1 Introduction

A computer is a mechanism for interpreting a language. It interprets (performs
the actions specified in) sentences in a language which is known as the com-
puter’s machine language. It follows, therefore, that a study of the organization
of computers is related to the study of the organization of computer languages.

Computer languages are classified in a variety of ways. Machine languages
are rather directly interpreted by computers. Higher level computer languages
are often somewhat independent from a particular computer and require trans-
lation (compilation) to machine language before programs may be interpreted
(executed).

Abstraction is an important concept in computing. Generally, higher level
languages are more abstract. A key tool of abstraction is the use of names. An
item of some complexity is given a name. This name is then used as a building
block of another item which in turn is named and so on. Abstraction is a tool
for managing complexity.

2 What is Functional Programming?

Functional programming languages (applicative languages) differ from conven-
tional programming languages (imperative languages) in at least the following
ways:

- In imperative languages, names are associated with memory cells whose
  values (state) change during the course of a computation.

- In applicative languages, names are associated with items which are stored
  in memory. Once created, in memory, an item is never changed. Names
  are assigned to items which are stored in memory so that they may be
  referenced. Items stored in memory are used as arguments for subsequent
  function applications during the course of a functional computation.

For example, in C we might write:

    int foo;
In this example we associate the name foo with a particular memory cell of size sufficient to hold an integer. Its state at that moment is unspecified. Later foo is assigned the value 4, i.e., its state is changed to 4. In J we might write:

```j
foo =: 4
```

An item 4 is created in memory and the name foo is assigned to that item. Note that in C we say that the value 4 is assigned to foo, but in J we say that the name foo is assigned to the value 4. The difference is subtle. With imperative languages the focus is on the memory cells and how their state changes during the execution of a program.

With functional languages the focus is on the items in memory. Once an item is created, it is never changed. Names are more abstract in the sense that they provide a reference to something which is stored in memory, but is not necessarily an ordinary data value. Functions are applied to items producing new items and the process is repeated. Names need be assigned to items only when needing to refer to an item in memory. The focus is on the items themselves rather than memory cells. The names are used as an abstraction tool.

- In imperative languages, computations involve the state changes of named memory cells.

For example, consider the following C program:

```c
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])  
{ int sum, count, n;
  count = 0;
  sum = 0;
  while (1 == scanf("%d", &n))  
  { sum = sum + n;
    count++;
  }
  printf("%f", (float)sum / (float)count);
  exit(0);
}
```

This program reads a standard input stream of integer values and computes the average of these values and writes that average value on the
standard output stream. At the beginning of the average computation, memory cells count and sum are initialized to have a state of 0. The memory cell n is changed to each integer read. Also, the state of sum accumulates the sum of each of the integers read and the state of count is incremented to count the number of integers read. Once all of the integers have been read, their sum is accumulated and the count is known so that the average may be computed.

To use the C average program one must compile the program.

[jhowland@Ariel jhowland]$ make ave
cc  ave.c  -o ave
[jhowland@Ariel jhowland]$ echo "1 2 3" | ave
2.000000
[jhowland@Ariel jhowland]$

• In functional languages, computations involve function application. Complex computations require the results of one application be used as the argument for another application. This process is known as functional composition. Functional languages have special composition rules which may be used in programs. Functional languages, being based on the mathematical idea of a function, benefit from their mathematical heritage. The techniques and tools of mathematics may be used to reason (simplify, transform and prove correctness) about programs.

For example, consider the following J program:

```
+/ % #
```

This program computes the average of a list of numbers in the standard input stream. The result (because no name assignment is done) is displayed on the standard output stream.

- The program is small.
- There are no names in the program.
- Numbers are not dealt with one by one.

To use the J average program you put the program and list of numbers in the standard input stream of a J machine (interpreter).

[jhowland@Ariel jhowland]$ echo "(+/ % #) 1 2 3" | jconsole
(+/ % #) 1 2 3
2
[jhowland@Ariel jhowland]$

The J average program consists of three functions.

- */ sum
- \% divide
- \# tally

When a three function program is applied to a single argument, x, the following composition rule, fork, is used.

\[(f \ g \ h) \ x = (f \ x) \ g \ (h \ x)\]

In the J average program, \(f\) is the function \(+/\), sum, and \(g\) is the function \(\%\), divide, and \# is the function tally which counts the number of elements in a list.

In the J average program, \(/\) is a function whose domain is the set of all available two argument functions (dyads) and whose result is a function which repeatedly applies its argument function between the items of the derived function’s argument. This is notable because functions may be applied to functions producing functions as results. Most imperative languages do not have this capability.

For example, the derived function \(+/\) sums the items in its argument while the derived function \(*/\) computes the product of the items to which it is applied.

- Functional languages deal exclusively with expressions which result in values.

- Imperative languages use language constructs which describe the state changes of named memory cells during a computation, for example, a \texttt{while} loop in C. Such languages have many sentences which produce no values or which produce changes of other items as a side-effect.

- Imperative languages may have statements which have \textit{side-effects}. For example, the C average program contained the statement \texttt{count++;} which references the value of \texttt{count} (the average program did not use this reference) and then increments its value by 1 after the reference. The C average program relied on the side-effect.

- Pure functional languages have no side-effects.