly greater than the given partitioning element. A λProlog implementation of `partition/4` can be written using `reduce/5` by combining the two output lists into a single list which is manipulated appropriately by the constructor predicates:

```
partition A B C D :- reduce A [C,D]
    (X\Y\Z\(X = [Y|Z]))
    (X\Y\Z\(sigma V\(sigma W\(
      (false is X>B, Y = [V,W], Z = [[X|V],W]);
      (truth is X>B, Y = [V,W], Z = [V,[X|W]])
    ))))
    (X\Y\(X = [], Y = [[B],[]])).
```

Notice the use of disjunction in the constructor predicate to permit putting the current element on the appropriate output list. The subgoal `Y = [V,W]` breaks the output into its two output lists `V` and `W`. If the current element is less than or equal to the partitioning element then the current element is added to the first output list with the subgoal `Z = [[X|V],W].` Otherwise, it is added to the second output list with the subgoal `Z = [V,[X|W]]`.

Now consider the task of merging two sorted lists into a single sorted list. This task requires decomposing two input lists. A λProlog implementation of `merge/3` can be written using `reduce/5` by combining the two input lists into a single input list which is manipulated appropriately by the destructor predicate:

```
merge A B C :- reduce [A,B] C
    (X\Y\Z\(sigma U\(sigma V\(sigma W\(
   (X = [[Y|U],[V|W]], truth is Y<V, Z = [U,[V|W]]);
   (X = [[V|W],[Y|U]], false is Y>V, Z = [[V|W],U])
    )))))
    (X\Y\Z\(Z = [X|Y]))
    (X\Y\((X = [[],Y]); (X = [Y,[]]))).
```

Disjunction is used in the destructor predicate in the definition of `merge/3` to enable removing the first element from only one of the input lists. The element is removed from whichever input list has the smallest element. If the smallest element is

contained in the first input list then it is "identified" with the subgoals `X = [[Y|U], [V|W]], truth is Y<V` and "removed" with the subgoals `X = [U,[V|W]], false is Y>V`. Likewise, if the smallest element is contained in the second input list (or if the first element in both input lists is identical) then it is "identified" with the subgoal `X = [[V|W],[Y|U]]` and "removed" from the second input list with the subgoal `X = [[V|W],U]`.

An alternative to developing recursive programs using higher-order programs is to instantiate prototype logic program schemata and extend them. It is important to note that the `reduceList/2` schema is very robust, capturing a large class of programs which also includes `reverse/2`, `insertion_sort/2`, `product/2`, `prefix/2`, and many others. However, although the logic program schemata given so far capture a large group of logic programs, there is still a large group of logic programs that they are unable to capture. We can extend `reduceList/2` to capture other programs like `append/3` and `count/3`. Programming techniques constitute components of programs and program schemata which enable the creation of specialized program schemata from more generalized program schemata. There are two major types of programming techniques: control flow techniques and context techniques. Control flow techniques are applied to program schemata as a way of defining the basic recursive control flow of the program schema. The two most common recursive control flow techniques are *single_recursion* and *double_recursion* which enable the creation of singly recursive and doubly recursive program schemata, respectively. Context techniques are applied to program schemata as a way of defining a context for arguments. The three most common context techniques are *same*, *decompose*, and *compose* which define an argument to be the same across recursive calls, decrease in size across recursive calls, and increase in size across recursive calls, respectively. The techniques *list subgoal* and *list head* are special forms of *decompose* and *compose* which apply to list arguments. It is possible to apply programming techniques to the following general λProlog program schema:

```
schema A1 ... An :- Goals.
```

to produce the `reduce/2` program schema or any of the programs given in this paper. One begins with a 2-argument version of this schema and apply the *single_recursion* technique. The *single_recursion* technique is used to split the body of a clause into a disjunction of two sets of subgoals. One of the subgoals contains base case clauses, while the other set of subgoals contains a recursive call to the predicate. Applying this technique to the general λProlog program schema produces:

```
schema A1 A2 :-
    BaseCase;
    (Before, schema B1 B2, After).
```

Now one can instantiate the `BaseCase` variable in the base portion and the `Before` and `After` variables in the recursive portion of this clause to the subgoals

`(Base A A2)`, `(Destructor A1 A B1)`, and `(Constructor A B2 A2)`, respectively. The resultant clause is identical (modulo variable renaming) to `reduce/2`. It is also possible to produce the global list processing `reduceList/2` schema by beginning with a 2-argument λProlog schema. Applying the *single_recursion* technique produces:

```
schema A1 A2 :-
    BaseCase;
    (Before, schema B1 B2, After).
```

Applying the *list head* technique to the first argument produces:

```
schema A1 A2 :-
    BaseCase;
    (Before, A1 = [A|B1], schema B1 B2, After).
```

Now one can remove `Before` (or instantiate it to `true`) and instantiate `BaseCase` and `After` to `(A1 = [], Base A2)` and `(Constructor A B2 A2)`, respectively. The resultant clause is identical (modulo variable renaming) to `reduceList/2` which captures all global list processing programs (e.g., `sum/2` and `reverse/2`). It is also possible to produce the `reduceNumber/2` schema from a 2-argument λProlog schema. An important thing to note about our approach to generating λProlog program schemata is that there is no need for a meta-language to represent the programming techniques. Furthermore, all intermediate programs are valid (albeit not very useful) λProlog programs.

One of the major differences between logic programs is the number of arguments. In addition to instantiating predicate variables in logic program schemata to produce logic programs, it is also possible to extend program schemata to include additional arguments. There are two types of argument extension that can be applied to the `reduceList/2` schema, corresponding to adding arguments to each of the predicate variables in `reduceList/2`. One can extend the `reduceList/2` schema to have an additional argument on the `Base` predicate:

```
reduceList [] Result ArgBase :-
    Base Result ArgBase.
reduceList [H|T] Result ArgBase :-
    reduceList T R ArgBase, Constructor H R Result.
```

An example of this type of extension is the creation of `append/3` from `prefix/2`:

```
prefix [] L.
prefix [H|T] [H|L] :- prefix T L.
```

The predicate `prefix/2` succeeds if its primary list is a prefix of its other list. The `prefix/2` predicate can be extended by the adding a new argument which

represents the second list (i.e., the list that is to be appended to the primary list). If one makes the new `Base` predicate unify its arguments then `append/3` can be created:

```
append [] L NewArg :- (X\Y\(X = Y)) L NewArg.
append [H|T] [H|L] NewArg :- append T L NewArg.
```

The other argument extension that can be applied to the `reduceList/2` schema is to add an argument to the `Constructor` predicate:

```
reduceList [] Result ArgCons :-
    Base Result.
reduceList [H|T] Result ArgCons :-
    reduceList T R ArgCons,
    Constructor H R Result ArgCons.
```

An example of this type of extension is the creation of `count/3` from `length/2`:

```
length [] 0.
length [H|T] L :- length T X, L is X + 1.
```

If the new `Constructor` predicate increments the count only when the head of the list is unifiable with the newly added argument then `count/3` can be created:

```
count [] 0 Element.
count [H|T] L Element :-
    count T X Element,
    ((H = Element, L is X + 1); L = X).
```

In addition to these semantics-altering extensions, there are two types of semantics-preserving extensions that can be applied to logic programs and schemata to produce equivalent logic programs and schemata: application of programming techniques and combination (or merging) and connection of logic programs and schemata. The first type of semantics-preserving extension is the application of programming techniques to logic program schemata. Programming techniques have been studied fairly extensively and a number of commonly occurring programming practices have been identified. One popular programming technique is the introduction of an accumulator, enabling the composition of the output from the right rather than from the left. Given that a program unifies with the `reduce/2` schema, it can be transformed into the more efficient accumlator implementation by instantiating the following `reduceAcc/2` schema with the same `Base` and `Constructor` predicates:

```
reduceAcc Input Result :-
    Base Dummy Acc, reduceAcc2 Input Result Acc.
reduceAcc2 Input Result Result :-
    Base Input Dummy.
reduceAcc2 Input Result Acc :-
    Destructor Input H T, Constructor Acc H A,
    reduceAcc2 T Result A.
```

assuming `Constructor` is associative. As an example, consider `sum/2` again. The more efficient (tail recursive) accumulator implementation of `sum/2` is produced by instantiating this `reduceAcc/2` schema:

```
sum List Result :-
    (X\(X = 0)) Acc, sum2 List Result Acc.
sum2 [] Result Result.
sum2 [H|T] Result Acc :-
    (X\Y\Z\(Z is X + Y)) Acc H A, sum2 T Result A.
```

Learning this general accumulator schema enables the student programmer to mentally transform straightforward non-accumulator programs into more efficient ones. The other type of semantics-preserving logic program extension which student programmers can use to improve the efficiency of their programs is the combination (or merging) of logic program schemata. The idea is to merge two logic program schemata whenever they have a common argument. Probably the most obvious logic program schemata is to combine logic program schemata which have a common primary input. The `reduceListList/3` combines two `reduceList/2` schemata that have a common primary input:

```
reduceListList [] Result1 Result2 :-
    Base1 Result1, Base2 Result2.
reduceListList [H|T] Result1 Result2 :-
    reduceListList T R1 R2,
    Constructor1 H R1 Result1,
    Constructor2 H R2 Result2.
```

An example of the use of `reduceListList/2` is the creation of a singly-recursive implementation of the `average/2` predicate:

```
average List Average :-
    length List Length, sum List Sum,
    Average is Sum / Length.

length [] 0.
length [H|T] L :- length T X, L is X + 1.

sum [] 0.
sum [H|T] S :- sum T X, S is X + H.
```

which calculates the average value for the elements of a list. Using the `reduceList-List/3` schema, it is possible to transform the `average/2` predicate into the following more efficient implementation:

```
average List Average :-
    average2 List Length Sum, Average is Sum / Length.
average2 [] 0 0.
average2 [H|T] Length Sum :-
    average2 T L S, Length is L + 1, Sum is S + H.
```

Because the `reduceListList/3` schema was created by combining two global list processing schemata which share a common primary input and have distinct outputs, the same process can be used to combine two accumulated implementations of global list processing schemata (or even one of each). In addition to combining logic program schemata, it is also possible to connect two logic program schemata that share a common primary input and the result of one schema is an additional input to the other schema. For example, consider finding the middle element in an arbitrary list:

```
middle List Middle :-
    length List Length, Half is (Length + 1) div 2,
    position List Half Middle.

length  [] 0.
length [H|T] L :- length T X, L is X + 1.

position 1 [X|T] X.
position N [H|T] X :- M is N - 1, position M T X.
```

In order to capture `position/3`, the student must be introduced to another logic program schemata. The `reduceLN/3` schema simultaneously reduces a list and a number. In addition to including `position/3`, the `reduceLN/3` schema also captures programs like `take/3` and `drop/3` which keep or remove the first n elements, respectively. The `reduceLN/3` schema looks like:

```
reduceLN [] N Result.
reduceLN L N Result :-
    L = [H|T], M is N - 1, reduceLN T M R,
    ((N = 1, Base L Result); Constructor H R Result).
```

An equivalent schema which reduces its number up from 1 to the maximum rather than from the maximum down to 1 looks like:

```
reduceLN L N Result :-
    reduceLN2 L 1 N Result.
reduceLN2 [] N Max Result.
reduceLN2 L N Max Result :-
    L = [H|T], M is N + 1, reduceLN2 T M Max R,
    ((N = Max, Base L Result); Constructor H R Result).
```

If one instantiates `Base` to `(X\Y\(sigma Z\(X = [Y|Z])))` and instantiate `Constructor` to `(X\Y\Z\(true))` then `position/3` can be produced from `reduceLN/3` assuming that the case of requesting the n^{th} element from a list of less than n elements is a ill-posed query:

```
position [] N Result.
position L N Result :-
    L = [H|T], M is N - 1, position T M R,
    ((N = 1, (X\Y\(sigma Z\(X = [Y|Z]))) L Result);
     (X\Y\Z\(Z = Y)) H R Result).
```

It is possible to connect the accumulated implementation of `reduceList/2` with the upward reduction implementation of `reduceLN/3` to produce the following connection schema:

```
reduceConnect List Result :-
    Base1 Acc, reduceC2 List Acc Length 1 Half Result.
reduceC2 [] Acc Acc N Max Result :- Connect Acc Max.
reduceC2 List Acc RL N Max Result :-
    List = [H|T], M is N + 1,
    Constructor1 Acc H A, reduceC2 T A RL M Max R,
    ((N = Max, Base2 List Result);
     Constructor2 H R Result).
```

where `Base1` and `Constructor1` are predicates from the accumulated implementation of `reduceList/2` and `Base2` and `Constructor2` are predicates from the upward reduction implementation of `reduceLN/3`. Applying this connection schema to the `middle/2` program produces the following more efficient implementation:

```
middle List Middle :-
    middle2 List 0 Length 1 Half Middle.
middle2 [] Length Length AH Half Middle :-
    Half is (Length + 1) div 2.
middle2 [H|T] AL Length AH Half Middle :-
    NL is AL + 1, NH is AH + 1,
    middle2 T NL Length NH Half Mid,
    ((AH = Half, Middle = H); Middle = Mid).
```

The class of reduction programs share a common destructor. As such, we can refer to the transformations defined so far as *destructor-specific* transformations. It is also possible to have *constructor-specific* transformations. Two of the most popular higher-order programming programs are `map/3` and `filter/3`. Mapping and filtering programs are a subclass of reduction programs that also share a common constructor. Rather than reducing a list by combining each element with the result of reducing the remainder of the list, sometimes it is desirable to map a function predicate across all the elements of a list. For example, we may want to double all of the elements in a list. In order to double all of the elements of a list, we must first apply a function predicate that

doubles each element and then put the doubled element in the front of the list produced by doubling all the elements in the remainder of the list. In general, the predicate `map/3` can be used to apply an arbitrary binary function predicate to each element of a list:

```
map [] Result P :-
    (X\(X = [])) Result.
map [H|T] Result P :-
    map T R P,
    (X\Y\Z\(sigma W\(P X W, Z = [W|Y]))) H R Result.
```

We can write `doubleAll/2` using this `map/3` predicate:

```
doubleAll List Result :
     map List Result (X\Y\(Y is 2 * X)).
```

The predicate `filter/3` takes a unary predicate and a list and filters out all elements from the list that do not satisfy the predicate. For example, we may want to filter out all non-positive numbers from a list of numbers. We can write `filter/3`:

```
filter [] Result P :-
    (X\(X = [])) Result.
filter [H|T] Result P :-
    filter T R P,
    (X\Y\Z\((P X, Z = [X|Y]); Z = Y)) H R Result.
```

We can write `positivesOnly/2` using this `filter/3` predicate:

```
positivesOnly List Result :-
    filter List Result (X\(truth is X > 0)).
```

It is possible to consider the mapping constructor and the filtering constructor as special cases of the following constructor:

```
(P X XX, Z = [XX|Y]); Z = Y
```

Notice that this constructor has the additional disjunctive subgoal `(Z = Y)` which is never invoked for mapping programs and it captures filtering constructors if we rewrite the filtering constructor to add an additional argument to its filtering predicate:

```
(A\B\(P A, A = B))
```

which represents the mapped element. Now we can define the following special case of `reduceList/2` for mapping/filtering programs:

```
mapList [] Result :-
    Base Result.
mapList [H|T] Result :-
    mapList T R,
    ((P H XX, Result = [XX|R]); Result = R).
```

We can connect `mapList/2` and `reduceList/2` where the mapping/filtering program maps a predicate across the elements of the input list and this mapped list is then reduced. For example, we can count the number of positive elements in a given list by filtering out the non-positive elements (using `positivesOnly/2`) and counting the number of elements in the filtered list (using `length/2`):

```
positiveCount List Result :-
    positivesOnly List X, length X Result.

positivesOnly [] [].
positivesOnly [H|T] Result :-
    positivesOnly T R,
    ((truth is H > 0, XX = H, Result = [XX|R]);
     Result = R).

length [] 0.
length [H|T] L :- length T X, L is X + 1.
```

Given `Base` and `Constructor` from the reduction program and `P` from the mapping/filtering program, the mapping/filtering connection to reduction transformation would look like:

```
mapReduceList [] Result :- Base Result.
mapReduceList [H|T] Result :-
    mapReduceList T R,
    ((P H XX, Constructor XX R Result); Result = R).
```

Applying this transformation to the original `positiveCount/2` program produces the following more efficient implementation:

```
positiveCount [] 0.
positiveCount [H|T] Result :-
    positiveCount T R,
    (((truth is H > 0, XX = H), Result is R + 1);
     Result = R).
```

Logic program schemata can be used to help in program development by enabling the programmer to produce a simple straightforward solution to the problem and then transform that solution into an efficient one by applying a set of program transformations. Other examples of schema combination transformations can be found in [27].

5 Conclusion

Imperative programming languages provide their programmers with a set of structured programming constructs. There are currently no structured constructs, however, in logic programming languages. Conditional iteration in the form of WHILE loops imposes structure on imperative languages, abstracting the essence of conditional repetition. WHILE loops are basic program schemata which capture commonly occurring imperative programming techniques. Conditional recursion serves the same role for logic programming languages.

In this paper, we have argued that it is possible to incorporate a structured style of programming into logic programming languages by exploiting program templates and schemata. Program schemata capture the notion of conditional recursion which serves the same role for logic programming languages that conditional iteration does for imperative languages. Representing program schemata requires a higher-order representation language. Previous approaches to representing program templates and schemata have relied on the introduction of an abstract meta-language. Higher-order logic programming languages like λProlog provide an alternative to the meta-language approach which can be introduced after one has mastered the meta-language. In addition to providing an alternative to an abstract meta-language, λProlog's ability to represent λProlog program schemata as λProlog programs enables λProlog to support conditional recursion naturally which promotes a more structured style of logic programming. We have successfully employed this schema-based approach to teaching recursion in several declarative programming languages (including Prolog, λProlog, Logo, Lisp, and Miranda) with much success over the past four years.

Schemata serve a fundamental role in most human cognitive processes. It has been shown that schemata enable the organization of meaningful information for complex domains like computer programming. In addition to being essential to expert programmers, program schemata have also been shown to be useful in teaching recursive Prolog programming to novices. Thus, the incorporation of structured constructs into logic programming made possible with logic program schemata enhances the logic programming paradigm for programmers of all levels.

References

[1] B. Adelson. Problem Solving and the Development of Abstract Categories in Programming Languages. *Memory & Cognition*, 9: 422-433, 1981.

[2] D. Barker-Plummer. Cliché Programming in Prolog. In M. Bruynooghe, editor, *Proceedings of the 2nd Workshop on Meta-Programming in Logic*, Leuven, Belgium, pages 247-256, 1990.

[3] A. Blaye, P. Light, R. Joiner, and S. Sheldon. Collaboration as a Facilitator of Planning and Problem Solving. *British Journal of Developmental Psychology*, 9: 471-493, 1991.

[4] A. Bowles and P. Brna. Programming Plans and Programming Techniques. In P. Brna, S. Ohlsson, and H. Pain, editors, *Proceedings of the 6th World Conference on Artificial Intelligence in Education*, Edinburgh, Scotland, pages 378-385, AACE Press, 1993 (Reprinted in this volume).

[5] P. Brna, A. Bundy, A. Dodd, M. Eisenstadt, C. Looi, H. Pain, D. Robertson, B. Smith, and M. van Someren. Prolog Programming Techniques. *Instructional Science*, 20: 111-133, 1991 (Reprinted in this volume).

[6] J.S. Brown, A. Collins, and P. Duguid. Situated Cognition and the Culture of Learning. *Educational Researcher*, 18(1): 32-42, 1989.

[7] A.L. Brown and R.A. Ferrara. Diagnosing Zones of Proximal Development. In J.V. Wertsch, editor, *Culture, Communication, and Cognition: Vygotskian Perspectives*, pages 139-150, Cambridge University Press, 1985.

[8] J.C. Campione, A.L. Brown, R.A. Ferrara, and N.R. Bryant. The Zone of Proximal Development: Implications for Individual Differences and Learning. In B. Rogoff and J.V. Wertsch, editors, *Children's Learning in the "Zone of Proximal Development"*, pages 77-91, Jossey-Bass, 1984.

[9] T. Chan and A.B. Baskin. Learning Companion Systems. In C. Frasson and G. Gauthier, editors, *Intelligent Tutoring Systems: At the Crossroads of Artificial Intelligence and Education*, pages 6-33, Ablex, 1990.

[10] Cognition and Technology Group at Vanderbilt. Anchored Instruction and its Relationship to Situated Cognition. *Educational Researcher*, 19(6): 2-10, 1990.

[11] A. Collins, J.S. Brown, and S.E. Newman. Cognitive Apprenticeship: Teaching the Craft of Reading, Writing, and Mathematics. In L.B. Resnick, editor, *Knowing, Learning, and Instruction: Essays in Honor of Robert Glaser*, pages 453-494, Lawrence Erlbaum, 1989.

[12] W. Damon. Peer Education: The Untapped Potential. *Journal of Applied Developmental Psychology*, 5: 331-343, 1984.

[13] P. Dillenbourg and J. Self. A Computational Approach to Socially Distributed Cognition. *European Journal of Psychology of Education*, 7: 353-372, 1992.

[14] W. Doise, G. Mugny, and A. Perret-Clermont. Social Interaction and the Development of Cognitive Operations. *European Journal of Social Psychology*, 5: 367-383, 1975.

[15] W. Doise, G. Mugny, and A. Perret-Clermont. Social Interaction and Cognitive Development: Further Evidence. *European Journal of Social Psychology*, 6: 245-247, 1976.

[16] P. Flener. *Logic Program Synthesis from Incomplete Information*. Kluwer Academic Publishers, 1995.

[17] P. Flener and Y. Deville. Logic Program Synthesis from Incomplete Specifications. *Journal of Symbolic Computation*, 15: 775-805, 1993.

[18] P. Flener and Y. Deville. Logic Program Transformation Through Generalization Schemata. In M. Proietti, editor, *Proceedings of the 5th International Workshop on Logic Program Synthesis and Transformation*, Utrecht, The Netherlands, pages 171-173, Springer-Verlag, 1995.

[19] E.A. Forman and C.B. Cazden. Exploring Vygotskian Perspectives in Education: The Cognitive Value of Peer Interaction. In J.V. Wertsch, editor, *Culture, Communication, and Cognition: Vygotskian Perspectives*, pages 323-347, Cambridge University Press, 1985.

[20] N.E. Fuchs and M.P.J. Fromherz. Schema-Based Transformations of Logic Programs. In T.P. Clement and K. Lau, editors, *Proceedings of the 1st International Workshop on Logic Program Synthesis and Transformation*, Manchester, England, pages 111-125, Springer-Verlag, 1991.

[21] T.S. Gegg-Harrison. *Basic Prolog Schemata*. Technical Report CS-1989-20, Department of Computer Science, Duke University, Durham, North Carolina, 1989.

[22] T.S. Gegg-Harrison. Learning Prolog in a Schema-Based Environment. *Instructional Science*, 20: 173-192, 1991.

[23] T.S. Gegg-Harrison. Adapting Instruction to the Student's Capabilities. *Journal of Artificial Intelligence in Education*, 3: 169-181, 1992.

[24] T.S. Gegg-Harrison. *Exploiting Program Schemata in a Prolog Tutoring System*. Ph.D. Dissertation, Department of Computer Science, Duke University, Durham, North Carolina, 1993.

[25] T.S. Gegg-Harrison. Exploiting Program Schemata in an Automated Program Debugger. *Journal of Artificial Intelligence in Education*, 5: 255-278, 1994.

[26] T.S. Gegg-Harrison. Representing Logic Program Schemata in λProlog. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, Kanagawa, Japan, pages 467-481, MIT Press, 1995.

[27] T.S. Gegg-Harrison. Extensible Logic Program Schemata. In J. Gallagher, editor, *Proceedings of the 6th International Workshop on Logic Program Synthesis and Transformation*, Stockholm, Sweden, Springer-Verlag, 1996.

[28] A. Hamfelt and J.F. Nilsson. Declarative Logic Programming with Primitive Recursive Relations on Lists. In M. Maher, editor, *Proceedings of the 13th Joint International Conference and Symposium on Logic Programming*, Bonn, Germany, pages 230-243, MIT Press, 1996.

[29] M. Kirschenbaum and L.S. Sterling. Applying Techniques to Skeletons. In J. Jacquet, editor, *Constructing Logic Programs*, pages 127-140, MIT Press, 1993.

[30] G. Nadathur and D. Miller. An Overview of λProlog. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the 5th International Conference and Symposium on Logic Programming*, Seattle, Washington, pages 810-827, MIT Press, 1988.

[31] G. Nadathur and D. Miller. Higher-Order Horn Clauses. *Journal of the ACM*, 37: 777-814, 1990.

[32] M. Perlmutter, S.D. Berhrend, F. Kuo, and A. Muller. Social Influence on Children's Problem Solving. *Developmental Psychology*, 25: 744-754, 1989.

[33] J. Piaget. Piaget's Theory. In P.H. Mussen, editor, *Carmichael's Handbook of Child Psychology (Volume I)*, pages 703-732, John Wiley & Sons, 1970.

[34] J.A. Self. Bypassing the Intractable Problem of Student Modeling. In C. Frasson and G. Gauthier, editors, *Intelligent Tutoring Systems: At the Crossroads of Artificial Intelligence and Education*, pages 107-123, Ablex, 1990.

[35] D. Sleeman and J.S. Brown, editors. *Intelligent Tutoring Systems*. Academic Press, 1982.

[36] M.W. van Someren. What's Wrong? Beginners' Problems with Prolog. *Instructional Science*, 19: 257-282, 1990.

[37] W.W. Vasconcelos and N.E. Fuchs. An Opportunistic Approach for Logic Program Analysis and Optimisation Using Enhanced Schema-Based Transformations. In M. Proietti, editor, *Proceedings of the 5th International Workshop on Logic Program Synthesis and Transformation*, Utrecht, The Netherlands, pages 174-188, Springer-Verlag, 1995.

[38] L.S. Vygotsky. *Mind in Society: The Development of Higher Psychological Processes*. Harvard University Press, 1978.

[39] T. Yokomori. Logic Program Forms. *New Generation Computing*, 4: 305-319, 1986.