Chapter 6
SELF-DEFINITION OF PROGRAMMING LANGUAGES

The execution of a program written in a high-level language provides an informal, operational specification of the program. Two primary approaches are used to implement high-level languages: as an interpreter for the language or as a compiler for the language. Since interpreters and compilers are able to process any syntactically legal program, they themselves can provide an operational definition for a programming level language. Program translation will be discussed in Chapter 7. In this chapter we focus on program interpreters—in particular, a Lisp interpreter written in Lisp and a Prolog interpreter written in Prolog. In each case, the interpreter itself is written in the programming language it is interpreting. We call such an approach an operational self-definition of a programming language and refer to the implementation as a **metacircular interpreter**.

6.1 SELF-DEFINITION OF LISP

Lisp, initially developed by John McCarthy in 1958, is the second oldest programming language (after Fortran) in common use today. In the early 1960s McCarthy realized that the semantics of Lisp can be defined in terms of a few Lisp primitives and that an interpreter for Lisp can be written as a very small, concise Lisp program. Such an interpreter, referred to as a metacircular interpreter, can handle function definitions, parameter passing, and recursion as well as simple S-expressions of Lisp. The small size of the interpreter is striking, considering the thousands of lines of code needed to write a compiler for an imperative language.

We have elected to construct the interpreter in Scheme, a popular dialect of Lisp. Although we implement a subset of Scheme in Scheme, the interpreter is similar to the original self-definition given by McCarthy. The basic operations to decompose a list, **car** and **cdr**, and the list constructor **cons** are described in Figure 6.1. Combined with a predicate **null?** to test for an empty list, a conditional expression **cond**, and a method to define functions using

define, it is possible to write useful Scheme functions. See Appendix B for a more detailed description of Scheme. Our first example concatenates two lists. A function for concatenating lists is usually predefined in Lisp systems and goes by the name "append".

```
(define (concat lst1 lst2)
    (cond   ((null? lst1) lst2)
                    (#t (cons (car lst1) (concat (cdr lst1) lst2)))))
```

List Operations

(car <list>)	return the first item in <list>
(cdr <list>)	return <list> with the first item removed
(cons <item> <list>)	add <item> as first element of <list>

Arithmetic Operations

(+ <e_1> <e_2>)	return sum of the values of <e_1> and <e_2>
(- <e_1> <e_2>)	return difference of the values of <e_1> and <e_2>
(* <e_1><e_2>)	return product of the values of <e_1> and <e_2>
(/ <e_1> <e_2>)	return quotient of the values of <e_1> and <e_2>

Predicates

(null? <list>)	test if <list> is empty
(equal? <s_1> <s_2>)	test the equality of S-expressions <s_1> and <s_2>
(atom? <s>)	test if <s> is an atom

Conditional

(cond (<p_1> <e_1>)	sequentially evaluate predicates <p_1>, <p_2>, ... till
(<p_2> <e_2>)	one of them, say <p_i>, returns a not false (not #f)
: :	result; then the corresponding expression e_i is
(<p_n> <e_n>))	evaluated and its value returned from the cond

Function Definition and Anonymous Functions

(define (<name> <formals>) <body>)	allow user to define function <name> with formal parameters <formals> and function body <body>
(lambda (<formals>) <body>)	create an anonymous function
(let (<var-bindings>) <body>)	an alternative to function application; <var-bindings> is a list of (variable S-expression) pairs and the body is a list of S-expressions; let returns the value of last S-expression in <body>

Other

(quote <item>)	return <item> without evaluating it
(display <expr>)	print the value of <expr> and return that value
(newline)	print a carriage return and return ( )

Figure 6.1: Built-in Functions of Scheme

The symbols #t and #f represent the constant values true and false. Anonymous functions can be defined as **lambda** expressions. The **let** expression is a variant of function application. If we add an equality predicate **equal?** and an atom-testing predicate **atom?**, we can write other useful list processing functions with this small set of built-in functions. In the replace function below, all occurrences of the item s are replaced with the item r at the top level in the list lst.

```
(define (replace s r lst)
    (cond   ((null? lst) lst)
            ((equal? (car lst) s) (cons r (replace s r (cdr lst))))
            (#t (cons (car lst) (replace s r (cdr lst))))))
```

In order to test the metacircular interpreter, it is necessary to have a function **quote** that returns its argument unevaluated and a function **display** that prints the value of an S-expression. The basic built-in functions of Scheme are shown in Figure 6.1.

We have elected to expand the basic interpreter by adding four arithmetic operations, **+**, **-**, *****, and **/**, so that we can execute some recursive arithmetic functions that are familiar from imperative programming.

```
(define (fibonacci n)
    (cond   ((equal? n 0) 1)
            ((equal? n 1) 1)
            (#t (+ (fibonacci (- n 1)) (fibonacci (- n 2))))))

(define (factorial n)
    (cond   ((equal? n 0) 1)
            (#t (* n (factorial (- n 1))))))
```

Metacircular Interpreter

When designing a metacircular interpreter, it is easy to confuse those expressions belonging to the language being interpreted and those belonging to the language that is doing the interpreting, since both are Scheme. To reduce confusion, we use two different fonts: san-serif font for the interpreter code and normal serif font for the interpreted language. We need three major functions to construct the interpreter:

- The top-level function **micro-rep** **r**eads an S-expression (an atom or a list), **e**valuates it, and **p**rints the result, thus the name **rep**. The function micro-rep begins with an empty environment in which all identifiers are unbound.

- The function **micro-eval** accepts an S-expression and an environment and returns the value of the S-expression in the context of the environment.
- The function **micro-apply** accepts a function name or lambda abstraction, a list of the results from evaluating the actual parameters, and an environment, and returns the result of applying the given function to the parameters in the given environment.

The functions micro-eval and micro-apply are mutually recursive and continue simplifying the task at hand until they reach a base case that can be solved directly. An environment is an association list, a list of (name value) pairs. Function definitions and variable bindings share the same association list.

The recursive function micro-rep repeatedly reads an S-expression after printing a prompt, evaluates the expression, prints the result, and calls itself with the new environment reflecting any definitions that have been elaborated. The function micro-rep handles two situations:

- If the S-expression is the atom quit, micro-rep prints “Goodbye” and exits the interpreter.
- If the S-expression is a function definition, micro-rep uses the utility function updateEnv to add the function name to the environment with an associated value, which is a lambda expression encapsulating the parameter list and the body of the function. Then micro-rep displays the name of the function.

All other S-expressions are passed on to micro-eval for evaluation. Note that atoms are recognized first so that we only apply car to a list.

```
(define (micro-rep env)
    (let ((prompt (display ">> ")) (s (read)))
        (if  (equal? s 'quit)
             (begin (newline) (display "Goodbye") (newline))
             (cond
                  ((atom? s)  (begin   (newline)
                                       (display (micro-eval s env))
                                       (newline)
                                       (micro-rep env)))
                  ((equal? (car s) 'define)
                           (let ((newenv (updateEnv env
                                                (caadr s)
                                                (list 'lambda (cdadr s) (caddr s)))))
                                (begin   (newline)
                                         (display (caadr s))
                                         (newline)
                                         (micro-rep newenv))))
```

```
(#t  (begin  (newline)
             (display (micro-eval s env))
             (newline)
             (micro-rep env)))))))
```

The utility function updateEnv adds a new binding onto the front of the given environment.

```
(define (updateEnv env ide binding) (cons (list ide binding) env))
```

The function micro-eval deals with several forms of S-expressions as described below:

- An atom is either a constant (#t, #f, or a numeral) or a variable whose value is returned.
- A quoted expression is returned unevaluated.
- The function "display" evaluates its argument, displays that value, and returns the value of the expression printed.
- The function "newline" prints a carriage return.
- A conditional expression "cond" is handled separately since, unlike most other functions, its arguments are only evaluated on an "as needed" basis.
- A "let" expression augments the environment with the new variable bindings and evaluates the body of the let in this environment.

All other S-expressions are function calls to be processed by micro-apply, which receives three arguments:

- A function object, either an identifier bound to a function or a lambda expression.
- The actual parameters after their evaluation, accomplished by mapping micro-eval over the actual parameter list.
- The current environment.

We first present the main function micro-eval. The environment is an association list where the first item in each entry is an identifier. The utility function applyEnv uses the built-in function assoc to search for a given identifier in an association list and return the first list entry that matches the identifier.

```
(define (applyEnv ide env) (cadr (assoc ide env)))
```

Map, which is used to evaluate a list of actual parameters, is a built-in function that applies a functional argument to every item in a list and returns the list of results.

```
(define (micro-eval s env)
    (cond ((atom? s)
                (cond  ((equal? s #t) #t)
                       ((equal? s #f) #f)
                       ((number? s) s)
                       (else (applyEnv s env))))
          ((equal? (car s) 'quote) (cadr s))
          ((equal? (car s) 'lambda) s)
          ((equal? (car s) 'display)
                (let ((expr-value (micro-eval (cadr s) env)))
                     (display expr-value) expr-value))
          ((equal? (car s) 'newline) (begin (newline) '( )))
          ((equal? (car s) 'cond) (micro-evalcond (cdr s) env))
          ((equal? (car s) 'let)
                (micro-evallet (cddr s) (micro-let-bind (cadr s) env)))
          (else (micro-apply  (car s)
                              (map (lambda (x) (micro-eval x env )) (cdr s))
                              env))))
```

Observe that the value of a lambda expression in this implementation is the lambda expression itself. So lambda expressions are handled in the same way as boolean constants and numerals, and the internal representation of a function is identical to its syntactic representation.

The arguments for the cond function are a sequence of lists, each with two parts: a predicate to be tested and an expression to be evaluated if the result of the test is non-#f (not false). These lists are evaluated sequentially until the first non-#f predicate is encountered. If all predicates return #f, cond returns #f. The function micro-evalcond is used to perform the necessary evaluations on an "as needed" basis.

```
(define (micro-evalcond clauses env)
    (cond   ((null? clauses) #f)
            ((micro-eval (caar clauses) env) (micro-eval (cadar clauses) env))
            (else (micro-evalcond (cdr clauses) env))))
```

We show two simple uses of the let expression before discussing its implementation. The following let expression returns 5:

```
(let ((a 2) (b 3)) (+ a b))
```

In the case of nested let's, the nearest local binding is used.

```
(let ((a 5))  (display a)
        (let ((a 6)) (display a))
        (display (display a)))
```

prints (all on one line)

5 *from the first display*
6 *from the second display inside the inner let*
5 *from the final nested display*
5 *from the final outer display*
5 *from the outer let.*

Notice that the value returned from the inner let is not displayed since it is not the final S-expression in the outer let. The function micro-evallet receives a list of one or more S-expressions from the body of the let and the environment constructed using micro-let-bind applied to the list of bindings. These S-expressions are evaluated until the final one is reached, and that one is returned after being evaluated.

```
(define (micro-evallet exprlist env)
    (if   (null? (cdr exprlist))
          (micro-eval (car exprlist) env)
          (begin  (micro-eval (car exprlist) env)
                  (micro-evallet (cdr exprlist) env))))
```

The environment for the execution of a let is the current environment of the let augmented by the bindings created by the list of (identifier value) pairs.

```
(define (micro-let-bind pairlist env)
    (if   (null? pairlist)
          env
          (cons (list (caar pairlist) (micro-eval (cadar pairlist) env))
                                      (micro-let-bind (cdr pairlist) env))))
```

We now turn our attention to micro-apply. If the object passed to micro-apply is one of the predefined functions car, cdr, cons, atom?, null?, equal?, +, -, *, or /, the appropriate function is executed. If the object is a user-defined function, micro-apply is called recursively with the value (a lambda expression) associated with the function identifier, retrieved from the environment using micro-eval, as the first parameter. If fn, the first formal parameter of micro-apply, is not an atom, it must already be a lambda expression (an explicit check can be added if desired). Calling micro-apply with a lambda expression causes micro-eval to be called with the body of the function as the S-expression and the environment augmented by the binding of the formal parameters to the actual parameter values. This binding is accomplished by micro-bind, which accepts a list of formal parameters, a list of values, and the current environment and adds the (identifier value) pairs, one at a time, to the environment. Notice that the bindings are added to the front of the environ-

ment, which acts like a stack, so that the most recent value is always retrieved by applyEnv.

```
(define (micro-apply fn args env)
    (if  (atom? fn)
        (cond  ((equal? fn 'car) (caar args))
               ((equal? fn 'cdr) (cdar args))
               ((equal? fn 'cons) (cons (car args) (cadr args)))
               ((equal? fn 'atom?) (atom? (car args)))
               ((equal? fn 'null?) (null? (car args)))
               ((equal? fn 'equal?) (equal? (car args) (cadr args)))
               ((equal? fn '+) (+ (car args) (cadr args)))
               ((equal? fn '-) (- (car args) (cadr args)))
               ((equal? fn '*) (* (car args) (cadr args)))
               ((equal? fn '/) (/ (car args) (cadr args)))
               (else (micro-apply (micro-eval fn env) args env)))
        (micro-eval (caddr fn) (micro-bind (cadr fn) args env))))

(define (micro-bind key-list value-list env)
    (if  (or (null? key-list) (null? value-list))
        env
        (cons (list (car key-list) (car value-list))
                    (micro-bind (cdr key-list) (cdr value-list) env))))
```

This completes the code for our interpreter, which is initiated by entering

```
(micro-rep '( )).
```

Running the Interpreter

To illustrate its operation, we trace the interpretation of a simple user-defined function "first" that is a renaming of the built-in function car.

```
>>  (define (first lst) (car lst))
first
```

Now consider the execution of the function call:

```
>>  (first (quote (a b c)))
a
```

This S-expression is not dealt with by micro-eval, but is passed to micro-apply with three arguments:

first	a function identifier
((a b c))	evaluation of the actual parameters
((first (lambda (lst) (car lst))))	the current environment

The evaluation of the actual parameters results from mapping micro-eval onto the actual parameter list. In this case, the only actual parameter is an expression that calls the function quote, which is handled by micro-eval directly.

Since micro-apply does not recognize the object first, it appeals to micro-eval to evaluate first. So micro-eval looks up a value for first in the environment and returns a lambda expression to micro-apply, which then calls itself recursively with the following arguments:

((lambda (lst) (car lst)))	a function object
((a b c))	evaluation of the actual parameter
((first (lambda (lst) (car lst))))	the current environment

Since the object is not an atom, this results in a call to micro-eval, with the function body as the first parameter and the environment, augmented by the binding of formal parameters to actual values.

(car lst)	S-expression to be evaluated
((lst (a b c))	
(first (lambda (lst) (car lst))))	the current environment

But micro-eval does not deal with car directly; it now calls micro-apply with the first parameter as car, the evaluation of the actual parameters, and the environment.

car	a function identifier
((a b c))	evaluation of the actual parameters
((lst (a b c))	
(first (lambda (lst) (car lst))))	the current environment

The actual parameter value is supplied when micro-eval evaluates the actual parameter lst. The function car is something that micro-apply knows how to deal with directly; it returns the caar of the arguments, namely the atom a. This result is returned through all the function calls back to micro-rep, which displays the result.

This interpreter can handle simple recursion, as illustrated by the Fibonacci and factorial functions given earlier, and it can also handle nested recursion, as illustrated by Ackermann's function shown below. We illustrate by calling Ackermann's with values 3,2 and 3,3, but with no higher values due to the explosion of recursion calls.

```
>> (define (ackermann x y)
       (cond  ((equal? x 0) (+ y 1))
              ((equal? y 0) (ackermann (- x 1) 1))
              (#t (ackermann (- x 1)  (ackermann x (- y 1))))))
ackermann

>> (ackermann 3 2)
29

>> (ackermann 3 3)
61
```

The interpreter can also deal with anonymous lambda functions, as illustrated by

```
>> ((lambda (lst) (car (cdr lst))) (quote (1 2 3)))
2
```

A let expression can also bind identifiers to lambda expressions, as illustrated by:

```
>> (let ((addition (lambda (x y) (+ x y))) (a 2) (b 3)) (addition a b))
5
```

Let expressions that are nested use the innermost binding, including lambda functions.

```
>> (let ((addb 4) (b 2))
       (let ((addb (lambda (x) (+ b x))) (b 3)) (display (addb b))))
6   from the display itself
6   from the inner let passing its result back through the outer let.
```

These values are printed on the same line.

Because of the way our interpreter works, let can be recursive (a letrec in Scheme), as illustrated by the following example:

```
>> (let ((fact (quote (lambda (n)
                (cond  ((equal? n 0) 1)
                       (#t (* n (fact (- n 1))))))))))
       (fact 5))
120
```

We complete our discussion of the Scheme interpreter by examining two strategies for evaluating nonlocal variables. Most programming languages, including Scheme and Common Lisp, use static scoping—that is, nonlocal variables are evaluated in their lexical environment. However, our interpreter and early versions of Lisp use dynamic scoping for which nonlocal variables are evaluated in the environment of the caller. This scoping strategy results in the **funarg** (**fun**ction **arg**ument) problem, which is best illustrated by an example. We first define a function twice that has a function argument and a value. The function twice returns the application of the function argument to the value, and a second application of the function to the result.

```
>> (define (twice func val) (func (func val)))
twice
```

Suppose we define a function double that multiplies its argument times two.

```
>> (define (double n) (* n 2))
double
```

Now we call twice passing double as the function and three as the value:

```
>> (twice double 3)
12
```

The value returned is 12, as expected, since doubling 3 once gives 6, and doubling it again gives 12. We now generalize the double function by writing a function, called times, that multiplies its argument by a preset value, called val.

```
>> (define (times x)  (* x val))
```

If val is set to 2, we expect the times function to perform just like the double function. Consider the following let expression:

```
>> (let ((val 2)) (twice times 3))
27
```

Surprisingly, the value of 27 is returned rather than the value 12. To understand what is happening, we must carefully examine the environment at

each step of execution. At the time of the function call the environment has three bindings:

```
((times (lambda (x) (* x val)))
 (twice (lambda (func val) (func (func val))))
 (val 2))
```

The execution of twice adds its parameter bindings to the environment before executing the body of the function.

```
((val 3)
 (func times)
 (times (lambda (x) (* x val)))
 (twice (lambda (func val) (func (func val))))
 (val 2))
```

Now we see the source of difficulty; when we start executing the function body for times and it fetches a value for val, it fetches 3 instead of 2. So, times became a tripling function, and tripling 3 twice gives 27. Once the execution of the function is completed, all parameter bindings disappear from the environment.

Although dynamic scoping is easy to implement, unexpected results, as illustrated above, have led designers of modern programming languages to abandon this approach. The exercises suggest some modifications to the interpreter so that it implements static scoping.

Exercises

1. Given the following function definition

```
>> (define (even n)
       (cond   ((equal? n (* (/ n 2) 2)) #t)
               (#t #f))),
```

   trace the evaluation of the function call:

```
>> (even 3)
```

2. Add predicates to the interpreter for the five arithmetic relational operations: <, <=, =, >, and >=.

3. Add the functions “add1” and “sub1” to the interpreter.

4. Add the functions (actually special forms since they do not always evaluate all of their parameters) "if", "and", and "or" to the interpreter.

5. Modify the interpreter to save the environment of function definition at the time of a define. Make sure that this new interpreter solves the funarg problem and gives the expected results, as shown by the following sequence of S-expressions:

```
>> (define (twice func val) (func (func val)))
>> (define (times x)  (* x val))
>> (let ((val 2)) (twice times 3))                    ; returns 12
>> (let ((val 4)) (twice times 3))                    ; returns 48
```

6. Implement the predicate zero? and change cond so that an else clause is allowed. Test the resulting implementation by applying these functions:

```
>> (define (even n)
            (cond  ((zero? n) #t)
                   (else (odd (- n 1)))))
>> (define (odd n)
            (cond  ((zero? n) #f)
                   (else (even (- n 1)))))
```

7. The implementation of Scheme in this section allows the definition only of functions using a particular format. Augment the implementation so that the following definitions are also allowed:

```
>> (define n 55)
>> (define map (lambda (fn lst)
                  (cond ((null? lst) (quote ()))
                        (#t (cons (fn (car lst)) (map fn (cdr lst)))))))
```

6.2 SELF-DEFINITION OF PROLOG

We first build a very simple meta-interpreter in Prolog that handles only the conjunction of goals and the chaining goals. A goal succeeds for one of three reasons:

1. The goal is true.
2. The goal is a conjunction and both conjuncts are true.
3. The goal is the head of a clause whose body is true.

All other goals fail. A predefined Prolog predicate clause searches the user database for a clause whose head matches the first argument; the body of the clause is returned as the second argument.

```
prove(true).

prove((Goal1, Goal2)) :- prove(Goal1), prove(Goal2).

prove(Goal) :- clause(Goal, Body), prove(Body).

prove(Goal) :- fail.
```

We define a membership function, called memb so it will not conflict with any built-in membership operation.

```
memb(X,[X|Rest]).
memb(X,[Y|Rest]) :- memb(X,Rest).
```

Here is the result of the testing:

```
:- prove((memb(X,[a,b,c]),memb(X,[b,c,d]))).
X = b ;   % semicolon requests the next answer, if any
X = c ;
no
:- prove((memb(X,[a,b,c]),memb(X,[d,e,f]))).
no
:- prove(((memb(X,[a,b,c]),memb(X,[b,c,d])),memb(X,[c,d,e]))).
X = c ;
no
```

These results are correct, but they provide little insight into how they are obtained. We can overcome this problem by returning a "proof tree" for each clause that succeeds. The proof for true is simply true, the proof of a conjunction of goals is a conjunction of the individual proofs, and a proof of a clause whose head is true because the body is true will be represented as "Goal<==Proof". We introduce a new infix binary operator <== for this purpose. The proof tree for failure is simply fail.

```
:- op(500,xfy,<==).

prove(true, true).

prove((Goal1, Goal2),(Proof1, Proof2)) :-   prove(Goal1,Proof1),
                                            prove(Goal2,Proof2).

prove(Goal, Goal<==Proof) :- clause(Goal, Body), prove(Body, Proof).

prove(Goal,fail) :- fail.
```

Here are the results of our test cases:

```
:- prove((memb(X,[a,b,c]),memb(X,[b,c,d])),Proof).
X = b
Proof = memb(b,[a,b,c])<==memb(b,[b,c])<==true,
memb(b,[b,c,d])<==true

:- prove((memb(X,[a,b,c]),memb(X,[d,e,f])), Proof).
no

:- prove(((memb(X,[a,b,c]),memb(X,[b,c,d])),
                              memb(X,[c,d,e])), Proof).
X = c
Proof =
(memb(c,[a,b,c])<==memb(c,[b,c])<==memb(c,[c])<==true,
          memb(c,[b,c,d])<==memb(c,[c,d])<==true),
          memb(c,[c,d,e])<==true
```

Displaying Failure

We still have no display for the second test where the proof fails. Another alternative is to add a trace facility to show each step in a proof, whether it succeeds or fails. This capability can be added to the second version of the meta-interpreter, but for simplicity we return to the first version of the program and add a tracing facility. Every time we chain a rule from a goal to a body, we will indent the trace two spaces. Therefore we add an argument that provides the level of indentation. This argument is initialized to zero and is incremented by two every time we prove a clause from its body.

Before the application of a user-defined rule, we print "Call: " and the goal. If we exit from the body of the goal successfully, we print "Exit: " and the goal. If a subsequent goal fails, we have to backtrack and retry a goal that previously succeeded. We add a predicate retry that is true the first time it is called but prints "Retry: ", the goal, and fails on subsequent calls. When a goal fails, "Fail: " and the goal are printed. Here is the meta-interpreter implementing these changes.

```
prove(Goal) :-  prove(Goal, 0).

prove(true, _).

prove((Goal1, Goal2), Level) :- prove(Goal1, Level), prove(Goal2, Level).

prove(Goal, Level) :-  tab(Level), write('Call: '), write(Goal), nl,
                       clause(Goal, Body),
                       NewLevel is Level + 2,
                       prove(Body, NewLevel),
                       tab(Level), write('Exit: '), write(Goal), nl,
                       retry(Goal, Level).
```

```
prove(Goal, Level) :- tab(Level), write('Fail: '), write(Goal), nl, fail.
retry(Goal, Level) :- true ;
                      tab(Level), write('Retry: '), write(Goal), nl,
                      fail.
```

In the first test we call prove with the query

prove((memb(X,[a,b,c]),memb(X,[b,c,d]))).

X first binds to a, but this fails for the second list. The predicate memb is retried for the first list and X binds to b. This succeeds for the second list, so the binding of X to b succeeds for both clauses.

```
Call: memb(_483,[a,b,c])
Exit: memb(a,[a,b,c])
Call: memb(a,[b,c,d])
  Call: memb(a,[c,d])
    Call: memb(a,[d])
      Call: memb(a,[])
      Fail: memb(a,[])
    Fail: memb(a,[d])
  Fail: memb(a,[c,d])
Fail: memb(a,[b,c,d])
Retry: memb(a,[a,b,c])
  Call: memb(_483,[b,c])
  Exit: memb(b,[b,c])
Exit: memb(b,[a,b,c])
Call: memb(b,[b,c,d])
Exit: memb(b,[b,c,d])
```

Consider the second query, which has no solution.

prove((memb(X,[a,b,c]),memb(X,[d,e,f]))).

X binds to a, then b, then c, all of which fail to be found in the second list. When the program backtracks to find any other bindings for X in the first list, it fails and the entire proof thus fails.

```
Call: memb(_483,[a,b,c])
Exit: memb(a,[a,b,c])
Call: memb(a,[d,e,f])
  Call: memb(a,[e,f])
    Call: memb(a,[f])
      Call: memb(a,[])
      Fail: memb(a,[])
    Fail: memb(a,[f])
  Fail: memb(a,[e,f])
```

```
Fail: memb(a,[d,e,f])
Retry: memb(a,[a,b,c])
  Call: memb(_483,[b,c])
  Exit: memb(b,[b,c])
Exit: memb(b,[a,b,c])
Call: memb(b,[d,e,f])
  Call: memb(b,[e,f])
    Call: memb(b,[f])
      Call: memb(b,[])
      Fail: memb(b,[])
    Fail: memb(b,[f])
  Fail: memb(b,[e,f])
Fail: memb(b,[d,e,f])
Retry: memb(b,[a,b,c])
  Retry: memb(b,[b,c])
    Call: memb(_483,[c])
    Exit: memb(c,[c])
  Exit: memb(c,[b,c])
Exit: memb(c,[a,b,c])
Call: memb(c,[d,e,f])
  Call: memb(c,[e,f])
    Call: memb(c,[f])
      Call: memb(c,[])
      Fail: memb(c,[])
    Fail: memb(c,[f])
  Fail: memb(c,[e,f])
Fail: memb(c,[d,e,f])
Retry: memb(c,[a,b,c])
  Retry: memb(c,[b,c])
    Retry: memb(c,[c])
      Call: memb(_483,[])
      Fail: memb(_483,[])
    Fail: memb(_483,[c])
  Fail: memb(_483,[b,c])
Fail: memb(_483,[a,b,c])
```

The final query succeeds.

prove(((memb(X,[a,b,c]),memb(X,[b,c,d])),memb(X,[c,d,e]))).

X first binds to a, but this fails for the second list. Backtracking to the first list, X binds to b, which succeeds for the second list but fails for the third list. There are no other occurrences of b in the second list, so the program back-

tracks to the first list and binds X to c. This succeeds for the second and third lists.

```
Call: memb(_486,[a,b,c])
Exit: memb(a,[a,b,c])
Call: memb(a,[b,c,d])
  Call: memb(a,[c,d])
    Call: memb(a,[d])
      Call: memb(a,[])
      Fail: memb(a,[])
    Fail: memb(a,[d])
  Fail: memb(a,[c,d])
Fail: memb(a,[b,c,d])
Retry: memb(a,[a,b,c])
  Call: memb(_486,[b,c])
  Exit: memb(b,[b,c])
Exit: memb(b,[a,b,c])
Call: memb(b,[b,c,d])
Exit: memb(b,[b,c,d])
Call: memb(b,[c,d,e])
  Call: memb(b,[d,e])
    Call: memb(b,[e])
      Call: memb(b,[])
      Fail: memb(b,[])
    Fail: memb(b,[e])
  Fail: memb(b,[d,e])
Fail: memb(b,[c,d,e])
Retry: memb(b,[b,c,d])
  Call: memb(b,[c,d])
    Call: memb(b,[d])
      Call: memb(b,[])
      Fail: memb(b,[])
    Fail: memb(b,[d])
  Fail: memb(b,[c,d])
Fail: memb(b,[b,c,d])
Retry: memb(b,[a,b,c])
  Retry: memb(b,[b,c])
    Call: memb(_486,[c])
    Exit: memb(c,[c])
  Exit: memb(c,[b,c])
Exit: memb(c,[a,b,c])
Call: memb(c,[b,c,d])
  Call: memb(c,[c,d])
  Exit: memb(c,[c,d])
Exit: memb(c,[b,c,d])
```

```
Call: memb(c,[c,d,e])
Exit: memb(c,[c,d,e])
```

Other improvements can be made to the interpreter, but these are left as exercises. The interpreter works only with user-defined clauses. This limitation is fairly easy to overcome by adding *call* for built-in clauses. There is no provision for disjunction. Perhaps the most difficult problem to handle is the addition of the cut clause to control the underlying search mechanism. Peter Ross discusses some alternatives that can handle the cut correctly. (See the further readings at the end of this chapter).

Our Prolog interpreter written in Prolog does not explicitly implement the built-in backtracking of Prolog or show the unification process. The trace facility allows us to follow the backtracking but does not illustrate its implementation. It is also possible to develop a simple Prolog interpreter written in Lisp where both the backtracking and unification are explicit. The interested reader may consult the references at the end of the chapter.

Exercises

1. Add a rule for prove that handles built-in predicates correctly by using call. Be careful to ensure that user-defined clauses are not called twice.

2. Add the capability to handle the disjunction of clauses correctly.

3. Research the implementation of the cut clause (see the references). Implement cut in the meta-interpreter.

4. Investigate the implementation of a unification function in Lisp or Scheme. Write and test your function.

5. Use your unification function from exercise 4 to build a logic interpreter in Lisp or Scheme.

6.3 FURTHER READING

The self-definition of a programming language is a special case of a more general technique: using a high-level programming language as a metalanguage for defining the semantics of a high-level programming language. The use of programming languages as metalanguages is discussed in [Pagan76] and [Anderson76]. [Pagan81] gives a definition of the minilanguage Pam using Algol68 as a metalanguage.

Lisp was developed during the late 1950s; the seminal publication was [McCarthy60]. Our self-definition of Scheme using Scheme, a variant of Lisp, is similar to the original presentation of a Lisp interpreter written in Lisp given in [McCarthy65b]. Other versions appear in many textbooks on programming languages. The use of Scheme as a metalanguage to define a logic interpreter is described in [Abelson85]. Other good references for the Scheme programing language include [Springer89] and [Dybvig87].

We present Prolog as a metalanguage throughout this text. A variety of issues dealing with the implementation of Prolog are discussed in [Campbell84]. [Nillson84] presents a very concise interpreter for Prolog written in Lisp. Our treatment of a Prolog interpreter written in Prolog closely follows the more detailed presentation by [Ross89].