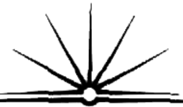


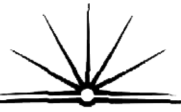
A I In the imperative paradigm a function and a procedure are sub-modules or subroutines that perform (theoretically) one logical task. Both accept and return data through parameters although a function is different to a procedure in that it returns a value with the function name such that it can be used in directly in assignment. Consider the Pascal:

```
procedure Sum(Num1, Num2, Res : Integer)
begin
    Res := Num1 + Num2;
end;

function Sum(Num1: Integer; Num2: Integer)
begin
    Sum := Num1 + Num2;
end;
```

The function can be called simpler:

```
Writeln( Sum(3,2));
```



which would display '5'. To do so with the procedure would involve a variable to hold Res.

In imperative programming all function and procedures are called, originally from the program main line.

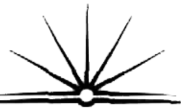
In the functional paradigm the entire program is a continuous functions. That is to say that rather than being invoked in a logical sequence governed by sequence and selection control structures the program executes by executing one function after the other in a given order depending on the language. Sequence can be controlled by repeatedly calling the same function again with a stopping condition (recursion) and selection can be performed by writing a function that branches control. When we say the program is entirely made of functions, it is best illustrated in LISP:

```
(+ (+ 3 4 6) (* 3 2) (+ 1 3))
```

Here there are four functions, the code is fairly self explanatory only two pre-defined functions are used (+ and *) that perform as in maths. But the structure of the program is such that only functions exist and when executed the result of 23 is displayed.

BOARD OF STUDIES
NEW SOUTH WALES

II the functional paradigm comes out of a two-fold need. Firstly it is designed to more closely model mathematical functions. The functional languages are therefore extremely powerful like mathematical functions, APL being famous for 'being able to create an accounting package in a few undecipherable keystrokes' and retain many of the properties of functions like domain and range but are able to deal with atoms or elements of all types and thus extend beyond numbers and into string data. By facilitating recursion much so more simply than imperative languages they also gain an advantage in complex, repetitive algorithms. The second reason for the development of functional languages, particularly LISP was for AI. Whilst this didn't eventuate and most expert systems today use Prolog, LISPs ability to work with large list functions and process therein seems ideal for making decisions based on a variety of facts provided. Functional languages went a long way towards freeing the developer from the constraints of the von Neumann architecture allowing programmers to worry about what they wished to achieve not how the computer was processing it.



BOARD OF STUDIES
NEW SOUTH WALES

B I The paradigm shown here is the logic paradigm. While this program appears to have some flow and structure it also bears all its marks. Firstly, in the init_database sub-module a list of facts is entered, that is, it is asserted that Australia, spain France are countries canberra paris madrid are capitals and then matches the two. Next we see a series of rules or predicates for matching up countries to capitals. The show-capital predicate states that for country(x) and capital(x,y), display the capital and country, it also says to say that if city(x), then show that x is a city not a country. If both these predicates fail then a 'not in database' message is shown. Finally recursion is used in the 'go' routine, if x is not 'quit' then 'go' is called again, The stopping case is that x = 'quit'

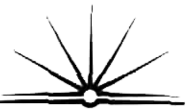
II show_capital (x) :-

```
        write (x) write($ add to database? $), nl,
        read (z),
        z = yes,
        add (x),
        !.
```

a new 'module' see next

. . .

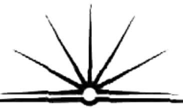
03 WB4



[Cont]

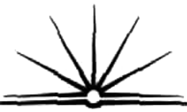
```
add(X) - nl,
        write($enter capital name$),
        read(A),
        assert(country(X)),
        assert(city(A)),
        assert(capital(X, A)).
add(x) :- nl.
```

c OOP has become huge in terms of software development, this is because it retains the power and efficiency of imperative programming, remains close to the von Neumann architecture, and is often an extension of widely used imperative languages (even Object COBOL exists). The imperative approach is generally fine for this sort of project as it contains clearly defined modules and sub-modules with specific abilities, the bank also has the resources to undertake the large-scale development process involved with imperative development, and indeed the mission-critical nature of the systems means such a process would be engaged no matter what paradigm was used. OOP does however have advantages over imperative that has led



to its wide adoption and use in this case. There are namely encapsulation and polymorphism. Encapsulation allows the separation of attributes (data) and the methods that access them, thus here the customer details can be separated from methods used to retrieve and set them, this means that the data cannot be set out of a specific range that the system is not designed to handle, it makes the system more full proof and is particularly important if other systems interface with this system as they may be created a different time and interact differently to components of the existing system. A probable other system that needs to interface with this system is that for ATMs where much business is conducted. The Object-Oriented paradigm also provides inheritance and polymorphism. There are many commonalities between processing a cash deposit and cheque deposit (and electronic deposit for that matter), OOP allows a class structure to be built with common features in a parent class. Polymorphism then allows these objects to behave as their parents and invoke common methods.

The functional paradigm involves splitting a problem into its smallest components, in the case of a bank this still involves a rigid and sequential process for many things, for this the



BOARD OF STUDIES
NEW SOUTH WALES

Functional paradigm is not designed and tends to fail in a mess of many functions.

The logic paradigm is designed to make decisions based on many facts. Yet in the case of a bank, again a strict procedure is to be followed. there are no rules and decisions to infer as such, just a process that shares some commonality to other processes.

Hence functional and logic fail to ~~w~~ work successfully in a banking situation and OOP offers numerous advantages ~~of~~ over imperative design.