Functional Programming

Imperative vs Applicative

Imperative languages rely heavily on assignment and on programmed changes in memory to accomplish a program's objectives. Control is basically routing processes from one assignment statement to another.

Applicative languages rely on function application, passing computational results from one function to another. Control is achieved through function nesting.

Functions are mathematical objects which have single entry and exit points. Functions return a value. A function itself *names* its return value. If the function is unevaluated, then the name is compound. When the function is evaluated, it returns a simpler name. Since functions are names, the text of a function, such as F[x], serves the same purpose as a variable, to name a particular part of a computation. More generally, function names can be compounded, so that two composed functions yield a compound name for the composition of the two functions.

The hallmark of a functional program is that it is composed of function invocations and conditionals only. Functional programs contain no variables, no loops, no explicit sequencing, and no assignments.

Functions do not permit side effects. Therefore whenever a function is evaluated with the same arguments, it produces the same results. This is called ***referential transparency***; a type of what-you-see-is-what-you-get. Absence of side effects makes the meaning, or semantics, of a functional program fairly simple. Another advantage is a simple syntax which is easy to parse and error-check. Thus both syntax and semantics of functional programs is cleaner that imperative programs.

Imperative languages are architecture specific in that they evolved for programming a vonNeumann processor. Applicative languages suggest a functional architecture. Many such machines have been built in university settings, however a diversity of hardware architectures has yet to reach personal computing.

Uniformity and Simplicity

Pure LISP is unique in that it has only one data structure (the ***list***), and two control structures (***function-invocation as recursion*** and ***if-then-else***). This smallness and uniformity makes the language design and implementation very elegant and efficient.

The most significant implication of uniformity is that data and program have the same structure (both are lists). Thus LISP programming emphasizes *metaprogramming*: writing programs which return other programs as output. LISP encourages *extreme abstraction*. For example, rather than writing a parser for a particular language, LISP style would be to write a parser generator which takes a language as input and returns a parsing program for that language as output.

Another level of uniformity in functional languages is that during processing, all functions have only one argument. The process of converting binary (and in general n-ary) functions to unary functions is called *currying*. Schematically:

```
F[a,b] => G[a][b]
```

The function `G[a]` is a functional, returning a function `G'`. `G'` then applies to the single argument `b`.

Another simpler way to achieve unary functions is to collect all arguments into a list and pass the list as a single argument. Schematically:

```
F[a,b] => G[(a,b)]
```

Recursion vs Iteration

Almost all mathematical structures are defined and proved using induction. Recursion implements induction. Thus recursive style has these benefits:

- aligns with the mathematical approach
- easier for most data structures in most cases
- elegant and difficult to learn

Here is an example function which is straight-forward when written recursively and quite difficult when written iteratively. It tests the equality of two trees.

```
equal[x,y] =def=
      (atom[x] and atom[y] and x=y) or
        (not[atom[x]] and not[atom[y]] and
          equal[first[x],first[y]] and equal[rest[x],rest[y]])
```

The iterative version requires parallel iteration of two variables, one for tree `x` and one for tree `y`. The difficulty for the iterative version occurs when `x` and `y` are not atoms. How does a loop manage to deconstruct the left and right branches of the tree without using recursion?

This example illustrates that although iteration and recursion are theoretically equivalent, from a programming point of view, recursion is both more powerful and more elegant than iteration.

Imperative and applicative styles can be contrasted by comparing iteration with recursion.

Iterative: `for I in 1..X do <accumulate results in A>`

The above imperative loop contains three variables, each with a significantly different purpose.

`I`: the *iteration* variable, declared and scoped by the loop itself

`A`: the *accumulation* variable, again scoped within the loop, used to return values

`X`: the *input* variable, used as a parameter of the function containing the loop

Functional programs minimize the use of variables, since functional programs do not rely on memory manipulation for storing computational results.

Applicative: `if X=0 then nil else (A + <recur on incremented X>)`

In applicative, or functional, programming, the loop construct is replaced by recursion. Each recursive call modifies the input parameter until that parameter reaches a ground value.

Iteration variables are eliminated in favor of structured recursive descent

Accumulation variables are eliminated in favor of accumulating the results of each recursive function call as they return values to the containing function.

Input variables directly incorporate iterations and accumulations. The input variable is incremented to achieve iteration. The output of a recursive function incorporates the accumulated results.

Pure Functions and Lambda

Pure functions, or **operators**, are functions which do not have any arguments. Rather operators apply to their context, to whatever forms are juxtaposed to the operator. Operators have a space for arguments, but the arguments are not constrained to any particular type, to a reduced expression, or even to data.

Lambda calculus is the mathematical system upon which functional programming is based. It has two defined operations, *apply* and *abstract*. A lambda expression defines an operator. For example, the *square* operator:

```
square =def= lambda[#,#*#]
```

The hashmark # stands in place of a variable, but it is not a variable per se. Rather it represents a *space* or a *hole*, a place where any form can be placed. The second form, *#*#*, is not an argument, it is rather the body of the lambda expression. A hole stands in place of the **context** of the expression.

The *Rule of Application* is simply that any form following the lambda expression is substituted in place of the hashmark. Thus, lambda expressions can take functions or data structures as "arguments". In the above example:

```
lambda[#,#*#] 4 ==> 4*4 ==> 16

lambda[#,#*#] F[x+1] ==> F[x+1]*F[x+1]
```

In the first example, the `4` following the square-lambda replaces the hashmark, leaving a body `4*4` to be evaluated. In the second example, the hashmark is replaced by a function call. In general:

```
Apply[E1,E2] =def= <substitute E2 for # in E1>
```

The *Rule of Abstraction* is the inverse of apply; it provides a mechanism for removing the dependence on variables by functional expressions. In general:

```
Abstract[F[x]] =def= lambda[#,F]
```

Mapping and Functionals

Consider squaring each member of a list of integers. Here are three programs representing three different styles:

Iterative:
```
for each element in list do
  <square element; store into accumulator>
```

Recursive:
```
if list=nil then nil else
  <add square[first[list]] to recur[rest[list]]
```

Mappable:
```
map <square> onto list
```

A data structure is *mappable* when a function applied to the entire list gives the same result as applying the function directly to each element in the list.

The **map** function takes two arguments, a function and a list. The function argument applies to a single item, or element on the list. The map function takes care of the mechanics of iteration. When a function takes another function as an argument, that function is called a *functional*, a function which acts on other functions. Many languages do not allow functionals.

Filtering is a type of mapping in which the applied function is a Boolean test for membership in the returned list. The filter function is a functional:

```
filter[test-fn, list] =def= map[test-fn, list]

test-fn[i] =def= if <i meets criteria> then i else <nothing>
```

Functionals can also return functions as output. Here is an example which selects the appropriate operator for a data type:

```
pick-op[i] =def=
  case type[i]
    integer:      +               ;add
    list:         cons            ;push onto list
    string        ·               ;prefix
```

Another example is the functional `apply`: to apply a binary function to a list of arguments the arguments are taken two at a time, combined using the function, and then the result is returned to the list. When all arguments are processed, a single value remains. *Example*:

```
apply[+,(1,3,5)] ==> apply[+,(4,5)] ==> 9
```

Mapping and similar functionals treat entire *data structures as single objects*. This is an example of the extreme abstraction of functional languages. Functionals provide the right level of abstraction for any compound data structure.

In all processes which address a program rather than a data structure (within a program) require function-level analysis: the "objects" being transformed are functions rather than data. Examples of program manipulation include proof of correctness, optimization and compilation, program derivation, and metaprogramming.

Combinators

Combinators can be seen as macros for lambda calculus. More accurately, they are a set of functions which provide the same functionality as lambda calculus without the lambda construct. Combinators encapsulate all control structures. Another perspective is that combinators are a pattern-matching language for functions.

The syntax of a lambda calculus expression is a sequence of forms. For example, factorial expressed as a pure function (@ is used as another blank notation, like #):

```
fac =def= Y lambda[#, lambda[@, if @=0 then 1 else @*#[@-1]]]
```

The `Y` combinator is the way lambda calculus handles recursion. In effect, it rewrites the entire definition whenever it is called recursively.

There are two elementary combinators: `K` and `S`. `K` is True, and `S` is Sequence. These combinators replace lambda expressions using the following set of recursively defined rules:

```
lambda[#,E] ==> K E                    when E = constant, variable, or combinator

lambda[#,var] ==> S K K                when var=#

lambda[#,E1 E2] ==> S lambda[@,E1] lambda[%,E2]
```

A set of pattern matching rules reduces combinator expressions. For example,

```
K E1 E2 ==> E1
```

This can be read as a conditional:

```
if K then E1 else E2                   where K = True.
```