# 15 Unification and Embedded Languages in Lisp

**Chapter Objectives**

Pattern matching in Lisp:
- Database examples

Full unification as required for Predicate Calculus problem solving
- Needed for applying inference rules
- General structure mapping
- Recursive for embedded structures

Building interpreters and embedded languages
- Example: `read-eval-print` loop

Example: infix interpreter

**Chapter Contents**



## 15.1 Pattern Matching: Introduction

In Chapter 15 we first design an algorithm for matching patterns in general list structures. This is the basis for the unify function which supports full pattern matching and the return of sets of unifying substitutions for matching patterns in predicate calculus expressions. We will see this as the basis for interpreters for logic programming and rule-based systems in Lisp, presented in Chapters 16 and 17.

**Pattern Matching in Lisp**

Pattern matching is an important AI methodology that has already been discussed in the Prolog chapters and in the presentation of production systems. In this section we implement a recursive pattern matcher and use it to build a pattern-directed retrieval function for a simple database.

The heart of this retrieval system is a function called `match`, which takes as arguments two s-expressions and returns `t` if the expressions match. Matching requires that both expressions have the same *structure* as well as having identical atoms in corresponding positions. In addition, match allows the inclusion of variables, denoted by `?`, in an s-expression. Variables are allowed to match with any s-expression, either a list or an atom, but do not save bindings, as with full unification (next). Examples of the desired behavior for `match` appear below. If the examples seem reminiscent of the Prolog examples in Part II, this is because `match` is actually a simplified version of the unification algorithm that forms the heart of the Prolog environment, as well as of many other pattern-directed AI systems. We will later expand `match` into the full unification algorithm by allowing named variables and returning a list of bindings required for a match.

```
> (match '(likes bill wine) '(likes bill wine))
t
> (match '(likes bill wine) '(likes bill milk))
nil
> (match '(likes bill ?) '(likes bill wine))
t
> (match '(likes ? wine) '(likes bill ?))
t
> (match '(likes bill ?) '(likes bill (prolog lisp
smalltalk))
t
> (match '(likes ?) '(likes bill wine))
nil
```

`match` is used to define a function called `get-matches`, which takes as arguments two s-expressions. The first argument is a pattern to be matched against elements of the second s-expression, which must be a list. `get-matches` returns a list of the elements of the list that match the first argument. In the example below, `get-matches` is used to retrieve records from an employee database as described earlier in Part III.

Because the database is a large and relatively complex s-expression, we have bound it to the global variable `*database*` and use that variable as an argument to `get-matches`. This was done to improve readability of the examples.

```
> (setq *database* '(((lovelace ada) 50000.00 1234)
          ((turing alan) 45000.00 3927)
          ((shelley mary) 35000.00 2850)
          ((vonNeumann john) 40000.00 7955)
          ((simon herbert) 50000.00 1374)
          ((mccarthy john) 48000.00 2864)
          ((russell bertrand) 35000.00 2950))
*database*
> (get-matches '((turing alan) 45000.00 3927)
*database*)
((turing alan) 45000.00 3927)
> (get-matches '(? 50000.00 ?) *database*)
;people who make 50000
(((lovelace ada) 50000.00 1234) ((simon herbert)
50000.00 1374))
> (get-matches '((? john) ? ?) *database*)
;all people named john
(((vonNeumann john) 40000.00 7955) ((mccarthy john)
48000.00 2864))
```

We implement `get-matches` using `cdr` recursion: each step attempts to match the target pattern with the first element of the database (the `car` of the list). If the is a match, the function will `cons` it onto the list of

matches returned by the recursive call to form the answer for the pattern. `get-matches` is defined:

```
(defun get-matches (pattern database)
   (cond ((null database) ( ))
      ((match pattern (car database))
         (cons (car database)
               (get-matches pattern
                  (cdr database))))
        (t (get-matches pattern
                  (cdr database)))))
```

The heart of the system is the `match` function, a predicate that determines whether or not two s-expressions containing variables actually match. `match` is based on the idea that two lists match if and only if their respective `car`s and `cdr`s match, suggesting a `car-cdr` recursive scheme for the algorithm.

The recursion terminates when either of the arguments is atomic (this includes the empty list, `nil`, which is both an atom and a list). If both patterns are the same atom or if one of the patterns is a variable atom, `?`, which can match with anything, then termination is with a successful match; otherwise, the match will fail. Notice that if either of the patterns is a variable, the other pattern need not be atomic; variables may match with variables or with s-expressions of arbitrary complexity.

Because the handling of the terminating conditions is complex, the implementation of `match` uses a function called `match-atom` that takes two arguments, one or both of which is an `atom`, and checks to see whether the patterns match. By hiding this complexity in `match-atom` the `car-cdr` recursive structure of match is more apparent:

```
(defun match (pattern1 pattern2)
   (cond (or (atom pattern1) (atom pattern2))
             (match-atom pattern1 pattern2))
      (t (and (match (car pattern1) (car pattern2))
              (match (cdr pattern1)
                     (cdr pattern2))))))
```

The implementation of `match-atom` makes use of the fact that when it is called, at least one of the arguments is an `atom`. Because of this assumption, a simple test for equality of patterns is all that is needed to test that both patterns are the same `atom` (including both being a variable); it will fail either if the two patterns are different atoms or if one of them is nonatomic. If the first test fails, the only way match can succeed is if one of the patterns is a variable. This check constitutes the remainder of the function definition.

Finally, we define a function `variable-p` to test whether or not a pattern is a variable. Treating variables as an abstract data type now will simplify later extensions to the function, for example, the extension of the function to named variables as in Prolog.

```
(defun match-atom (pattern1 pattern2)
     (or (equal pattern1 pattern2)
         (variable-p pattern1)
         (variable-p pattern2)))
(defun variable-p (x) (equal x '?))
```

**A Recursive Unification Function**

We have just completed the implementation of a recursive pattern-matching algorithm that allowed the inclusion of unnamed variables in patterns. Our next step will be to extend this simple pattern matcher into the full unification algorithm. See Luger (2009, Section 2.3) for a pseudocode version of this algorithm.

The function, `unify`, allows named variables in both of the patterns to be matched, and returns a substitution list of the variable bindings required for the match. This unification function is the basis of the inference systems for logic and expert system interpreters developed later in Chapters 16 and 17.

As follows the definition of unification, patterns are either constants, variables, or list structures. We will distinguish variables from one another by their names. Named variables will be represented as lists of the form `(var <name>)`, where `<name>` is usually an atomic symbol. `(var x)`, `(var y)`, and `(var newstate)` are all examples of legal variables.

The function `unify` takes as arguments two patterns to be matched and a set of variable substitutions (bindings) to be employed in the match. Generally, this set will be empty (`nil`) when the function is first called. On a successful match, `unify` returns a (possibly empty) set of substitutions required for a successful match. If no match was possible, `unify` returns the symbol `failed`; `nil` is used to indicate an empty substitution set, i.e., a match in which no substitutions were required. An example of the behavior of `unify`, with comments, is:

```
> (unify '(p a (var x)) '(p a b) ( ))
(((var x) . b))
             ;Returns substitution of b for (var x).
> (unify '(p (var y) b) '(p a (var x)) ( ))
(((var x) . b) ((var y) . a))
                 ;Variables appear in both patterns.
> (unify '(p (var x)) '(p (q a (var y))) ( ))
(((var x) q a (var y)))
            ;Variable is bound to a complex pattern.
> (unify '(p a) '(p a) ( ))
nil
            ;nil indicates no substitution required.
> (unify '(p a) '(q a) ( ))
failed
         ;Returns atom "failed" to indicate failure.
```

We will explain the "." notation, as in `((var x) . b)`, after we present the function `unify`. `unify`, like the pattern matcher of earlier in this section, uses a `car-cdr` recursive scheme and is defined by:

```
(defun unify (pattern1 pattern2 substitution-list)
     (cond ((equal substitution-list 'failed)
              'failed)
           ((varp pattern1)
            (match-var pattern1
                    pattern2 substitution-list))
           ((varp pattern2)
            (match-var pattern2
                    pattern1 substitution-list))
           ((is-constant-p pattern1)
             (cond ((equal pattern1 pattern2)
                    substitution-list)
                   (t 'failed)))
           ((is-constant-p pattern2) 'failed)
            (t (unify (cdr pattern1)
                  (cdr pattern2)
                  (unify (car pattern1)
                    (car pattern2)
                       substitution-list)))))
```

On entering `unify`, the algorithm first checks whether the `substitution-list` is `equal` to `failed`. This could occur if a prior attempt to unify the `car`s of two patterns had failed. If this condition is met, the entire unification operation fails, and the function returns `failed`.

Next, if either pattern is a variable, the function `match-var` is called to perform further checking and possibly add a new binding to `substitution-list`. If neither pattern is a variable, `unify` tests whether either is a constant, returning the unchanged substitution list if they are the same constant, otherwise it returns `failed`.

The last item in the `cond` statement implements the tree-recursive decomposition of the problem. Because all other options have failed, the function concludes that the patterns are lists the must be unified recursively. It does this using a standard tree-recursive scheme: first, the `car`s of the patterns are unified using the bindings in `substitution-list`. The result is passed as the third argument to the call of `unify` on the `cdr`s of both patterns. This allows the variable substitutions made in matching the `car`s to be applied to other occurrences of those variables in the `cdr`s of both patterns.

`match-var`, for the case of matching a variable and a pattern, is defined:

```
(defun match-var (var pattern substitution-list)
   (cond ((equal var pattern) substitution-list)
         (t (let ((binding
                   (get-binding var substitution-list)))
              (cond (binding (unify
                              (get-binding-value binding)
                               pattern substitution-list))
                    ((occursp var pattern) 'failed)
                     (t (add-substitution var pattern
                                          substitution-list)))))))
```

match-var first checks whether the variable and the pattern are the same; unifying a variable with itself requires no added substitutions, so substitution-list is returned unchanged.

If `var` and `pattern` are not the same, `match-var` checks whether the variable is already bound. If a binding exists, `unify` is called recursively to match the value of the binding with `pattern`. Note that this binding value may be a constant, a variable, or a pattern of arbitrary complexity; requiring a call to the full unification algorithm to complete the match.

If no binding currently exists for `var`, the function calls `occursp` to test whether `var` appears in `pattern`. As explained in (Luger 2009), the occurs check is needed to prevent attempts to unify a variable with a pattern containing that variable, leading to a circular structure. For example, if `(var x)` was bound to `(p (var x))`, any attempt to apply those substitutions to a pattern would result in an infinite structure. If `var` appears in pattern, `match-var` returns `failed`; otherwise, it adds the new substitution pair to `substitution-list` using `add-substitution`.

`unify` and `match-var` are the heart of the unification algorithm. `occursp` (which performs a tree walk on a pattern to find any occurrences of the variable in that pattern), `varp`, and `is-constant-p` (which test whether their argument is a variable or a constant, respectively) appear below. Functions for handling substitution sets are discussed below.

```
(defun occursp (var pattern)
   (cond ((equal var pattern) t)
      ((or (varp pattern)
           (is-constant-p pattern))
        nil)
      (t (or (occursp var (car pattern))
             (occursp var (cdr pattern))))))

(defun is-constant-p (item)
     (atom item))
```

```
(defun varp (item)
     (and (listp item)
          (equal (length item) 2)
          (equal (car item) 'var)))
```

Sets of substitutions are represented using a built-in Lisp data type called the *association list* or *a-list*. This is the basis for the functions `add-substitutions`, `get-binding`, and `binding-value`. An association list is a list of data records, or *key/data* pairs. The `car` of each record is a *key* for its retrieval; the `cdr` of each record is called the *datum*. The datum may be a list of values or a single atom. Retrieval is implemented by the function `assoc`, which takes as arguments a key and an association list and returns the first member of the association list that has the key as its `car`. An optional third argument to `assoc` specifies the test to be used in comparing keys. The default test is the Common Lisp function `eql`, a form of equality test requiring that two arguments be the same object (i.e., either the same memory location or the same numeric value). In implementing substitution sets, we specify a less strict test, `equal`, which requires only that the arguments match syntactically (i.e., are designated by identical names). An example of `assoc`'s behavior appears next:

```
> (assoc 3 '((1 a) (2 b) (3 c) (4 d)))
(3 c)
> (assoc 'd '((a b c) (b c d e) (d e f) (c d e))
:test #'equal)
(d e f)
> (assoc 'c '((a . 1) (b . 2) (c . 3) (d . 4)) :test
#'equal)
(c . 3)
```

Note that `assoc` returns the entire record matched on the key; the datum may be retrieved from this list by the `cdr` function. Also, notice that in the last call the members of the a-list are not lists but a structure called *dotted pairs*, e.g., `(a . 1)`.

The dotted pair, or `cons` pair, is actually the fundamental constructor in Lisp. It is the result of `cons`ing one s-expression onto another; the list notation that we have used throughout the chapter is just a notational variant of dotted pairs. For example, the value returned by `(cons 1 nil)` is actually `(1 . nil)`; this is equivalent to `(1)`. Similarly, the list `(1 2 3)` may be written in dotted pair notation as `(1 . (2 . ( 3 . nil)))`. Although the actual effect of a `cons` is to create a dotted pair, the list notation is cleaner and is generally preferred.

If two atoms are `cons`ed together, the result is always written using dotted pair notation. The `cdr` of a dotted pair is the second element in the pair, rather than a list containing the second atom. For example:

```
> (cons 'a 'b)
(a . b)
```

```
> (car '(a . b))
a
> (cdr '(a . b))
b
```

Dotted pairs occur naturally in association lists when one atom is used as a key for retrieving another atom, as well as in other applications that require the formation and manipulation of pairs of atomic symbols. Because unifications often substitute a single atom for a variable, dotted pairs appear often in the association list returned by the unification function.

Along with `assoc`, Common Lisp defines the function `acons`, which takes as arguments a key, a datum, and an association list and returns a new association list whose first element is the result of `cons`ing the key onto the datum. For example:

```
> (acons 'a 1 nil)
((a . 1))
```

Note that when `acons` is given two atoms, it adds their `cons` to the association list:

```
> (acons 'pets '(emma jack clyde)
     '((name . bill) (hobbies music skiing movies)
        (job . programmer)))
((pets emma jack clyde) (name . bill) (hobbies music
skiing movies)(job . programmer))
```

The members of an association list may themselves be either dotted pairs or lists.

Association lists provide a convenient way to implement a variety of tables and other simple data retrieval schemes. In implementing the unification algorithm, we use association lists to represent sets of substitutions: the keys are the variables, and the data are the values of their bindings. The datum may be a simple variable or constant or a more complicated structure.

Using association lists, the substitution set functions are defined:

```
(defun get-binding (var substitution-list)
     (assoc var substitution-list :test #'equal))
(defun get-binding-value (binding) (cdr binding))
(defun add-substitution (var pattern
           substitution-list)
     (acons var pattern substitution-list))
```

This completes the implementation of the unification algorithm. We will use the unification algorithm again in Section 15.1 to implement a simple Prolog in Lisp interpreter, and again in Section 16.2 to build an expert system shell.

## 15.2 Interpreters and Embedded Languages

The top level of the Lisp interpreter is known as the *read-eval-print* loop. This describes the interpreter's behavior in reading, evaluating, and printing the value of s-expressions entered by the user. The `eval` function, defined in Section 11.2, is the heart of the Lisp interpreter; using `eval`, it is possible to write Lisp's top-level `read-eval-print` loop in Lisp itself. In the next example, we develop a simplified version of this loop. This version is simplified chiefly in that it does not have the error-handling abilities of the built-in loop, although Lisp does provide the functionality needed to implement such capabilities.

To write the `read-eval-print` loop, we use two more Lisp functions, `read` and `print`. `read` is a function that takes no parameters; when it is evaluated, it returns the next s-expression entered at the keyboard. `print` is a function that takes a single argument, evaluates it, and then prints that result to standard output. Another function that will prove useful is `terpri`, a function of no arguments that sends a newline character to standard output. `terpri` also returns a value of `nil` on completion. Using these functions, the `read-eval-print` loop is based on a nested s-expression:

```
(print (eval (read)))
```

When this is evaluated, the innermost s-expression, (`read`), is evaluated first. The value returned by the `read`, the next s-expression entered by the user, is passed to `eval`, where it is evaluated. The result of this evaluation is passed to `print`, where it is sent to the display screen. To complete the loop we add a `print` expression to output the prompt, a `terpri` to output a newline after the result has been printed, and a recursive call to repeat the cycle. Thus, the final `read-eval-print` loop is defined:

```
(defun my-read-eval-print ( )
     (print ':)  ;output a prompt (":")
     (print (eval (read)))
     (terpri) ;output a newline
     (my-read-eval-print))   ;do it all again
```

This may be used "on top of" the built-in interpreter:

```
> (my-read-eval-print)
:(+ 1 2);note the alternative prompt
3
:                                                    ;etc
```

As this example illustrates, by making functions such as `quote` and `eval` available to the user, Lisp gives the programmer a high degree of control over the handling of functions. Because Lisp programs and data are both represented as s-expressions, we may write programs that perform any desired manipulations of Lisp expressions prior to evaluating them. This underlies much of Lisp's power as an imperative representation language because it allows arbitrary Lisp code to be stored, modified, and evaluated when needed. It also makes it simple to write specialized interpreters that

may extend or modify the behavior of the built-in Lisp interpreter in some desired fashion. This capability is at the heart of many Lisp-based expert systems, which read user queries and respond to them according to the expertise contained in their knowledge base.

As an example of the way in which such a specialized interpreter may be implemented in Lisp, we modify `my-read-eval-print` so that it evaluates arithmetic expressions in an infix rather than a prefix notation, as we see in the following example (note the modified prompt, `infix->`):

```
infix-> (1 + 2)
3
infix-> (7 – 2)
5
infix-> ((5 + 2) * (3 – 1))   ;Loop handles nesting.
15
```

To simplify the example, the infix interpreter handles only arithmetic expressions. A further simplification restricts the interpreter to binary operations and requires that all expressions be fully parenthesized, eliminating the need for more sophisticated parsing techniques or worries about operator precedence. However, it does allow expressions to be nested to arbitrary depth and handles Lisp's binary arithmetic operators.

We modify the previously developed `read-eval-print` loop by adding a function that translates infix expressions into prefix expressions prior to passing them on to `eval`. A first attempt at writing this function might look like:

```
(defun simple-in-to-pre (exp)
     (list (nth 1 exp)
               ;Middle element becomes first element.
     (nth 0 exp)
     ;first operand
     (nth 2 exp)
     ;second operand
```

`simple-in-to-pre` is effective in translating simple expressions; however, it is not able to correctly translate nested expressions, that is, expressions in which the operands are themselves infix expressions. To handle this situation properly, the operands must also be translated into prefix notation. Recursion is halted by testing the argument to determine whether it is a number, returning it unchanged if it is. The completed version of the `infix-to-prefix` translator is:

```
(defun in-to-pre (exp)
   (cond ((numberp exp) exp)
           (t (list (nth 1 exp)
                    (in-to-pre (nth 0 exp))
                (in-to-pre (nth 2 exp))))))
```

Using this translator, the `read-eval-print` loop may be modified to interpret infix expressions, as defined next:

```
(defun in-eval ( )
      (print 'infix->)
      (print (eval (in-to-pre (read))))
      (terpri)
      (in-eval))
```

This allows the interpretation of binary expressions in infix form:

```
> (in-eval)
infix->(2 + 2)
4
infix->((3 * 4) – 5)
7
```

In the above example, we have implemented a new language in Lisp, the language of infix arithmetic. Because of the facilities Lisp provides for symbolic computing (lists and functions for their manipulation) along with the ability to control evaluation, this was much easier to do than in many other programming languages. This example illustrates an important AI programming methodology, that of *meta-linguistic abstraction*.

Very often in AI programming, a problem is not completely understood, or the program required to solve a problem is extremely complex. *Meta-linguistic abstraction* uses the underlying programming language, in this case, Lisp, to implement a specialized, high-level language that may be more effective for solving a particular class of problems. The term "meta-linguistic abstraction" refers to our use of the base language to implement this other programming language, rather than to directly solve the problem. As we saw in Chapter 5, Prolog also gives the programmer the power to create meta-level interpreters. The power of meta-interpreters to support programming in complex domains was discussed in Part I.

## Exercises

1. Newton's method for solving roots takes an estimate of the value of the root and tests it for accuracy. If the guess does not meet the required tolerance, it computes a new estimate and repeats. Pseudo-code for using Newton's method to get the square root of a number is:

```
function root-by-newtons-method (x, tolerance)
guess := 1;
repeat
      guess := 1/2(guess + x/guess)
until absolute-value(x – guess  guess) < tolerance
```

Write a recursive Lisp function to compute square roots by Newton's method.

2. Write a random number generator in Lisp. This function must maintain a global variable, seed, and return a different random number each time the function is called. For a description of a reasonable random number algorithm, consult any basic algorithms text.

3. Test the `unify` form of Section 15.1 with five different examples of your own creation.

4. Test the `occursp` form of Section 15.1 on five different examples of your own creation

5. Write a binary post-fix interpreter that takes arbitrarily complex structures in post-fix form and evaluates them. Two examples of post-fix are `(3 4 +)` and `(6 (5 4 +) *)`.