Structure And Interpretation Of Computer Programs (SICP)

Building Abstractions with Procedures

The Elements of Programming

Expressions

Prefix Notation

(+ 137 349)
486

(- 1000 334)
666

(* 5 99)
495

(/ 10 5)
2
; operator
(+ 2.7 10)
; operands
12.7

Expressions such as these, formed by delimiting a list of expressions within parentheses in order to denote procedure application, are called combinations. The leftmost element in the list is called the operator, and the other elements are called operands. The value of a combination is obtained by applying the procedure specified by the operator to the arguments that are the values of the operands.

Advantages

One of them is that it can accommodate procedures that may take an arbitrary number of arguments

```
(+ 21 35 12 7)
75

(* 25 4 12)
1200
```

A second advantage of prefix notation is that it extends in a straightforward way to allow combinations to be nested, that is, to have combinations whose elements are themselves combinations:

```
(+ (* 3 5) (- 10 6))
19
```

Evaluating Combinations

To evaluate a combination, do the following:

1. Evaluate the subexpressions of the combination. ; recursive
2. Apply the procedure that is the value of the leftmost subexpression (the operator) to the arguments that are the values of the other subexpressions (the operands).

First, observe that the first step dictates that in order to accomplish the evaluation process for a combination we must first perform the evaluation process on each element of the combination. Thus, the evaluation rule is recursive in nature; that is, it includes, as one of its steps, the need to invoke the rule itself

Next, observe that the repeated application of the first step brings us to the point where we need to evaluate, not combinations, but primitive expressions such as numerals, built-in operators, or other names

- the values of numerals are the numbers that they name,
- the values of built-in operators are the machine instruction sequences that carry out the corresponding operations, and
- the values of other names are the objects associated with those names in the environment.

special forms

evaluating (define x 3) does not apply define to two arguments the purpose of the define is precisely to associate x with a value.

Summary

Lisp has a very simple syntax; that is, the evaluation rule for expressions can be described by a simple general rule together with specialized rules for a small number of special forms.

Naming and the Environment

```
(define size 2)
```

define is our language's simplest means of abstraction

the possibility of associating values with symbols and later retrieving them means that the interpreter must maintain some sort of memory that keeps track of the name-object pairs. This memory is called the environment

The Substitution Model for Procedure Application

To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

Compound Procedures

Procedure definitions

The general form of a procedure definition is

```
(define (⟨name⟩ ⟨formal parameters⟩)
  ⟨body⟩)

(define (square    x)         (*      x         x))
  |       |       |           |      |         |
 To     square  something,  multiply  it  by  itself.
```

Conditional Expressions and Predicates

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

```
(and ⟨e1⟩ ... ⟨en⟩)
```

```
(or ⟨e1⟩ ... ⟨en⟩)
```

```
(not ⟨e⟩)
```

Example: Square Roots by Newton's Method

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))
```

sqrt-iter, on the other hand, demonstrates how iteration can be accomplished using no special construct other than the ordinary ability to call a procedure.

Procedures as Black-Box Abstractions

block structure

```
(define (sqrt x)
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess x) (average guess (/ x guess)))
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1.0 x))
```

lexical scoping

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

Abstraction

- ***primitive expressions,*** which represent the simplest entities the language is concerned with,
- ***means of combination***, by which compound elements are built from simpler ones, and
- ***means of abstraction***, by which compound elements can be named and manipulated as units.

In programming, we deal with two kinds of elements: procedures and data.

any powerful programming language should be able to describe primitive data and primitive procedures and should have methods for combining and abstracting procedures and data.

Building Abstractions with Procedures

Lisp possesses unique features that make it an excellent medium for studying important programming constructs and data structures and for relating them to the linguistic features that support them. The most significant of these features is the fact that Lisp descriptions of processes, called procedures, can themselves be represented and manipulated as Lisp data. The importance of this is that there are powerful program-design techniques that rely on the ability to blur the traditional distinction between "passive" data and "active" processes.

Procedures and the Processes They Generate

Formulating Abstractions with Higher-Order Procedures