Chapter 15, part I

Functional Programming Languages

--- with an Introduction to Scheme

You must <u>unlearn</u> what you have learned. -- Yoda

Contents

1.1. Overview of the four main programming paradigms [1]

The imperative paradigm

First do this and next do that

The 'first do this, next do that' is a short phrase which really in a nutshell describes the spirit of the imperative paradigm. The phrase also reflects that the order to the commands is important.

The object-oriented paradigm

Send messages between objects to simulate the temporal evolution of a set of real world phenomena

Three major principals are abstract data types, inheritance, and polymorphism. The object-oriented paradigm has gained great popularity in the recent decade. The primary and most direct reason is undoubtedly the strong support of encapsulation and the logical grouping of program aspects. These properties are very important when programs become larger and larger.

The functional paradigm

Evaluate an expression and use the resulting value for something

In functional programming what you have basically are a set of functions each of which performs a task. **Selectively** executing these function results in the solution to the problem at hand.

Functional programming is in many respects a simpler and cleaner programming paradigm than the imperative one. The reason is that the paradigm originates from a purely mathematical discipline: the theory of functions, while the imperative paradigm is rooted in the key technological ideas of the digital computer, which are more complicated and less 'clean' than mathematical function theory. In imperative languages, functions basically behave as fancy goto-statements with parameter passing. In functional languages, however, the notion of a function is closer to the mathematician's notion of function, namely, as an entity itself. A function is more naturally **thought of as an object rather than an operation to be executed**.

The logic paradigm

Answer a question via search for a solution

The logic paradigm is dramatically different from the other three main programming paradigms. The logic paradigm fits extremely well when applied in problem domains that deal with the

extraction of knowledge from basic facts and relations. The logical paradigm seems less natural in the more general areas of computation.

1.2. Brief history

- 1940s: Alonzo Church and Haskell Curry developed the lambda calculus, a simple but powerful mathematical theory of functions.
- 1960s: John McCarthy developed Lisp, the first functional language.
- 1978: John Backus publishes award winning article on FP, a functional language that emphasizes higher-order functions and calculating with programs.
- Mid 1970s: Robin Milner develops ML (Meta Language), the first of the modern functional languages, which introduced type inference and polymorphic types.
- Late 1970s - 1980s: David Turner develops a number of lazy functional languages leading up to Miranda, a commercial product.
- 1988: A committee of prominent researchers publishes the first definition of Haskell.
- 1990s–2000s – New typed functional languages emerge: OCaml, F#, Scala. FP ideas (immutability, higher-order functions) begin spreading into mainstream languages.
- 2010s – Growth of practical FP languages (Elm, Elixir, PureScript, Idris). Industry languages like Python, Java, and JavaScript adopt FP features such as lambdas and map/reduce.
- 2020s–today – Strong types (Idris, Lean), effect systems, and hybrid languages (Rust, Gleam, Mojo) push FP into systems, AI, and web development. FP concepts are now widely used even outside “pure” FP languages.

1.3. Characteristic of functional programming languages

We now describe the main properties of functional programming languages.

- Functions are first class values. That is, everything you can do with "data" can be done with functions themselves, such as passing a function to another function.
- Recursion is used as a primary control structure since traditional looping constructs (for, while) are absent or discouraged.
- Lists are a fundamental data structure (as seen in Lisp), though many functional languages also emphasize other immutable structures such as trees, tuples, and vector.
- "Pure" functional languages avoid side effects. Examples of side effects are modifying a global variable or static variable, modifying one of its arguments, writing data to a display or file, and more.

- Function Programming either discourages or outright disallows *statements*, and instead works with the evaluation of expressions (in other words, functions plus arguments). In the pure case, one program is one expression (plus supporting definitions).
- Much FP utilizes "higher order" functions (in other words, functions that operate on functions that operate on functions).

1.4. Data types

Numbers

Scheme (Guile) supports a rich “tower” of numerical types — integer, rational, real and complex — and provides an extensive set of mathematical and scientific functions for operating on numerical data. C-type languages have a variety of integer data types whereas Scheme has only one. The Scheme integer type is quite powerful, though, in that it can hold arbitrarily large numbers (there is no integer overflow). Similarly, C-type languages have two floating-point data types (float and double) whereas Scheme only has one. Most Schemes also have built-in support for rational numbers and complex numbers, but we won't be using them.

Booleans

Scheme has a real boolean type. "True" is designated by "#t" and "false" by "#f".

Characters

Scheme has a character data type:

	Scheme
'b' 'c' '\n' ' '	#\b #\c #\newline #\space

As you can see, some characters have mnemonic names e.g. #\newline.

Strings

Strings are one of the primitive expression types in Scheme. A string is a series of zero or more characters, spaces, and numbers, surrounded by double quotes.

When a string is evaluated, it returns that string. For example,

```
guile> "sam"
"sam"
guile> "Big 'ole bag of dirt"
"Big 'ole bag of dirt"
```

Pairs

Pairs are used to combine two Scheme objects into one compound object. Hence the name: A pair stores a pair of objects.

This syntax consists of opening a parenthesis, the first element of the pair, a dot, the second element and a closing parenthesis. The following example shows how a pair consisting of the two numbers 1 and 2, and a pair containing the symbols foo and bar can be entered. It is very important to write the whitespace before and after the dot, because otherwise the Scheme parser would not be able to figure out where to split the tokens.

```
(1 . 2)
(foo . bar)
```

But beware, if you want to try out these examples, you have to *quote* the expressions. The correct way to try these examples is as follows.

```
guile> '(1 . 2)
⇒(1 . 2)
Guile> '(foo . bar)
⇒(foo . bar)
```

Lists

A very important data type in Scheme—as well as in all other Lisp dialects—is the data type list. This is the short definition of what a list is:

- Either the empty list (),
- or a pair which has a list in its cdr.

Vectors

Both C-type languages and Scheme have arrays, but in Scheme, arrays are called "vectors" and can contain objects of any type, whereas in C-type languages, they can usually only contain objects of a single type.

Vectors are written using the notation #(obj ...). For example, a vector of length 3 containing the number zero in element 0, the list (2 2 2 2) in element 1, and the string “Anna” in element 2 can be written as following:

```
#(0  (2 2 2 2)  "Anna")
```

1.5. Atoms and lists

There are basically two main constructs in scheme: atoms and lists. Atoms are elements that **cannot be broken down any further** and cannot be used as used as containers to hold other

data. They are mostly numbers, strings, symbols, booleans, and characters. A list is a sequence of elements, each of which is either atoms or lists. Lists are denoted by enclosing the elements within a pair of matching parentheses.

Atoms	Lists
1	(x y z)
2.3	(1 2 3 4 5)
X	((1 2) (3 4) 5)

For instance,

```
(A B C D)
```

is a list consisting of four elements. The first is symbol A. A symbol is one character or a character sequence. They are often used for identifier names. Refer to “Symbols_strings_variabls.pdf” for the difference between symbols, variables and strings.

```
(A (B C D) E (F (G H)))
```

is a list of four elements. The second is a list of three symbols, namely, (B C D). The third is symbol E. Finally, the fourth element is a list, which consists of a single symbol F and a list of two elements (G H).

Note that, unlike most other languages, extra parentheses may change the semantics of the code. So, for instance,

```
(A)
```

is different from

```
((A))
```

since the former is a list of a single element A, while the latter is a list of a single list (A), which in turn consists of a single element A.

Comments

Usually the only other syntactic form you will see in Scheme is comments, which start with a semicolon ";" and go to the end of the line:

```
; This is a comment in Scheme.
```

There is no standard multi-line comment in Scheme. Some implementation has its own way of block comments. It is quite typical in Scheme to have a comment that starts with more than one semicolon; this is just to make the comment look nicer and has no other meaning:

```
;; This is another comment in Scheme.
```

1.6. Basic Arithmetic

In scheme, mathematical operations are expressed in prefix syntax rather than in infix syntax. "Infix" means that the operator (e.g. +, -, *, /) comes between the operands (e.g. numbers or variables) whereas "prefix" means that the operator comes before the operands. Here are some examples:

C	Scheme
a + b	(+ a b)
(a - b) * 2	(* (- a b) 2)

In Scheme, "operators" are just ordinary functions in Scheme and there are no operator precedence levels. For instance, compare these examples:

C	Scheme
(a * b) – c	(- (* a b) c)
a * (b - c)	(* a (- b c))

In the C case, does "a * b - c" mean "(a * b) - c" or "a * (b - c)"? In fact, it means the former, but we need to know that the precedence of "*" is higher than that of "-" to figure this out. In the case of Scheme, the two possibilities are syntactically different so there is never any ambiguity. The price we pay for this is that parentheses are not optional in Scheme; every parenthesis has to be there, and additional parentheses change the meaning of an expression.

Scheme includes the four basic arithmetic operations of +, -, *, and /. Here are some examples:

Expression	Value
`42`	`42`
`(* 3 7 )`	`21`
`(+ 4 2 7)`	`13`
`(- 5 1 3)`	`1`
`(/ 24 2 3)`	`4`
`(- 24 (* 4 3 ))`	`12`

Again, note that extra parentheses make a difference. So, for example, (+ 4 ((+ 3 5))) really doesn't make sense since it reduces to (+ 4 (8)) and it doesn't make sense to try to add the literal 4 to the list (8).

1.7. Quoting

The very simplest program outputs a single literal. For example, consider the following one-line program:

```
4
```

Running this program through the Scheme interpreter outputs the result 4. Big deal. However, consider instead the one-line program:

```
(4 5)
```

In this case, the scheme interpreter reports an error. That's because, by default, scheme interprets a list as a function invocation. In particular, any construct in the form:

(function parameter1 parameter2 parameter3 ...)

is seen by the interpreter as a function invocation. Thus, something like

```
(4 5)
```

is interpreted as a function name 4 followed by a parameter of 5.

To override this default behavior and force scheme to interpret a list as simply a list and not a function invocation, we precede the list with a single quote. So, for example, the single-line program:

```
'(4 5)
```

outputs the list (4 5). This technique is called quoting. Note that we can replace the single quotation mark with the more verbose notation QUOTE, i.e.

```
QUOTE (4 5)
```

Problem: What does the following program output:

```
(+  4  5 )
```

What about:

```
'(+  4  5)
```

1.8. Predicate Functions

There are several predicate functions in scheme, e.g. eq?, null?, and list?. Each of these returns a Boolean value. The two Boolean values are #t or #f. Alternatively, in Scheme, the empty list is synonymous with false, while any nonempty list is synonymous with true.

eq? tests for the same object (essentially a pointer comparison). This is fast, and can be used when searching for a particular object, or when working with symbols (which are always unique objects). For example:

Expression	**Value**
`(eq?  'A  'B)`	#f
`(eq?  'A  'A)`	#t
`(eq?  'A  'a)`	#f
`(eq?  'A  '(A B))`	#f
`(eq?  #\b #\b)`	#t
`(eq?  "today" "today")`	#f
`(eq?  '(A B)  '(A B))`	not specified (Guile returns #f)

As the last case illustrates, the result of comparing two lists is actually compiler dependent; some Scheme compilers return true while others return false. Why? The reason for this difference is that eq? is often implemented **as a pointer comparison**, and two lists that are exactly the same are often not duplicated, but not always.

It's important to keep in mind that eq? can really only be reliably applied to symbolic atoms and not (as we saw above) to lists, strings, etc. Moreover, one cannot reliably apply eq? to numerical atoms. A better idea is to use the equals function instead: e.g.

```
(=  4  8)
```

More generally, one can use a variety of predicate functions designed specifically for numerical data:

```
=, <>, >, <, >=, <=, even?, odd?, zero?
```

Note that the = predicate works for numerical but not necessarily symbolic elements. For string comparison, use `string=?` Use `char=?` for character comparison.

```
(string=? "today" "today") ; returns #t
(char=? #\b #\b)           ; returns  #t
(char=? #\b #\c)           ; returns  #f
```

In some instances, it's convenient to be able to test for equality when it's not known whether the quantities being compared are numerical or symbolic. In these cases, one can alternatively use a different predicate eqv? in place of = or eq?. **eqv? extends eq? to look at the value of numbers and characters.** However, one should use eqv? sparingly since its run-time execution is longer. `eqv?` compares both type and value.

```
(eqv? 3 (+ 1 2)) ⇒ #t
(eqv? 1 1.0)     ⇒ #f
```

Another useful equality comparison is equal? **equal? goes further, it looks (recursively) into the contents of lists, vectors, etc**.

```
(equal? (list 1 2 3) (list 1 2 3))   ⇒ #t
(equal? (list 1 2 3) (vector 1 2 3)) ⇒ #f
```

Two other useful list predicate are list? and null?. The former accepts a single parameter and return true if the parameter is a list and false otherwise. The latter accepts a single parameter and returns true if that parameter is an empty list and false otherwise. Note that something like:

```
(null? '( () ))
```

returns false since the parameter is not actually an empty list; rather, it is a list having a single element consisting of an empty list.

1.9. Processing Lists

There are two operations for chopping up lists: car and cdr. There are two operations for constructing lists: cons and list.

The car function returns the first element of a given list. For example:

Expression	Value
`(car '(A B C ) )`	`A`
`(car '((A B) C D))`	`(A B)`
`(car 'A)`	`error message`
`(car '(A))`	`A`
`(car '())`	`error message`

The cdr function returns the remainder of a given list after its car is removed:

Expression	Value
`(cdr '(A B C))`	`(B C)`
`(cdr '((A B) C D)`	`(C D)`
`(cdr 'A)`	`error`
`(cdr '(A))`	`()`

While car and cdr are used to chop up a list into parts, the cons operation is used to build up lists.

The cons function takes two arguments. The first argument can be either an atom or a list; the second is usually a list. The effect is to create a new list which has the first parameter as its first element (i.e. as its car part) and the second parameter as the remainder of the list (i.e. as its cdr part). So, for instance,

Expression	Value
`(cons 'A '(B C))`	`(A B C)`
`(cons '(A B) '(C))`	`((A B) C)`
`(cons '(A B) '(C D))`	`((A B) C D)`
`(cons '() '(A B))`	`(() A B)`
`(cons 'A '())`	`(A)`

LIST is basically a shorthand notation for building a list, e.g.

```
(list    'foo1   'foo2  'banana)  ; (foo1  foo2  banana)
```

produces the list (foo1 foo2 banana).

Problem: What does following program output:

```
(cons    (car '(A   B  C  D)) (cdr    '(A  B   C  D)))
```

1.10. Lambda Functions

The mathematical function is a mapping from a domain to a range and a function transfers values from the domain to values in the range. Lambda calculus is a very terse notation of functions and function application.

"Lambda calculus (also written as **λ-calculus**) is a formal system in mathematical logic and computer science for expressing computation based on function abstraction and application using variable binding and substitution."

The use of the term "variable" is also a little misleading in that you don't associate it with storing values and memory locations. It is simply a symbol that you can manipulate. It turns out - the lambda calculus is a functional programming language. In Scheme, one can use the lamda function to create functions. For example, the following:

```
(lambda (L) (car (cdr L)))
```

defines a nameless function whose task is to return the second element from the list L. More generally, the syntax for defining a nameless function looks like:

```
(lambda (parameter1 parameter2 ...) (body_of_function))
```

Once we define a nameless function, we can apply it to a list; e.g.:

```
((lambda  (L)  (car (cdr    L)))   '(A B C))
```

Problem: Write a nameless function that returns the third element from the list L.

1.12. Defining Stuff

The define function can be used in two ways: to bind a name to a value and to bind a name to a lambda expression. To bind a name to a value, we use the syntax:

```
(define   symbol  expression)
```

For instance,

```
(define   pi  3.14159)
(define  two_pi  (* 2 pi))
```

This would bind the value 3.14159 to the symbol pi and 6.28318 to the symbol two_pi.

Similarly, DEFINE can be used to bind a nameless function to a symbol. The syntax is the usual:

```
(define symbol nameless_function)
```

So, for example, the following scheme program creates a function that returns the second element of a list, binds this function to the name foo, and applies this function to the list (S C H E M E):

```
(define foo
(lambda (L)
 (car (cdr L))
))
```

To apply this function:

```
(foo '(S  C  H   E  M  E))
```

Problem: Write a function that returns the third element of a list L, bind this nameless function to the symbol foobar, then apply this function to the list (S C H E M E).

In the case of binding a symbol to a nameless function, we can use a slightly easier (more familiar) notation:

```
(DEFINE (function_name parameter1 parameter2 ...)
   body_list
)
```

So, for example, the following defines a function square, which essentially squares a number:

```
(define (square number)
(* number number)
)
```

We can then use this function, for instance, to compute the length of the hypotenuse of a right triangle given the length of the two legs:

```
(define (hypotenuse s1 s2)
(sqrt (+ (square s1) (square s2)))
)
```

Here, the function SQRT is a function already defined by Scheme. Note that the more verbose version of this definition looks like:

```
(define hypotenuse
(lambda (s1 s2) (sqrt (+ (square s1) (square s2))))
)
```

1.11. Flow of Control

Unlike imperative programming languages, functional programming languages like Scheme have two main mechanisms for flow of control: conditional and recursion. This reflects the fact that mathematicians typically use only these two operations for defining functions. Let's look at an example. Consider the mathematical function f(n) = n!. In mathematics, we define f(n) via:

$$f(n) \equiv \begin{cases} 1 & if\ n = 0 \\ n * f(n-1) & if\ n > 0 \end{cases}$$

In Scheme, we'd define factorial as follows:

```
(define (factorial n)
(if  (= n  0)  1
     (*  n  (factorial (-  n  1)))
))
```

Alternatively, we could use the more verbose version:

```
(define factorial
(lambda (n)
(if (= n 0)   1
       (* n (factorial (- n 1))))
))
```

More generally, the syntax for an IF predicate looks like:

```
(if  predicate  then_expression  else_expression )
```

In traditional recursion, the typical model is that you perform your recursive calls first, and then you take the return value of the recursive call and calculate the result. In this manner, you don't get the result of your calculation until you have returned from every recursive call.

A tail-recursive version of the same problem is below. In tail recursion, you perform your calculations first, and then you execute the recursive call, passing the results of your current step to the next recursive step. The consequence of this is that once you are ready to perform your next recursive step, you don't need the current stack frame any more. This allows for some optimization. In fact, with an appropriately written compiler, you should never have a stack overflow snicker with a tail recursive call.

```
(define (tail-factorial n a)
    (if (= n 0)
        a
        (tail-factorial (- n 1) (* n a)))
  )

(tail-factorial 5 1)  ; to call the function
```

A more general selector, resembling an if-else chain or switch statement, looks like:

```
(cond
   (predicate_1 a_bunch_of_expressions)
   (predicate_2 a_bunch_of_expressions)
)
```

Here's how cond works: the predicates of the parameters are evaluated one at a time, in order from first to last. The first predicate that yields a #t is selected. The expressions that follow this predicate are then evaluated and the value of the last expression is returned as cond's return value.

For instance, here's scheme code that can be used to display an appropriate message upon comparing x to y:

```
(define (compare x y)
  (cond
  ((> x y) (list x 'is 'bigger 'than y))
  ((< x y) (list x 'is 'smaller 'than y))
  (#t (list x y "are equal"))
  ))
```

Note that the final option is the literal true, which basically serves as a trailing else-statement.

Finding the Length of a List

Scheme has a standard function length that determines the number of elements in a list. What does its implementation look like? The pseudocode for length looks something like this:

```
boolean length(lst){
if (lst is empty){
return 0
}
else {
return length(cdr(lst)) + 1
}
}
```

This analysis yields the following Scheme code:

```
(define (length lst)
(cond
    ((null? lst)     0)
    (#t   (+ 1 (length (cdr lst))))
))
```

We again emphasize that length is already a standard function in Scheme.

The tail-recursion version

```
(define (length-1 lst result)
  (if (null? lst) result
      (length-1 (cdr lst) (+ result 1))))

(length-1 '(c a n y o n) 0)
```

Problem: Write a simple function that adds the first n integers. (e.g. (sum 5) returns 15).