1
The Basics of Interpretation

THIS chapter introduces a basic interpreter that will serve as the foundation for most of this book. Deliberately simple, it's more closely related to Scheme than to Lisp, so we'll be able to explain Lisp in terms of Scheme that way. In this preliminary chapter, we'll broach a number of topics in succession: the articulations of this interpreter; the well known pair of functions, `eval` and `apply`; the qualities expected in environments and in functions. In short, we'll start various explorations here to pursue in later chapters, hoping that the intrepid reader will not be frightened away by the gaping abyss on either side of the trail.

The interpreter and its variations are written in native Scheme without any particular linguistic restrictions.

Literature about Lisp rarely resists that narcissistic pleasure of describing Lisp in Lisp. This habit began with the first reference manual for Lisp 1.5 [MAE+62] and has been widely imitated ever since. We'll mention only the following examples of that practice: (There are many others.) [Rib69], [Gre77], [Que82], [Cay83], [Cha80], [SJ93], [Rey72], [Gor75], [SS75], [All78], [McC78b], [Lak80], [Hen80], [BM82], [Cli84], [FW84], [dRS84], [AS85], [R3R86], [Mas86], [Dyb87], [WH88], [Kes88], [LF88], [Dil88], [Kam90].

Those evaluators are quite varied, both in the languages that they define and in what they use to do so, but most of all in the goals they pursue. The evaluator defined in [Lak80], for example, shows how graphic objects and concepts can emerge naturally from Lisp, while the evaluator in [BM82] focuses on the size of evaluation.

The language used for the *definition* is important as well. If assignment and surgical tools (such as `set-car!`, `set-cdr!`, and so forth) are allowed in the definition language, they enrich it and thus minimize the size (in number of lines) of descriptions; indeed, with them, we can precisely simulate the language *being defined* in terms that remind us of the lowest level machine instructions. Conversely, the description uses more concepts. Restricting the definition language in that way complicates our task, but lowers the risk of semantic divergence. Even if the size of the description grows, the language being defined will be more precise and, to that degree, better understood.

Figure 1.1 shows a few representative interpreters in terms of the complexity

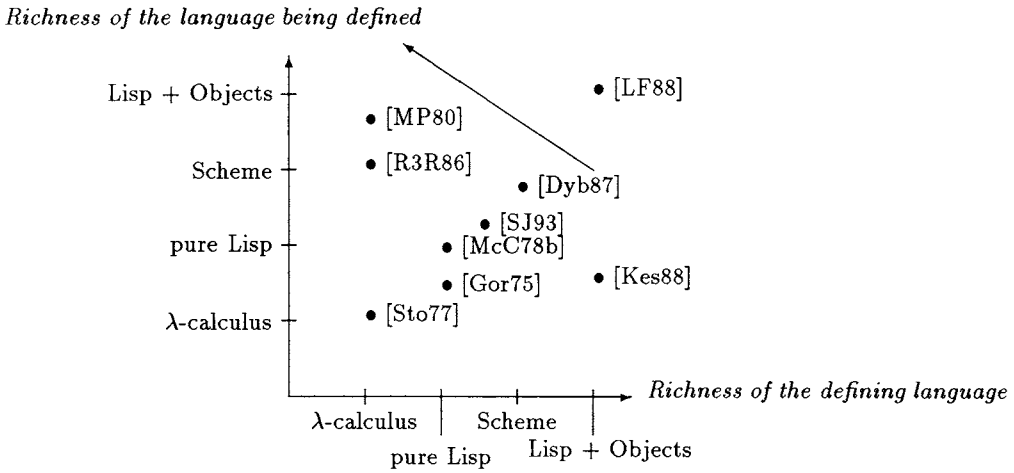

Figure 1.1

of their definition language (along the x-axis) and the complexity of the language being defined (along the y-axis). The knowledge progression shows up very well in that graph: more and more complicated problems are attacked by more and more restricted means. This book corresponds to the vector taking off from a very rich version of Lisp to implement Scheme in order to arrive at the λ-calculus implementing a very rich Lisp.

1.1 Evaluation

The most essential part of a Lisp interpreter is concentrated in a single function around which more useful functions are organized. That function, known as **`eval`**, takes a program as its argument and returns the value as output. The presence of an explicit evaluator is a characteristic trait of Lisp, not an accident, but actually the result of deliberate design.

We say that a language is *universal* if it is as powerful as a Turing machine. Since a Turing machine is fairly rudimentary (with its mobile read-write head and its memory composed of binary cells), it's not too difficult to design a language that powerful; indeed, it's probably more difficult to design a useful language that is not universal.

Church's thesis says that any function that can be computed can be written in any universal language. A Lisp system can be compared to a function taking programs as input and returning their value as output. The very existence of such systems proves that they are computable: thus a Lisp system can be written in a universal language. Consequently, the function **`eval`** itself can be written in Lisp, and more generally, the behavior of Fortran can be described in Fortran, and so forth.

What makes Lisp unique—and thus what makes an explication of **`eval`** non-trivial—is its reasonable size, normally from one to twenty pages, depending on the

level of detail.[1] This property is the result of a significant effort in design to make the language more regular, to suppress special cases, and above all to establish a syntax that's both simple and abstract.

Many interesting properties result from the existence of `eval` and from the fact that it can be defined in Lisp itself.

- You can learn Lisp by reading a reference manual (one that explains functions thematically) or by studying the `eval` function itself. The difficulty with that second approach is that you have to know Lisp in order to read the definition of `eval`—though knowing Lisp, of course, is the *result* we're hoping for, rather than the *prerequisite*. In fact, it's sufficient for you to know only the subset of Lisp used by `eval`. The language that defines `eval` is a pared-down one in the sense that it procures only the essence of the language, reduced to special forms and primitive functions.

  It's an undeniable advantage of Lisp that it brings you these two intertwined approaches for learning it.

- The fact that the definition of `eval` is available in Lisp means that the programming environment is part of the language, too, and costs little. By programming environment, we mean such things as a tracer, a debugger, or even a reversible evaluator [Lie87]. In practice, writing these tools to control evaluation is just a matter of elaborating the code for `eval`, for example, to print function calls, to store intermediate results, to ask the end-user whether he or she wants to go on with the evaluation, and so forth.

  For a long time, these qualities have insured that Lisp offers a superior programming environment. Even today, the fact that `eval` can be defined in Lisp means that it's easy to experiment with new models of implementation and debugging.

- Finally, `eval` itself is a programming tool. This tool is controversial since it implies that an application written in Lisp and using `eval` must include an entire interpreter or compiler, but more seriously, it must give up the possibility of many optimizations. In other words, using `eval` is not without consequences. In certain cases, its use is justified, notably when Lisp serves as the definition and the implementation of an incremental programming language.

  Even apart from that important cost, the semantics of `eval` is not clear, a fact that justifies its being separated from the official definition of Scheme in [CR91b]. [see p. 271]

1.2 Basic Evaluator

Within a program, we distinguish *free variables* from *bound variables*. A variable is free as long as no binding form (such as `lambda`, `let`, and so forth) qualifies it; otherwise, we say that a variable is bound. As the term indicates, a free variable is unbound by any constraint; its value could be anything. Consequently, in order

1. In this chapter, we define a Lisp of about 150 lines.

to know the value of a fragment of a program containing free variables, we must know the values of those free variables themselves. The data structure associating variables and values is known as an *environment*. The function **evaluate**[2] is thus binary; it takes a program accompanied by an environment and returns a value.

```
(define (evaluate exp env) ...)
```

1.3 Evaluating Atoms

An important characteristic of Lisp is that programs are represented by expressions of the language. However, since any representation assumes a degree of encoding, we have to explain more about how programs are represented. The principal conventions of representation are that a variable is represented by a symbol (its name) and that a functional application is represented by a list where the first term of the list represents the function to apply and the other terms represent arguments submitted to that function.

Like any other compiler, **evaluate** begins its work by syntactically analyzing the expression to evaluate in order to deduce what it represents. In that sense, the title of this section is inappropriate since this section does not literally involve evaluating atoms but rather evaluating programs where the representation is atomic. It's important, in this context, to distinguish the program from its representation (or, the message from its medium, if you will). The function **evaluate** works on the representation; from the representation, it deduces the expected intention; finally, it executes what's requested.

```
(define (evaluate exp env)
  (if (atom? exp)  ; (atom?  exp) ≡ (not (pair?  exp))
      ...
      (case (car exp)
        ...
        (else ...) ) ) )
```

If an expression is not a list, perhaps it's a symbol or actual data, such as a number or a character string. When the expression is a symbol, the expression represents a *variable* and its value is the one attributed by the environment.

```
(define (evaluate exp env)
  (if (atom? exp)
      (if (symbol? exp) (lookup exp env) exp)
      (case (car exp)
        ...
        (else ...) ) ) )
```

The function **lookup** (which we'll explain later on page 13) knows how to find the value of a variable in an environment. Here's the signature of **lookup**:

(**lookup** *variable environment*) → *value*

2. There is a possibility of confusion here. We've already mentioned the evaluator defined by the function **eval**; it's widely present in any implementation of Scheme, even if not standardized; it's often unary—accepting only one argument. To avoid confusion, we'll call the function **eval** that we are defining by the name **evaluate** and the associated function **apply** by the name **invoke**. These new names will also make your life easier if you want to experiment with these programs.

In consequence, an implicit conversion takes place between a symbol and a variable. If we were more meticulous about how we write, then in place of `(lookup exp env)`, we should have written:

```
... (lookup (symbol->variable exp) env) ...
```

That more scrupulous way of writing emphasizes that the symbol—the value of `exp`—must be changed into a variable. It also underscores the fact that the function `symbol->variable`[3] is not at all an identity; rather, it converts a syntactic entity (the symbol) into a semantic entity (the variable). In practice, then, a variable is nothing other than an imaginary object to which the language and the programmer attach a certain sense but which, for practical reasons, is handled only by means of its representation. The representation was chosen for its convenience: `symbol->variable` works like the identity because Lisp exploits the idea of a symbol as one of its basic types. In fact, other representations could have been adopted; for example, a variable could have appeared in the form of a group of characters, prefixed by a dollar sign. In that case, the conversion function `symbol->variable` would have been less simple.

If a variable were an imaginary concept, the function `lookup` would not know how to accept it as a first argument, since `lookup` knows how to work only on tangible objects. For that reason, once again, we have to encode the variable in a representation, this time, a key, to enable `lookup` to find its value in the environment. A precise way of writing it would thus be:

```
... (lookup (variable->key (symbol->variable exp)) env) ...
```

But the natural laziness of Lisp-users inclines them to use the symbol of the same name as the key associated with a variable. In that context, then, `variable->key` is merely the inverse of `symbol->variable` and the composition of those two functions is simply the identity.

When an expression is atomic (that is, when it does not involve a dotted pair) and when that expression is not a symbol, we have the habit of considering it as the representation of a constant that is its own value. This idempotence is known as the *autoquote* facility. An autoquoted object does not need to be quoted, and it is its own value. See [Cha94] for an example.

Here again, this choice is not obvious for several reasons. Not all atomic objects naturally denote themselves. The value of the character string `"a?b:c"` might be to call the C compiler for this string, then to execute the resulting program, and to insert the results back in Lisp.

Other types of objects (functions, for example) seem stubbornly resistant to the idea of evaluation. Consider the variable `car`, one that we all know the utility of; its value is the function that extracts the left child from a pair; but that function *car* itself—what is its value? Evaluating a function usually proves to be an error that should have been detected and prevented earlier.

Another example of a problematic value is the empty list `()`. From the way it is written, it might suggest that it is an empty application, that is, a functional application without any arguments where we've even forgotten to mention the

3. Personally, I don't like names formed like this x`->`y to indicate a conversion because this order makes it difficult to understand compositions; for example, (`y->z(x->y...)`) is less straightforward than (`z<-y(y<-x...)`). In contrast, x`->`y is much easier to read than y`<-`x. You can see here one of the many difficulties that language designers come up against.

function. That syntax is forbidden in Scheme and consequently is not defined as having a value.

For those kinds of reasons, we have to analyze expressions very carefully, and we should *autoquote* only those data that deserve it, namely, numbers, characters, and strings of characters. **[see p. 7]** We could thus write:

```
(define (evaluate e env)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e)(string? e)(char? e)(boolean? e)(vector? e))
             e )
            (else (wrong "Cannot evaluate" e)) )
      ... ) )
```

In that fragment of code, you can see the first case of possible errors. Most Lisp systems have their own exception mechanism; it, too, is difficult to write in portable code. In an error situation, we could call `wrong`[4] with a character string as its first argument. That character string would describe the kind of error, and the following arguments could be the objects explaining the reason for the anomaly. We should mention, however, that more rudimentary systems send out cryptic messages, like `Bus error: core dump` when errors occur. Others stop the current computation and return to the basic interaction loop. Still others associate an exception handler with a computation, and that exception handler catches the object representing the error or exception and decides how to behave from there. **[see p. 255]** Some systems even offer exception handlers that are quasi-expert systems themselves, analyzing the error and the corresponding code to offer the end-user choices about appropriate corrections. In short, there is wide variation in this area.

1.4 Evaluating Forms

Every language has a number of syntactic forms that are "untouchable": they cannot be redefined adequately, and they must not be tampered with. In Lisp, such a form is known as a *special form*. It is represented by a list where the first term is a particular symbol belonging to the set of *special operators*.[5]

A dialect of Lisp is characterized by its set of special forms and by its library of primitive functions (those functions that cannot be written in the language itself and that have profound semantic repercussions, as, for example, `call/cc` in Scheme).

In some respects, Lisp is simply an ordinary version of applied λ-calculus, augmented by a set of special forms. However, the special genius of a given Lisp is expressed in just this set. Scheme has chosen to minimize the number of special operators (`quote`, `if`, `set!`, and `lambda`). In contrast, COMMON LISP (CLtL2

4. Notice that we did not say "the function `wrong`." We'll see more about error recovery on page 255.

5. We will follow the usual lax habit of considering a special operator, say `if`, as being a "special form" although `if` is not even a form. Scheme treats them as "syntactic keywords" whereas COMMON LISP recognizes them with the `special-form-p` predicate.

[Ste90]) has more than thirty or so, thus circumscribing the number of cases where it's possible to generate highly efficient code.

Because special forms are coded as they are, their syntactic analysis is simple: it is based on the first term of each such form, so one **case** statement suffices. When a special form does not begin with a keyword, we say that is it is a *functional application* or more simply an application. For the moment, we're looking at only a small subset of the general special forms: **quote**, **if**, **begin**, **set!**, and **lambda**. (Later chapters introduce new, more specialized forms.)

```
(define (evaluate e env)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e)(string? e)(char? e)(boolean? e)(vector? e))
             e )
            (else (wrong "Cannot evaluate" e)) )
      (case (car e)
        ((quote)  (cadr e))
        ((if)     (if (evaluate (cadr e) env)
                      (evaluate (caddr e) env)
                      (evaluate (cadddr e) env) ))
        ((begin)  (eprogn (cdr e) env))
        ((set!)   (update! (cadr e) env (evaluate (caddr e) env)))
        ((lambda) (make-function (cadr e) (cddr e) env))
        (else     (invoke (evaluate (car e) env)
                          (evlis (cdr e) env) )) ) ) )
```

In order to lighten that definition, we've reduced the syntactic analysis to its minimum, and we haven't bothered to verify whether the quotations are well formed, whether **if** is really ternary[6] (accepting three parameters) each time, and so forth. We'll assume that the programs that we're analyzing are syntactically correct.

1.4.1 Quoting

The special form **quote** makes it possible to introduce a value that, without its quotation, would have been confused with a legal expression. The decision to represent a program as a value of the language makes it necessary, when we want to speak about a particular value, to find a means for discriminating between data and programs that usurp the same space. A different syntactic choice could have avoided that problem. For example, M-expressions, originally planned in [McC60] as the normal syntax for programs written in Lisp, would have eliminated this particular problem, but they would have forbidden macros—a marvelously useful tool for extending syntax; however, M-expressions disappeared rapidly [McC78a]. The special form **quote** is consequently the principal discriminator between program and data.

6. As a form, **if** is not necessarily ternary. That is, it does not have to have three parameters. Scheme and COMMON LISP, for example, support both binary and ternary **if**, whereas EULISP and IS-Lisp accept only ternary **if**; Le-Lisp supports at least binary **if** with a **progn** implicit in the alternative. (**progn** corresponds to Scheme **begin**.)

Quotation consists of returning, as a value, that term following the keyword. That practice is clearly articulated in this fragment of code:

```
... (case (car e)
     ((quote) (cadr e)) ...) ...
```

You might well ask whether there is a difference between implicit and explicit quotation, for example, between `33` and `'33` or even between `#(fa do sol)` and `'#(fa do sol)`.[7] The first comparison—between `33` and `'33`—impinges on immediate objects although the second one—between `#(fa do sol)` and `'#(fa do sol)`—impinges on composite objects (though they are atomic objects in Lisp terminology). It is possible to imagine divergent meanings for those two fragments. Explicit quotation simply returns its quotation as its value whereas `#(fa do sol)` could return a new instance of the vector for every evaluation—a new instance of a vector of three components initialized by three particular symbols. In other words, `#(fa do sol)` may be nothing other than the abbreviation of `(vector 'fa 'do 'sol)` (that's one of the possibilities in Scheme, though not the right one), and its behavior is quite different from `'#(fa do sol)` and from `(vector fa do sol)`, for that matter. We'll be returning later [see p. 140] to the question of what meaning to give to quotation, since, as you can see, the subject is far from simple.

1.4.2 Alternatives

As we look at *alternatives*, we'll consider the special form `if` as ternary, a control structure that evaluates its first argument (the *condition*), then according to the value it gets from that evaluation, chooses to return the value of its second argument (the *consequence*) or its third argument (the *alternate*). That idea is expressed in this fragment of code:

```
... (case (car e) ...
     ((if) (if (evaluate (cadr e) env)
               (evaluate (caddr e) env)
               (evaluate (cadddr e) env) )) ... ) ...
```

This program does not do full justice to the representation of Booleans. As you have no doubt noticed, we're mixing two languages here: the first is Scheme (or at least, a close enough approximation that it's indistinguishable from Scheme) whereas the second is also Scheme (or at least something quite close). The first language implements the second. As a consequence, there is the same relation between them as, for example, between Pascal (the language of the first implementation of TEX) and TEX itself [Knu84]. Consequently, there is no reason to identify the representation of Booleans of these two languages.

The function `evaluate` returns a value belonging to the language that is being defined. That value maintains no *a priori* relation to the Boolean values of the defining language. Since we follow the convention that any object different from the Boolean *false* must be considered as Boolean *true*, a more carefully written program would look like this:

```
... (case (car e) ...
```

7. In Scheme, the notation `#( ... )` represents a quoted vector.

```
((if) (if (not (eq? (evaluate (cadr e) env) the-false-value))
          (evaluate (caddr e) env)
          (evaluate (cadddr e) env) )) ... ) ...
```

Of course, we assume that the variable **the-false-value** has as its value the representation in the defining language of the Boolean *false* in the language being defined. There's a wide choice available to us for this value; for example,

```
(define the-false-value (cons "false" "boolean"))
```

The comparison of any value to the Boolean *false* is carried out by the physical comparer **eq?**, so a dotted pair handles the issue very well and can't be confused with any other possible value in the language being defined.

This discussion is not really trivial. In fact, Lisp chronicles are full of disputes about the differences between the Boolean value *false*, the empty list `()`, and the symbol **NIL**. The cleanest position to take in this controversy, quite independent of whether or not to preserve existing practice, is that *false* is different from `()`—which is, after all, simply an empty list—and that those two have nothing to do with the symbol spelled **N**, **I**, **L**.

That is, in fact, the position that Scheme takes; the position was adopted several weeks before being standardized by IEEE (the Institute of Electrical and Electronic Engineers) [IEE91].

Where things get worse is that `()` is pronounced *nil*! In "traditional" Lisp, *false*, `()`, and **NIL** are all one and same symbol. In Le-Lisp, **NIL** is a variable, its value is `()`, and the empty list has been assimilated with the Boolean *false* and with the empty symbol `||`.

1.4.3 Sequence

A *sequence* makes it possible to use a single syntactic form for a group of forms to evaluate sequentially. Like the well known **begin** ... **end** of the family of languages related to Algol, Scheme prefixes this special form by **begin** whereas other Lisps use **progn**, a generalization of **prog1**, **prog2**, etc. The sequential evaluation of a group of forms is "subcontracted" to a particular function: **eprogn**.

```
... (case (car e) ...
     ((begin) (eprogn (cdr e) env)) ... ) ...

(define (eprogn exps env)
  (if (pair? exps)
      (if (pair? (cdr exps))
          (begin (evaluate (car exps) env)
                 (eprogn (cdr exps) env) )
          (evaluate (car exps) env) )
      '() ) )
```

With that definition, the meaning of the sequence is now canonical. Nevertheless, we should note that in the middle of the definition of **eprogn**, there is a tail recursive call to **evaluate** to handle the final term of the sequence. That computation of the final term of a sequence is carried out as though it, and it alone, replaced the entire sequence. (We'll talk more about tail recursion later. [see p. 104])

We should also note the meaning we have to attribute to `(begin)`. Here, we've defined `(begin)` to return the empty list `()`, but why should we choose the empty list? Why not something else, anything else, like `bruce` or `brian`[8]? This choice is part of our heritage from Lisp, where the prevailing custom returns `nil` when nothing better seems to be obligatory. However, in a world where *false*, `()`, and `nil` are distinct, what should we return? We're going to specialize our language as it's defined in this chapter so that `(begin)` returns the value `empty-begin`, which will be the (almost) arbitrary number `813`[9] [Leb05].

```
(define (eprogn exps env)
  (if (pair? exps)
      (if (pair? (cdr exps))
          (begin (evaluate (car exps) env)
                 (eprogn (cdr exps) env) )
          (evaluate (car exps) env) )
      empty-begin ) )
(define empty-begin 813)
```

Our problem comes from the fact that the implementation that we are defining must necessarily return a value. Like Scheme, the language could attribute no particular meaning to `(begin)`; that choice could be interpreted in at least two different ways: either this way of writing is permitted within a particular implementation, in which case it is an extension that must return a value freely chosen by the implementation in question; or this way of writing is not allowed and thus an error. In light of the consequences, it's better to avoid using this form when no guarantee exists about its value. Some implementations have an object, `#<unspecified>`, that lends itself to this use, as well as more generally to any situation where we don't know what we should return because nothing seems appropriate. That object is usually printable; it should not be confused with the undefined pseudo-value. [see p. 60]

Sequences are of no particular interest if a language is purely functional (that is, if it has no side effects). What is the point of evaluating a program if we don't care about the results? Well, in fact there are situations in which we use a computation simply for its side effects. Consider, for example, a video game programmed in a purely functional language; computations take time, whether we are interested in their results or not; it may be just this very side effect—slowing things down—rather than the result that interests us. Provided that the compiler is not smart enough to notice and remove any useless computation, we can use such side effects to slow the program down sufficiently to accomodate a player's reflexes, say.

In the presence of conventional read or write operations, which have side effects on data flow, sequencing becomes very interesting because it is obviously clearer to pose a question (by means of `display`), then wait for the response (by means of `read`) than to do the reverse. Sequencing is, in that sense, the explicit form for putting a series of evaluations in order. Other special forms could also introduce a certain kind of order. For example, alternatives could do so, like this:

`(if` α β β`)` $\equiv$ `(begin` α β`)`

8. For the Monty Python fans in the audience
9. Fans of the gentleman thief Arsene Lupin will recognize the appropriateness of this choice.