16 Logic Programming in Lisp

Chapter Objectives

A Lisp-based logic programming interpreter:
- An example of meta-linguistic abstraction

Critical components of logic interpreter
Predicate Calculus like facts and rules
Horn clause form
Queries processed by unification against facts and rules
Successful goal returns unification substitutions
Supporting technology for logic interpreter
- Streams
- Stream processing
- Stream of variables substitutions filtered through conjunctive subgoals
- `gensym` used to standardize variables apart

Exercises expanding functionality of logic interpreter
- Adding `and`, `not`
- Additions of numeric and equality relations

Chapter Contents

16.1 A Simple Logic Programming Language
16.2 Streams and Stream Processing
16.3 A Stream-Based Logic Programming Interpreter

16.1 A Simple Logic Programming Language

Example

As an example of meta-linguistic abstraction, we develop a Lisp-based logic programming interpreter, using the unification algorithm from Section 15.2. Like Prolog, our logic programs consist of a database of facts and rules in the predicate calculus. The interpreter processes queries (or goals) by unifying them against entries in the logic database. If a goal unifies with a simple fact, it succeeds; the solution is the set of bindings generated in the match. If it matches the head of a rule, the interpreter recursively attempts to satisfy the rule premise in a depth-first fashion, using the bindings generated in matching the head. On success, the interpreter prints the original goal, with variables replaced by the solution bindings.

For simplicity's sake, this interpreter supports conjunctive goals and implications: or and not are not defined, nor are features such as arithmetic, I/O, or the usual Prolog built-in predicates. Although we do not implement full Prolog, and the exhaustive nature of the search and absence of the *cut* prevent the proper treatment of recursive predicates, the shell captures the basic behavior of the logic programming languages. The addition to the interpreter of the other features just mentioned is an interesting exercise.

Our logic programming interpreter supports Horn clauses, a subset of full predicate calculus (Luger 2009, Section 14.2). Well-formed formulae consist of terms, conjunctive expressions, and rules written in a Lisporiented syntax. A compound term is a list in which the first element is a predicate name and the remaining elements are the arguments. Arguments may be either constants, variables, or other compound terms. As in the discussion of `unify`, we represent variables as lists of two elements, the word `var` followed by the name of the variable. Examples of terms include:

```
(likes bill music)
(on block (var x))
(friend bill (father robert))
```

A conjunctive expression is a list whose first element is `and` and whose subsequent arguments are either simple terms or conjunctive expressions:

```
(and (smaller david sarah) (smaller peter david))
(and (likes (var x) (var y))
   (likes (var z) (var y)))
(and (hand-empty)
   (and (on block-1 block-2)
      (on block-2 table)))
```

Implications are expressed in a syntactically sweetened form that simplifies both their writing and recognition:

```
(rule if <premise> then <conclusion>)
```

where `<premise>` is either a simple or conjunctive proposition and `<conclusion>` is always a simple proposition. Examples include:

```
(rule if (and (likes (var x) (var z))
              (likes (var y) (var z)))
     then (friend (var x) (var y)))
(rule if (and (size (var x) small)
              (color (var x) red)
              (smell (var x) fragrant))
     then (kind (var x) rose))
```

The logic database is a list of facts and rules bound to a global variable, `*assertions*`. We can define an example knowledge base of `likes` relationships by a call to `setq` (we could have used the function `defvar`):

```
(setq *assertions*
     '((likes george beer)
       (likes george kate)
       (likes george kids)
       (likes bill kids)
       (likes bill music)
       (likes bill pizza)
       (likes bill wine)
       (rule if (and (likes (var x) (var z))
                     (likes (var y) (var z)))
            then (friend (var x) (var y)))))
```

The top level of the interpreter is a function, `logic-shell`, that reads goals and attempts to satisfy them against the logic database bound to `*assertions*`. Given the above database, `logic-shell` will have the following behavior, where comments follow the `;`:

```
> (logic-shell)                          ;Prompts with a ?
?(likes bill (var x))
(likes bill kids)
(likes bill music)
(likes bill pizza)
(likes bill wine)
?(likes george kate)
(likes george kate)
?(likes george taxes)          ;Failed query returns nothing.
?(friend bill george)
(friend bill george)     ;From (and(likes bill kids)
                              ;(likes george kids)).
?(friend bill roy)            ;roy not in knowledge base, fail.
?(friend bill (var x))
(friend bill george)      ;From (and(likes bill kids)
                               (likes george kids)).
(friend bill bill)        ;From (and(likes bill kids)
                                ;(likes bill kids)).
(friend bill bill)       ;From (and(likes bill music)
                               ;(likes bill music)).
(friend bill bill)       ;From (and(likes bill pizza)
                               ;(likes bill pizza)).
(friend bill bill)        ;From (and(likes bill wine)
                                ;(likes bill wine)).
?quit
bye
>
```

Before discussing the implementation of the logic programming interpreter, we introduce the *stream* data type.

16.2 Streams and Stream Processing

As the preceding example suggests, even a small knowledge base can produce complex behaviors. It is necessary not only to determine the truth or falsity of a goal but also to determine the variable substitutions that make that goal be true in the knowledge base. A single goal can match with different facts, producing different substitution sets; conjunctions of goals require that all conjuncts succeed and also that the variable bindings be consistent throughout. Similarly, rules require that the substitutions formed in matching a goal with a rule conclusion be made in the rule premise when it is solved. The management of these multiple substitution sets is the major source of complexity in the interpreter. Streams help address this complexity by focusing on the movement of a sequence of candidate variable substitutions through the constraints defined by the logic database.

A *stream* is a sequence of data objects. Perhaps the most common example of stream processing is a typical interactive program. The data from the keyboard are viewed as an endless sequence of characters, and the program is organized around reading and processing the *current* character from the input stream. Stream processing is a generalization of this idea: streams need not be produced by the user; they may also be generated and modified by functions. A *generator* is a function that produces a continuing stream of data objects. A *map function* applies some function to each of the elements of a stream. A *filter* eliminates selected elements of a stream according to the constraints of some predicate.

The solutions returned by an inference engine may be represented as a stream of different variable substitutions under which a goal follows from a knowledge base. The constraints defined by the knowledge base are used to modify and filter a stream of candidate substitutions, producing the result. Consider, for example, the conjunctive goal (using the logic database from the preceding section):

```
(and (likes bill (var z))
       (likes george (var z)))
```

The stream-oriented view regards each of the conjuncts in the expression as a *filter* for a stream of substitution sets. Each set of variable substitutions in the stream is applied to the conjunct and the result is matched against the knowledge base. If the match fails, that set of substitutions is eliminated from the stream; if it succeeds, the match may create new sets of substitutions by adding new bindings to the original substitution set.

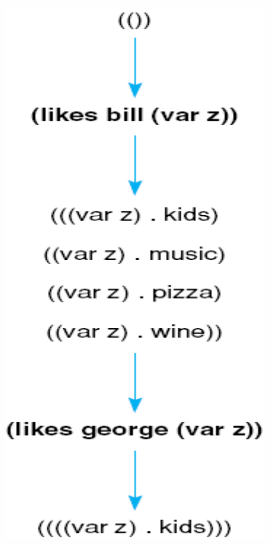

Figure 16.1 A stream of variable substitutions filtered through conjunctive subgoals.

Figure 16.1 illustrates the stream of substitutions passing through this conjunctive goal. It begins with a stream of candidate substitutions containing only the empty substitution set and grows after the first proposition matches against multiple entries in the database. It then shrinks to a single substitution set as the second conjunct eliminates substitutions that do not allow `(likes` `george (var z))` to succeed. The resulting stream, `((((var z) . kids)))`, contains the only variable substitution that allows both subgoals in the conjunction to succeed in the knowledge base.

As this example illustrates, a goal and a single set of substitutions may generate several new substitution sets, one for each match in the knowledge base. Alternatively, a goal will eliminate a substitution set from the stream if no match is found. The stream of substitution sets may grow and shrink as it passes through a series of conjuncts.

The basis of stream processing is a set of functions to create, augment, and access the elements of a stream. We can define a simple set of stream functions using lists and the standard list manipulators. The functions that constitute a list-based implementation of the stream data type are:

```
                     ;cons-stream adds a new first element to a stream.
(defun cons-stream (element stream)
   (cons element stream))
                  ;head-stream returns the first element of the stream.
(defun head-stream (stream) (car stream))
             ;tail-stream returns the stream with first element deleted.
(defun tail-stream (stream) (cdr stream))
                       ;empty-stream-p is true if the stream is empty.
(defun empty-stream-p (stream) (null stream))
                     ;make-empty-stream creates an empty stream.
(defun make-empty-stream ( ) nil)
                          ;combine-stream appends two streams.
(defun combine-streams (stream1 stream2)
     (cond ((empty-stream-p stream1) stream2)
          (t (cons-stream (head-stream stream1)
             (combine-streams
                (tail-stream stream 1)
                stream2)))))
```

Although the implementation of streams as lists does not allow the full power of stream-based abstraction, the definition of a stream data type helps us to view the program from a data flow point of view. For many problems, such as the logic programming interpreter of Section 16.3, this provides the programmer with a powerful tool for organizing and simplifying the code. In Section 17.1 we discuss some limitations of this list-based implementation of streams and present an alternative approach using streams with delayed evaluation.

16.3 A Stream-Based Logic Programming Interpreter

We invoke the interpreter through a function called `logic-shell`, a straightforward variation of the `read-eval-print` loop discussed in Section 15.3. After printing a prompt, "`?`", it reads the next s-expression entered by the user and binds it to the symbol `goal`. If `goal` is equal to `quit`, the function halts; otherwise, it calls `solve` to generate a stream of substitution sets that satisfy the `goal`. This stream is passed to `print-solutions`, which prints the `goal` with each of these different substitutions. The function then recurs. `logic-shell` is defined:

```
(defun logic-shell ( )
      (print '? )
      (let ((goal (read)))
           (cond ((equal goal 'quit) 'bye)
                 (t (print-solutions goal
                   (solve goal nil))
                   (terpri)
                   (logic-shell)))))
```

`solve` is the heart of the interpreter. `solve` takes a goal and a set of substitutions and finds all solutions that are consistent with the knowledge base. These solutions are returned as a stream of substitution sets; if there are no matches, solve returns the empty stream. From the stream processing point of view, solve is a source, or generator, for a stream of solutions. `solve` is defined by:

```
(defun solve (goal substitutions)
      (declare (special *assertions*))
      (if (conjunctive-goal-p goal)
          (filter-through-conj-goals (body goal)
               (cons-stream substitutions
                  (make-empty-stream)))
          (infer goal substitutions *assertions*)))
```

The declaration `special` tells the Lisp compiler that `*assertions*` is a special, or global, variable and should be bound dynamically in the environment in which `solve` is called. (This special declaration is not required in many modern versions of Lisp.)

`solve` first tests whether the goal is a conjunction; if it is, `solve` calls `filter-through-conj-goals` to perform the filtering described in Section 16.2. If `goal` is not a conjunction, `solve` assumes it is a simple goal and calls `infer`, defined below, to `solve` it against the knowledge base. `solve` calls `filter-through-conj-goals` with the body of the conjunction (i.e., the sequence of conjuncts with the and operator removed) and a stream that contains only the initial substitution set. The result is a stream of substitutions representing all of the solutions for this goal. We define `filter-through-conj-goals` by:

```
(defun filter-through-conj-goals (goals
         substitution-stream)
      (if (null goals) substitution-stream
      (filter-through-conj-goals (cdr goals)
                (filter-through-goal (car goals)
                  substitution-stream))))
```

If the list of goals is empty, the function halts, returning `substitution-stream` unchanged. Otherwise, it calls `filter-`

through-goal to filter substitution-stream through the first goal on the list. It passes this result on to a recursive call to filter-through-conj-goals with the remainder of the goal list. Thus, the stream is passed through the goals in left-to-right order, growing or shrinking as it passes through each goal.

filter-through-goal takes a single goal and uses it as a filter to the stream of substitutions. This filtering is done by calling solve with the goal and the first set of substitutions in the substitution-stream. The result of this call to solve is a stream of substitutions resulting from matches of the goal against the knowledge base. This stream will be empty if the goal does not succeed under any of the substitutions contained in the stream, or it may contain multiple substitution sets representing alternative bindings. This stream is combined with the result of filtering the tail of the input stream through the same goal:

```
(defun filter-through-goal
          (goal substitution-stream)
      (if (empty-stream-p substitution-stream)
      (make-empty-stream)
      (combine-streams
          (solve goal
          (head-stream substitution-stream))
            (filter-through-goal goal
                (tail-stream substitution-stream)))))
```

To summarize, filter-through-conj-goals passes a stream of substitution sets through a sequence of goals, and filter-through-goal filters substitution-stream through a single goal. A recursive call to solve solves the goal under each substitution set.

Whereas solve handles conjunctive goals by calling filter-through-conj-goals, simple goals are handled by infer, defined next, which takes a goal and a set of substitutions and finds all solutions in the knowledge base, kb, infer's third parameter, a database of logic expressions. When solve first calls infer, it passes the knowledge base contained in the global variable *assertions*. infer searches kb sequentially, trying the goal against each fact or rule conclusion.

The recursive implementation of infer builds the backward-chaining search typical of Prolog and many expert system shells. It first checks whether kb is empty, returning an empty stream if it is. Otherwise, it binds the first item in kb to the symbol assertion using a let* block. let* is like let except it is guaranteed to evaluate the initializations of its local variables in sequentially nested scopes, i.e., it provides an order to the binding and visibility of preceding variables. It also defines the variable match: if assertion is a rule, let* initializes match to the substitutions required to unify the goal with the conclusion of the rule; if assertion is a fact, let* binds match to those substitutions required to unify assertion with goal. After attempting to unify the goal with the first element of the knowledge base, infer tests whether the unification succeeded. If it failed to match, infer recurs, attempting to solve the goal using the remainder of the knowledge base. If the unification succeeded and assertion is a rule, `infer` calls `solve` on the premise of the rule using the augmented set of substitutions bound to `match`. `combine-stream` joins the resulting stream of solutions to that constructed by calling `infer` on the rest of the knowledge base. If `assertion` is not a rule, it is a fact; `infer` adds the solution bound to `match` to those provided by the rest of the knowledge base. Note that once the goal unifies with a fact, it is solved; this terminates the search. We define `infer`:

```
(defun infer (goal substitutions kb)
  (if (null kb)
     (make-empty-stream)
     (let* ((assertion
              (rename-variables (car kb)))
          (match (if (rulep assertion)
              (unify goal (conclusion assertion)
                  substitutions)
              (unify goal assertion substitutions))))
              (if (equal match 'failed)
                 (infer goal substitutions (cdr kb))
              (if (rulep assertion)
                 (combine-streams
                   (solve (premise assertion) match)
                   (infer goal substitutions
                          (cdr kb)))
                 (cons-stream match
                    (infer goal substitutions
                                  (cdr kb))))))))
```

Before the first element of `kb` is bound to `assertion`, it is passed to `rename-variables` to give each variable a unique name. This prevents name conflicts between the variables in the goal and those in the knowledge base entry; e.g., if `(var x)` appears in a goal, it must be treated as a different variable than a `(var x)` that appears in the rule or fact. (This notion of standardizing variables apart is an important component of automated reasoning in general. Luger (2009, Section 14.2) demonstrates this in the context of resolution refutation systems). The simplest way to handle this is by renaming all variables in `assertion` with unique names. We define `rename-variables` at the end of this section.

This completes the implementation of the core of the logic programming interpreter. To summarize, `solve` is the top-level function and generates a stream of substitution sets (`substitution-stream`) that represent solutions to the goal using the knowledge base. `filter-through-conj-goals` solves conjunctive goals in a left-to-right order, using each goal as a filter on a stream of candidate solutions: if a goal cannot be proven true against the knowledge base using a substitution set in the stream, `filter-through-conj-goals` eliminates those substitutions from the stream. If the goal is a simple literal, `solve` calls `infer` to generate a stream of all substitutions that make the goal succeed against the knowledge base. Like Prolog, our logic programming interpreter takes a goal and finds all variable bindings that make it true against a given knowledge base.

All that remain are functions for accessing components of knowledge base entries, managing variable substitutions, and printing solutions. `print-solutions` takes as arguments a `goal` and a `substitution-stream`. For each set of substitutions in the stream, it prints the goal with variables replaced by their bindings in the substitution set.

```
(defun print-solutions (goal substitution-stream)
     (cond ((empty-stream-p substitution-stream)
                          nil)
            (t (print (apply-substitutions goal
                     (head-stream
                    substitution-stream)))
              (terpri)
              (print-solutions goal
                 (tail-stream substitution-stream)))))
```

The replacement of variables with their values under a substitution set is done by `apply-substitutions`, which does a car-cdr recursive tree walk on a pattern. If the pattern is a constant (`is-constant-p`), it is returned unchanged. If it is a variable (`varp`), `apply-substitutions` tests if it is bound. If it is unbound, the variable is returned; if it is bound, `apply-substitutions` calls itself recursively on the value of this binding. Note that the binding value may be either a constant, another variable, or a pattern of arbitrary complexity:

```
(defun apply-substitutions
          (pattern substitution-list)
    (cond ((is-constant-p pattern) pattern)
       ((varp pattern)
          (let ((binding
                    (get-binding pattern
                   substitution-list)))
            (cond (binding (apply-substitutions
                    (get-binding-value binding)
                     substitution-list))
                 (t pattern))))
         (t (cons (apply-substitutions
                 (car pattern)
                substitution-list)
             (apply-substitutions (cdr pattern)
                substitution-list)))))
```

`infer` renamed the variables in each knowledge base entry before matching it with a goal. This is necessary, as noted above, to prevent undesired name collisions in matches. For example, the goal `(p a (var x))` should match with the knowledge base entry `(p (var x) b)`, because the scope of each `(var x)` is restricted to a single expression. As unification is defined, however, this match will not occur. Name collisions are prevented by giving each variable in an expression a unique name. The basis of our renaming scheme is a Common Lisp built-in function called `gensym` that takes no arguments; each time it is called, it returns a unique symbol consisting of a number preceded by `#:G`. For example:

```
> (gensym)
#:G4
> (gensym)
#:G5
> (gensym)
#:G6
>
```

Our renaming scheme replaces each variable name in an expression with the result of a call to `gensym`. `rename-variables` performs certain initializations (described below) and calls `rename-rec` to make substitutions recursively in the pattern. When a variable (`varp`) is encountered, the function rename is called to return a new name. To allow multiple occurrences of a variable in a pattern to be given consistent names, each time a variable is renamed, the new name is placed in an association list bound to the special variable `*name-list*`. The special declaration makes all references to the variable dynamic and shared among these functions. Thus, each access of `*name-list*` in rename will access the instance of `*name-list*` declared in `rename-variables`. `rename-variables` initializes `*name-list*` to `nil` when it is first called on an expression. These functions are defined:

```
(defun rename-variables (assertion)
     (declare (special *name-list*))
     (setq *name-list* nil)
     (rename-rec assertion))

(defun rename-rec (exp)
     (declare (special *name-list*))
     (cond ((is-constant-p exp) exp)
           ((varp exp) (rename exp))
           (t (cons (rename-rec (car exp))
                (rename-rec (cdr exp))))))

(defun rename (var)
     (declare (special *name-list*))
```

```
          (list ’var (or (cdr (assoc var *name-list*
                              :test #’equal))
              (let ((name (gensym)))
                 (setq *name-list*
                    (acons var name *name-list*))
                name))))
```

The final functions access components of rules and goals and are self-explanatory:

```
   (defun premise (rule) (nth 2 rule))

   (defun conclusion (rule) (nth 4 rule))

   (defun rulep (pattern)
        (and (listp pattern) (equal (nth 0 pattern)
            ’rule)))

   (defun conjunctive-goal-p (goal)
        (and (listp goal) (equal (car goal) ’and)))

   (defun body (goal) (cdr goal))
```

In Chapter 17 we extend the ideas of Chapter 16 to delayed evaluation using lexical closures. Finally we build a goal-driven expert system shell in Lisp.

Exercises

1. Expand the logic programming interpreter to include Lisp `write` statements. This will allow rules to print messages directly to the user. Hint: modify `solve` first to examine if a goal is a `write` statement. If it is, evaluate the `write` and return a stream containing the initial substitution set.

2. Rewrite print-solutions in the logic programming interpreter so that it prints the first solution and waits for a user response (such as a carriage return) before printing the second solution.

3. Implement the general map and filter functions, `map-stream` and `filter-stream`, described in Section 16.3.

4. Expand the logic programming interpreter to include `or` and `not` relations. This will allow rules to contain more complex relationships between its premises.

5. Expand the logic programming language to include arithmetic comparisons, `=`, `<`, and `>`. Hint: as

in Exercise 1, modify `solve` to detect these comparisons before calling `infer`. If an expression is a comparison, replace any variables with their values and evaluate it. If it returns `nil`, `solve` should return the empty stream; if it returns non-`nil`, `solve` should return a stream containing the initial substitution set. Assume that the expressions do not contain unbound variables.

6. For a more challenging exercise, expand the logic programming interpreter to define **=** so that it will function like the Prolog is operator and assign a value to an unbound variable and simply do an equality test if all elements are bound.