11 S-expressions, the Syntax of Lisp

Chapter Objectives

The Lisp, s-expression introduced
    Basic syntactic unit for the language
Structures defined recursively
The list as data or function
    `quote`
    `eval`
Creating new functions in Lisp:
    `defun`
Control structures in Lisp
    Functions
        `cond`
        `if`
    Predicates
        `and`
        `or`
        `not`

Chapter Contents

11.1 Introduction to Symbol Expressions

The S-expression

The syntactic elements of the Lisp programming language are *symbolic expressions*, also known as *s-expressions*. Both programs and data are represented as s-expressions: an s-expression may be either an *atom* or a *list*. Lisp atoms are the basic syntactic units of the language and include both numbers and symbols. Symbolic atoms are composed of letters, numbers, and the non-alphanumeric characters.

Examples of Lisp atoms include:

```
3.1416
100
hyphenated-name
*some-global*
nil
```

A *list* is a sequence of either atoms or other lists separated by blanks and enclosed in parentheses. Examples of lists include:

```
(1 2 3 4)
(george kate james joyce)
(a (b c) (d (e f)))
()
```

Note that lists may be elements of lists. This nesting may be arbitrarily deep and allows us to create symbol structures of any desired form and complexity. The empty list, "`()`", plays a special role in the construction and manipulation of Lisp data structures and is given the special name `nil`. `nil` is the only s-expression that is considered to be both an atom and a list. Lists are extremely flexible tools for constructing representational structures. For example, we can use lists to represent expressions in the predicate calculus:

```
(on block-1 table)
(likes bill X)
(and (likes george kate) (likes bill merry))
```

We use this syntax to represent predicate calculus expressions in the unification algorithm of this chapter. The next two examples suggest ways in which lists may be used to implement the data structures needed in a database application.

```
((2467 (lovelace ada) programmer)
 (3592 (babbage charles) computer-designer))
((key-1 value-1) (key-2 value-2) (key-3 value-3))
```

An important feature of Lisp is its use of Lisp syntax to represent programs as well as data. For example, the lists,

```
(* 7 9)
(– (+ 3 4) 7)
```

may be interpreted as arithmetic expressions in a prefix notation. This is exactly how Lisp treats these expressions, with `(* 7 9)` representing the product of 7 and 9. When Lisp is invoked, the user enters an interactive dialogue with the Lisp interpreter. The interpreter prints a prompt, in our examples ">", reads the user input, attempts to evaluate that input, and, if successful, prints the result. For example:

```
> (* 7 9)
63
>
```

Here, the user enters `(* 7 9)` and the Lisp interpreter responds with 63, i.e., the *value* associated with that expression. Lisp then prints another prompt and waits for more user input. This cycle is known as the *read-eval-print* loop and is the heart of the Lisp interpreter.

When given a list, the Lisp evaluator attempts to interpret the first element of the list as the name of a function and the remaining elements as its arguments. Thus, the s-expression `(f x y)` is equivalent to the more traditional looking mathematical function notation `f(x,y)`. The value printed by Lisp is the result of *applying* the function to its arguments. Lisp expressions that may be meaningfully evaluated are called *forms*. If the user enters an expression that may not be correctly evaluated, Lisp prints an error message and allows the user to trace and correct the problem. A sample Lisp session appears below:

```
> (+ 14 5)
19
> (+ 1 2 3 4)
10
> (– (+ 3 4) 7)
0
```

```
> (* (+ 2 5) (– 7 (/ 21 7)))
28
> (= (+ 2 3) 5)
t
> (> (* 5 6) (+ 4 5))
t
> (a b c)
Error: invalid function: a
```

Several of the examples above have arguments that are themselves lists, for example the expression `(– (+ 3 4) 7)`. This indicates the composition of functions, in this case "subtract 7 from the *result* of adding 3 to 4". The word "result" is emphasized here to indicate that the function—is not passed the s-expression "`(+ 3 4)`" as an argument but rather the result of *evaluating* that expression.

In evaluating a function, Lisp first evaluates its arguments and then applies the function indicated by the first element of the expression to the results of these evaluations. If the arguments are themselves function expressions, Lisp applies this rule recursively to their evaluation. Thus, Lisp allows nested function calls of arbitrary depth. It is important to remember that, by default, Lisp evaluates everything. Lisp uses the convention that numbers always evaluate to themselves. If, for example, `5` is typed into the Lisp interpreter, Lisp will respond with `5`. Symbols, such as `x`, may have a value *bound* to them. If a symbol is bound, the binding is returned when the symbol is evaluated (one way in which symbols become bound is in a function call; see Section 13.2). If a symbol is unbound, it is an error to evaluate that symbol.

For example, in evaluating the expression `(+ (* 2 3) (* 3 5`)), Lisp first evaluates the arguments, `(* 2 3)` and `(* 3 5)`. In evaluating `(* 2 3)`, Lisp evaluates the arguments `2` and `3`, which return their respective arithmetic values; these values are multiplied to yield `6`. Similarly, `(* 3 5)` evaluates to `15`. These results are then passed to the top-level addition, which is evaluated, returning `21`. A diagram of this evaluation appears in Figure 11.1.

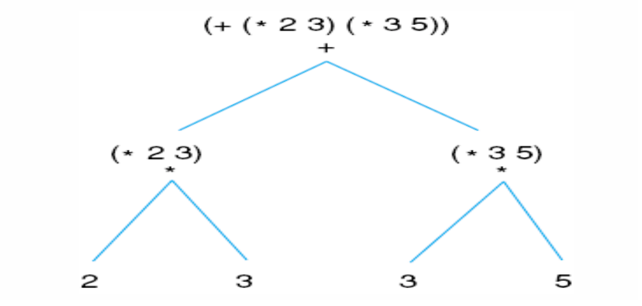

Figure 11.1. Tree representation of the evaluation of a simple Lisp function

In addition to arithmetic operations, Lisp includes a large number of functions that operate on lists. These include functions to construct and combine lists, to access elements of lists, and to test various properties. For example, `list` takes any number of arguments and constructs a list of those elements. `nth` takes a number and a list as arguments and returns the indicated element of the list. By convention, `nth` begins counting with `0`. Examples of these and other list manipulation functions include:

```
> (list 1 2 3 4 5)
(1 2 3 4 5)
> (nth 0 '(a b c d))
a
> (nth 2 (list 1 2 3 4 5))
3
> (nth 2 '((a 1) (b 2) (c 3) (d 4)))
(c 3)
> (length '(a b c d))
4
> (member 7 '(1 2 3 4 5))
nil
> (null ( ))
t
```

S-expressions Defined

DEFINITION

S-EXPRESSION

An *s-expression* is defined recursively:

An *atom* is an s-expression.

If s_1, s_2, …, s_n are s-expressions, then so is the list $(s_1\ s_2\ \ldots\ s_n)$.

A *list* is a non-atomic s-expression.

A form is an s-expression that is intended to be evaluated. If it is a list, the first element is treated as the function name and the subsequent elements are evaluated to obtain the function arguments.

In evaluating an s-expression:

If the s-expression is a number, return the value of the number.

If the s-expression is an atomic symbol, return the value bound to that symbol; if it is not bound, it is an error.

If the s-expression is a list, evaluate the second through the last arguments and apply the function indicated by the first argument to the results.

Lisp represents both programs and data as s-expressions. Not only does this simplify the syntax of the language but also, when combined with the ability to control the evaluation of s-expressions, it makes it easy to write programs that treat other Lisp programs as data. This simplifies the implementation of interpreters in Lisp.

11.2 Control of Lisp Evaluation

Using `quote` and `eval`

In the previous section, several of the examples included list arguments preceded by a single quotation mark: `'`. The `'`, which can also be represented by the function `quote`, is a special function which does not evaluate its argument but prevents evaluation, often because its argument is to be treated as data rather than as an evaluable form.

When evaluating an s-expression, Lisp will first try to evaluate all of its arguments. If the interpreter is given the expression `(nth 0 (a b c d))`, it will first try to evaluate the argument `(a b c d)`. This attempted evaluation will result in an error, because `a`, the first element of this s-expression, does not represent any known Lisp function. To prevent this, Lisp provides the user with the built-in function `quote`. `quote` takes one argument and returns that argument without evaluating it. For example:

```
> (quote (a b c))
(a b c)
> (quote (+ 1 3))
(+ 1 3)
```

Because `quote` is used so often, Lisp allows it to be abbreviated by a single quotation mark. Thus, the preceding examples could be written:

```
> ‘(a b c)
(a b c)
> ‘(+ 1 3)
(+ 1 3)
```

In general, `quote` is used to prevent the evaluation of arguments to a function when these arguments are intended to be treated as data rather than evaluable forms. In the earlier examples of simple arithmetic, `quote` was not needed, because numbers always evaluate to themselves. Consider the effect of quote in the following calls to the `list` function:

```
> (list (+ 1 2) (+ 3 4))
(3 7)
> (list ‘(+ 1 2) ‘(+ 3 4))
((+ 1 2) (+ 3 4))
```

In the first example, the arguments are not quoted; they are therefore evaluated and passed to `list` according to the default evaluation scheme. In the second example, `quote` prevents this evaluation, with the s-expressions themselves being passed as arguments to `list`. Even though `(+ 1 2)` is a meaningful Lisp form, `quote` prevents its evaluation. The ability to prevent evaluation of programs and manipulate them as data is an important feature of Lisp.

As a complement to `quote`, Lisp also provides a function, `eval`, that allows the programmer to evaluate an s-expression at will. `eval` takes one s-expression as an argument: this argument is evaluated as is usual for arguments to functions; however, the result is then evaluated *again* and this final result is returned as the value of `eval`. Examples of the behavior of `eval` and `quote`:

```
> (quote (+ 2 3))
(+ 2 3)
> (eval (quote (+ 2 3)))   ;eval undoes the effect of quote
5
> (list ‘* 2 5)   ;this constructs an evaluable s-expression
(* 2 5)
> (eval (list ‘* 2 5)) ;this constructs and evaluates the s-
expression
10
```

The `eval` function is precisely what is used in the ordinary evaluation of s-expressions. By making `quote` and `eval` available to the programmer, Lisp greatly simplifies the development of *meta-interpreters*: variations on the standard Lisp interpreter that define alternative or extended behaviors for the Lisp language. This important programming methodology is illustrated in the "infix-interpreter" of Section 15.2 and the design of an expert system shell in Section 17.2.

11.3 Programming in Lisp: Creating New Functions

Using defun

Common Lisp includes a large number of built-in functions, including:

- A full range of arithmetic functions, supporting integer, rational, real and complex arithmetic.
- A variety of looping and program control functions.
- List manipulation and other data structuring functions.
- Input/output functions.
- Forms for the control of function evaluation.
- Functions for the control of the environment and operating system.

Lisp includes too many functions to list in this chapter; for a more detailed discussion, consult a specialized Lisp text, the manual for your particular implementation, or see Chapter 20.

In Lisp, we program by defining new functions, constructing programs from this already rich repertoire of built-in functions. These new functions are defined using `defun`, which is short for *define function*. Once a function is defined it may be used in the same fashion as functions that are built into the language.

Suppose, for example, the user would like to define a function called `square` that takes a single argument and returns the square of that argument. `square` may be created by having Lisp evaluate the following expression:

```
(defun square (x)
     (* x x))
```

The first argument to `defun` is the name of the function being defined; the second is a list of the formal parameters for that function, which must all be symbolic atoms; the remaining arguments are zero or more s-expressions, which constitute the body of the new function, the Lisp code that actually defines its behavior. Unlike most Lisp functions, `defun` does not evaluate its arguments; instead, it uses them as specifications to create a new function. As with all Lisp functions, however, `defun` returns a value, although the value returned is simply the name of the new function.

The important result of evaluating a `defun` is the side effect of creating a new function and adding it to the Lisp environment. In the above example, `square` is defined as a function that takes one argument and returns the result of multiplying that argument by itself. Once a function is defined, it must be called with the same number of arguments, or "actual parameters," as there are formal parameters specified in the `defun`. When a function is called, the actual parameters are bound to the formal parameters. The body of the function is then evaluated with these bindings. For example, the call `(square 5)` causes `5` to be bound to the formal parameter `x` in the body of the definition. When the body `(* x x)` is evaluated, Lisp first evaluates the arguments to the function. Because `x` is bound to `5` by the call, this leads to the evaluation of `(* 5 5)`.

More concisely, the syntax of a `defun` expression is:

```
(defun <function name>
    (<formal parameters>) <function body>)
```

In this definition, descriptions of the elements of a form are enclosed in angle brackets: `< >`. We use this notational convention throughout this text to define Lisp forms. Note that the formal parameters in a `defun` are enclosed in a list.

A newly defined function may be used just like any built-in function. Suppose, for example, that we need a function to compute the length of the hypotenuse of a right triangle given the lengths of the other two sides. This function may be defined according to the Pythagorean theorem, using the previously defined `square` function along with the built-in function `sqrt`. We have added a number of comments to this sample code. Lisp supports "end of line comments": it ignores all text from the first "`;`" to the end of the same line.

```
(defun hypotenuse (x y)          ;the length of the hypotenuse is
    (sqrt (+ (square x)           ;the square root of the sum of
        (square y))))            ;the squares of the other sides.
```

This example is typical in that most Lisp programs are built up of relatively small functions, each performing a single well-defined task. Once defined, these functions are used to implement higher-level functions until the desired "top-level" behavior has been defined.

11.4 Program Control in Lisp: Conditionals and Predicates

Using cond

Lisp branching is also based on function evaluation: control functions perform tests and, depending on the results, selectively evaluate alternative forms. Consider, for example, the following definition of the `absolute-value` function (note that Lisp actually has a built-in function, `abs`, that computes absolute value):

```
(defun absolute-value (x)
      (cond ((< x 0) (– x))          ;if x < 0, return –x
            ((>= x 0) x)))              ;else return x
```

This example uses the function, `cond`, to implement a conditional branch. `cond` takes as arguments a number of *condition–action* pairs:

```
(cond (< condition1 > < action1 >)
    (< condition2 > < action2 >)
        …
    (< conditionn > < actionn >))
```

Conditions and actions may be arbitrary s-expressions, and each pair is enclosed in parentheses. Like `defun`, `cond` does not evaluate all of its arguments. Instead, it evaluates the conditions in order until one of them returns a non-nil value. When this occurs, it evaluates the associated action and returns this result as the value of the `cond` expression. None of the other actions and none of the subsequent conditions are evaluated. If all of the conditions evaluate to `nil`, `cond` returns `nil`.

An alternative definition of `absolute-value` is:

```
(defun absolute-value (x)
    (cond ((< x 0) (– x))            ;if x < 0, return –x
            (t x)))                      ;else, return x
```

This version notes that the second condition, `(>= x 0)`, is always true if the first is false. The "`t`" atom in the final condition of the `cond` statement is a Lisp atom that roughly corresponds to "true." By convention, `t` always evaluates to itself; this causes the last action to be evaluated if all preceding conditions return `nil`. This construct is extremely useful, as it provides a way of giving a `cond` statement a default action that is evaluated if and only if all preceding conditions fail.

Lisp Predicates Although any evaluable s-expressions may be used as the conditions of a `cond`, generally these are a particular kind of Lisp function called a *predicate*. A predicate is simply a function that returns a value of either true or false depending on whether or not its arguments possess some property. The most obvious examples of predicates are the relational operators typically used in arithmetic such as =, >, and >=. Here are some examples of arithmetic predicates in Lisp:

```
> (= 9 (+ 4 5))
t
> (>= 17 4)
t
> (< 8 (+ 4 2))
nil
> (oddp 3)           ;oddp tests whether or not its argument is odd
t
> (minusp 6)            ;minusp tests whether its argument < 0
nil
> (numberp 17)      ;numberp tests whether its argument is numeric
t
> (numberp nil)
nil
> (zerop 0)     ;zerop is true if its argument = 0, nil otherwise
t
> (plusp 10)                ;plusp is true if its argument > 0
t
> (plusp –2)
nil
```

Note that the predicates in the above examples do not return "true" or "false" but rather `t` or `nil`. Lisp is defined so that a predicate may return `nil` to indicate "false" and anything other than `nil` (not necessarily `t`) to indicate "true." An example of a function that uses this feature is the `member` predicate. `member` takes two arguments, the second of which must be a list. If the first argument is a member of the second, `member` returns the suffix of the second argument, containing the first argument as its initial element; if it is not, member returns `nil`. For example:

```
> (member 3 '(1 2 3 4 5))
(3 4 5)
```

One rationale for this convention is that it allows a predicate to return a value that, in the "true" case, may be of use in further processing. It also allows any Lisp function to be used as a condition in a `cond` form.

As an alternative to `cond`, the `if` form takes three arguments. The first is a test. `if` evaluates the test; if it returns a non-nil value, the `if` form evaluates its second argument and returns the result, otherwise it returns the result of evaluating the third argument. In cases involving a two-way branch, the `if` construct generally provides cleaner, more readable code than `cond`. For example, `absolute-value` could be defined using the `if` form:

```
(defun absolute-value (x)
      (if (< x 0) (– x) x))
```

In addition to `if` and `cond`, Lisp offers a wide selection of alternative control constructs, including iterative constructs such as do and while loops. Although these functions provide Lisp programmers with a wide range of control structures that fit almost any situation and programming style, we will not discuss them in this section; the reader is referred to a more specialized Lisp text for this information.

One of the more interesting program control techniques in Lisp involves the use of the logical connectives `and`, `or`, and `not`. `not` takes one argument and returns `t` if its argument is `nil` and `nil` otherwise. Both `and` and `or` may take any number of arguments and behave as you would expect from the definitions of the corresponding logical operators. It is important to note, however, that `and` and `or` are based on *conditional evaluation*.

In evaluating an `and` form, Lisp evaluates its arguments in left-to-right order, stopping when any one of the arguments evaluates to `nil` or the last argument has been evaluated. Upon completion, the `and` form returns the value of the last argument evaluated. It therefore returns non-`nil` only if all its arguments return non-`nil`. Similarly, the `or` form evaluates its arguments only until a non-`nil` value is encountered, returning this value as a result. Both functions may leave some of their arguments unevaluated, as may be seen by the behavior of the `print` statements in the following example. In addition to printing its argument, in some Lisp environments `print` returns a value of `nil` on completion.

```
> (and (oddp 2) (print "eval second statement"))
nil
> (and (oddp 3) (print "eval second statement"))
eval second statement
> (or (oddp 3) (print "eval second statement"))
t
> (or (oddp 2) (print "eval second statement"))
eval second statement
```

Because `(oddp 2)` evaluates to `nil` in the first expressions, the `and` simply returns `nil` without evaluating the `print` form. In the second expression, however, `(oddp 3)` evaluates to `t` and the `and` form then evaluates the `print`. A similar analysis may be applied to the `or` examples. It is important to be aware of this behavior, particularly if some of the arguments are forms whose evaluations have side effects, such as the `print` function. The conditional evaluation of logical connectives makes them useful in controlling the flow of execution of Lisp programs. For example, an `or` form may be used to try alternative solutions to a problem, evaluating them in order until one of them returns a non-`nil` result.

Exercises

1. Which of the following are legitimate s-expressions? If any is not, explain why it isn't.

```
(geo rge fred john)
(a b (c d (e f (g h)))
(3 + 5)
(quote (eval (+ 2 3)))
(or (oddp 4) (* 4 5 6)
```

2. Create a small database in Lisp for some application, such as for professional contacts. Have at least five fields in the data-tuples where at least one of the fields is itself a list of items. Create and test your own assess functions on this database.

3. Create a `cond` form that uses `and` and `or` that will test the items in the database created in exercise 2. Use these forms to test for properties of the data-tuples, such as to print out the name of a male person that makes more than a certain amount of money.

4. Create a function called `my-member` that performs the function of the `member` example that was presented in Section 11.4.