Introduction to Functional Programming

The functional language community

The functional language community is excessively dour. The functional ascetics forbid themselves facilities which less pious programmers regard as standard . When using functional languages we do away with notions such as variables and reassignments. This allows us to define programs which may be subjected to analysis much more easily. When a value is assigned it does not change during the execution of the program. This property is referential transparency. There is no state corresponding to the global variables of a traditional language or the instances of objects in an object oriented language. When a definition is made it sticks. Reassignment does not take place. Getting used to this and finding alternatives the traditional structures such as loops which require reassignment is one of the hardest tasks for a programmer "converting" from a traditional language. The line

```
x := x+1;
```

may appear in a 3rd generation language and is understood to indicate that 'box' or 'location' referred to as 'x' has its contents incremented at this stage. We do not admit such concepts. 'x' is 'x' and 'x+1' is one more than x; the one may not be changed into the other. A program without a state is a simpler thing - it is easier to write the code and easier to reason about the code once written. It is harder to write poor code.

Functional languages are considered, by their devotees, to be higher level than third generation languages. Functional languages are regarded as declarative rather than imperative. Ordinary third generation languages such as Pascal, C (including flavours such as C++) and assembly instruct the computer on how to solve a problem. A declarative language is one which the programmer declares what the problem is; the execution of the program is a low level concern. This is an attitude shared with the logic language community (Prolog people).

Towards Correct Programs

There has been a great deal of progress in recent years in defining methodologies and design techniques which allow programs to be constructed more reliably. Some would claim that object orientation for example builds on and improves on structured programming which undoubtedly contributes to a better process of software construction. Using a rational methodology software engineers can produce better code faster - this is to be applauded, however it does not bring us any closer to the goal of correct programs. A correct program is not just more reliable - it is reliable. It does not just rarely go wrong - it cannot go wrong. The correct program should be the philosophers stone for the programmer, the pole star of our efforts. Software engineering may allow the intellectual effort of the programmer to be used "more efficiently" however it does not necessarily give us accurate programs.

Away from testing

Testing is usually regarded as an important stage of the software development cycle. Testing will never be a substitute for reasoning. Testing may not be used as evidence of correctness for any but the most trivial of programs. Software engineers some times refer to "exhaustive" testing when in fact they mean "exhausting" testing. Tests are almost never exhaustive. Having lots of tests which give the right results may be reassuring but it can never be convincing. Rather than relying on testing we should be relying in reasoning. We should be relying on arguments which can convince the reader using logic.

The benefits and costs of correct programs

If correct programs were cheap and easy then we would all use them. In fact the intellectual effort involved in proving the correctness of even the simplest of programs is immense. However the potential benefits of a cast iron guarantee on a program would be attractive in many situations. Certainly in the field of "safety-critical" systems formal methods may have a role to play. It must however be admitted that the safety of many such systems cannot be ensured by software - no amount of mathematics is going to make a weapons system or a complex chemical plant safe. Formal methods may have a useful part to play in systems where there is a high cost of failure - examples such as power stations, air traffic control and military systems come to mind. The cost of failure in any of these cases may be in terms of human life. The really important market for such systems is in fact in financial systems where the cost of failure is money itself.

Why functional programming

Functional languages such as ML, Hope and Lisp allow us to develop programs which will submit logical analysis relatively easily. Using a functional language we can make assertions about programs and prove these assertions to be correct. It is possible to do the same for traditional, imperative programs - just much harder. It is also possible to write programs in ML which defy logic - just much harder. A functional language like ML offers all of the features that we have come to expect from a modern programming language. Objects may be packaged with details hidden. Input and output tend to be rather more primitive then we might expect, however there are packages which allow ML to interface with front ends such as X-windows.

Functional languages are particularly well suited to parallel processing - several research projects have demonstrated superior performance on parallel machines.

Summary

We compare Formal Methods and Functional Programming with some traditional Imperative Programming and traditional software engineering:

	Imperative Programming & Traditional Software Engineering	**Functional Programming & Formal Methods**
The	Using informal language a	Using logic we can state the

Development Cycle	specification may be open to interpretation. Using appropriate testing strategies we can improve confidence - but not in any measurable way. Mistakes/bugs are common and difficult to spot and correct.	specification exactly. Using mathematics we may be able to prove useful properties of our programs. Mistakes/bugs are common and difficult to spot and correct.
The Development Language	Using structured programming or object oriented techniques we can reuse code. Using structured programming or object orientation we can partition the problem into more manageable chunks.	Using structured programming or object oriented techniques we can reuse code. We can partition the problem into easy to use chunks - plus there are often "higher-level" abstractions which can be made ML which would be difficult or impossible in a traditional language.
The Run-time System	The compiler can produce fast compact code taking a fixed amount of memory. Parallel processing is not possible (in general). Fancy GUI's may be added.	The memory requirements are large and unpredicatable. Parallel processing is possible. Fancy GUI's may be added, with difficulty.

Introduction to ML

This is aimed at students with some programming skills, but new to functional languages. It consists almost entirely of exercises and diversions; these are intended to be completed at a machine with at least some supervision. It is not intended to replace teaching. It will most likely be possible to copy text from the hyper text viewer (possibly Netscape or Mosaic) and paste it directly into a window in which ML is running thus saving at least some re-typing.

Learning

This document is an attempt to guide the student in learning rather than to present the syntax and theory in an ordered fashion. A considerable amount of time must be invested in learning a new language; with ML it's worth it.

"Hello world"

All of the following tutorial material has been developed for Standard ML. It has been used with New Jersey ML and Edinburgh ML but should work with any other version. The ML prompt is "-". Expressions typed in are immediately evaluated and usually

displayed together with the resulting type. Expressions are terminated with ";" Using New Jersey ML the following dialogue might take place:

```
- "Hello World";
val it = "Hello World" : string
```

When used normally the ML accepts expressions and evaluates them. The result is printed to the screen together with the type of the result. The last result calculated may be referred to as `it`. In the example above the interpreter does not have to do any work to calculate the value of the expression entered - the expression is already in its simplest - or normal form. A trickier example would be the expression 3+4 this is evaluated to the value 7.

```
- 3+4;
it = 7 : int
```

Notice that the expression to be evaluated is terminated by a semicolon. The interpreter allows expressions to go over more than one line. Where this happens the prompt changes to "=" for example:

```
- 4 + 4 +
= 4;
val it = 12 : int
```

Defining functions

A function may be defined using the keyword fun. Simple function definitions take the form:

```
fun   = ;
```

For example

```
fun double x = 2*x;
fun inc x = x+1;
fun adda s = s ^ "a";
```

These functions may be entered as above. To execute a function simply give the function name followed by the actual argument. For example:

```
double 6;
inc 100;
adda "tub";
```

The system should give you the values `12: int` and `101 : int` and `"tuba" : string` for the expressions above.

Types

The basic types available are integer, real, string, char, boolean. From these we can construct objects using tuples, lists, functions and records, we can also create our own base types - more of this later. A tuple is a sequence of objects of mixed type. Some tuples:

```
(2,"Andrew")    : int * string
(true,3.5,"x")  : bool * real * string
((4,2),(7,3))   : (int * int) * (int * int)
```

While a tuple allows its components to be of mixed type and is of fixed length, a list must have identically typed components and may be of any length. Some lists:

```
[1,2,3]                 : int list
["Andrew","Ben"]        : string list
```

```
[(2,3),(2,2),(9,1)]    : (int * int) list
[[],[1],[1,2]]         : int list list
```
Note that the objects `[1,2]` and `[1,2,3]` have the same type `int list` but the objects `(1,2)` and `(1,2,3)` are of different types, `int*int` and `int*int*int` respectively. It is important to notice the types of objects and be aware of the restrictions. While you are learning ML most of your mistakes are likely to get caught by the type checking mechanism.

Polymorphism

Polymorphism allows us to write generic functions - it means that the types need not be fixed. Consider the function length which returns the length of a list. This is a pre-defined function. Obviously it does not matter if we are finding the length of a list of integers or strings or anything. The type of this function is thus
```
length : 'a list -> int
```
the type *variable* 'a can stand for any ML type.

Bindings

A binding allows us to refer to an item as a symbolic name. Note that a label is not the same thing as a variable in a 3rd generation language. The key word to create a binding is val. The binding becomes part of the environment. During a typical ML session you will create bindings thus enriching the global environment and evaluate expressions. If you enter an expression without binding it the interpreter binds the resulting value to it.
```
- val a = 12;
val a = 12 : int
- 15 + a;
val it = 27 : int
```

Pattern Matching

Unlike most other languages ML allows the left hand side of an assignment to be a structure. ML "looks into" the structure and makes the appropriate binding.
```
- val (d,e) = (2,"two");
val d = 2 : int
val e = "two" : string
- val [one,two,three] = [1,2,3];
std_in:0.0-0.0 Warning: binding not exhaustive
                one :: two :: three :: nil = ...
val one = 1 : int
val two = 2 : int
val three = 3 : int
```
Note that the second series of bindings does succeed despite the dire sounding warning - the meaning of the warning may become clear later.

Lists

The list is a phenomenally useful data structure. A list in ML is like a linked list in C or PASCAL but without the excruciating complexities of pointers. A list is a sequence of items of the same type. There are two list constructors, the empty list nil and the cons

operator ::. The nil constructor is the list containing nothing, the :: operator takes an item on the left and a list on the right to give a list one longer than the original. Examples

```
nil                     []
1::nil                  [1]
2::(1::nil)             [2,1]
3::(2::(1::nil))        [3,2,1]
```

In fact the cons operator is right associative and so the brackets are not required. We can write 3::2::1::nil for [3, 2, 1]. Notice how :: is always between an item and a list. The operator :: can be used to add a single item to the head of a list. The operator @ is used to append two lists together. It is a common mistake to confuse an item with a list containing a single item. For example to obtain the list starting with 4 followed by [5,6,7] we may write 4::[5,6,7] or [4]@[5,6,7] however 4@[5,6,7] or [4]::[5,6,7] both break the type rules.

```
::      : 'a * 'a list -> 'a list
nil     : 'a list
```

To put 4 at the back of the list [5,6,7] we might try [5,6,7]::4 however this breaks the type rules in both the first and the second parameter. We must use the expression [5,6,7]@[4] to get [5,6,7,4]

Curry

A function of more than one argument may be implemented as a function of a tuple or a "curried" function. (After H B Curry). Consider the function to add two integers Using tuples

```
- fun add(x,y)= x+y : int;
val add = fn int * int -> int
```

The input to this function is an `int*int` pair. The Curried version of this function is defined without the brackets or comma:

```
- fun add x y = x+y : int;
val add = fn : int -> int -> int
```

The type of this function is `int->(int->int)`. It is a function which takes an integer and returns a function from an integer to an integer. We can give both arguments without using a tuple

```
- add 2 3;
it = 5 : int
```

Giving one argument results in a "partial evaluation" of the function. For example, applying the function add to the number 2 alone results in a function which adds two to its input:

```
- add 2;
it = fn int-> int
- it 3;
it = 5 : int
```

Curried functions can be useful - particularly when supplying function as parameters to other functions.

Pattern Matching

In the examples so far we have been able to define functions using a single equation. If we need a function which responds to different input we would use the if _ then _ else structure or a case statement in a traditional language. We may use if then else in ML however pattern matching is preferred. Example: To change a verb from present to past tense we usually add "ed" as a suffix. The function past does this.

```
past "clean" = "cleaned"      past "polish" = "polished"
```

There are irregular verbs which must be treated as special cases such as run -> ran.

```
fun past "run"  = "ran"
|   past "swim" = "swam"
|   past x      = x ^ "ed";
```

When a function call is evaluated the system attempts to match the input (the actual parameter) with each equation in turn. Thus the call past "swim" is matched at the second attempt. The final equation has the free variable x as the formal parameter - this will match with any string not caught by the previous equations. In evaluating `past "stretch"` ML will fail to match the first two equations - on reaching the last equation `x` is temporarily bound to `"stretch"` and the right hand side, `x^"ed"` becomes `"stretch"^"ed"` evaluated to `"stretched"`.

In the following examples we use exactly two patterns for our functions. The first pattern is the base case which is typically `0` or `1` the second is `n` which matches with all other numbers.
A typical function takes the form:

`fun f(0) = ??` The equation used when the input is zero

`| f(n) = ??` The equation used when n is 1 or 2 or 3 ...

More on pattern matching later....

Recursion

Using recursive functions we can achieve the sort of results which would require loops in a traditional language. Recursive functions tend to be much shorter and clearer. A recursive function is one which calls itself either directly or indirectly. Traditionally, the first recursive function considered is factorial.

```
n       n!      Calculated as
0       1
1       1*0!    = 1*1   = 1
2       2*1!    = 2*1   = 2
3       3*2!    = 3*2   = 6
4       4*3!    = 4*6   = 24
5       5*4!    = 5*24  = 120
6       6*5!    = 6*120 = 720
7       7*6!    = 7*720 = 5040
...
12      12*11*10*..2*1  = 479001600
```

A mathematician might define factorial as follows
0! = 1

n! = n.(n-1)! for n>0
Using the prefix factorial in place of the postfix ! and using * for multiplication we have
```
fun factorial 0 = 1
|   factorial n = n * factorial(n-1);
```
This agrees with the definition and also serves as an implementation. To see how this works consider the execution of `factorial 3`. As `3` cannot be matched with `0` the second equation is used and we bind `n` to `3` resulting in
```
factorial 3 = 3 * factorial(3-1) = 3*factorial(2)
```
This generates a further call to factorial before the multiplication can take place. In evaluating `factorial 2` the second equation is used but this time `n` is bound to `2`.
```
factorial 2 = 2 * factorial(2-1) = 2*factorial(1)
```
Similarly this generates the call
```
factorial 1 = 1 * factorial 0
```
The expression `factorial 0` is dealt with by the first equation - it returns the value `1`. We can now "unwind" the recursion.
```
factorial 0    = 1
factorial 1    = 1 * factorial 0      = 1*1   = 1
factorial 2    = 2 * factorial 1      = 2*1   = 2
factorial 3    = 3 * factorial 2      = 3*2   = 6
```
Note that in practice, execution of this function requires stack space for each call and so in terms of memory use the execution of a recursive program is less efficient than a corresponding iterative program.

Take care

It is very easy to write a non-terminating recursive function. Consider what happens if we attempt to execute `factorial ~1` (the tilde ~ is used as unary minus). To stop a non terminating function press control C. Be warned that some functions consume processing time and memory at a frightening rate. Do not execute the function:
```
fun bad x = (bad x)^(bad x);
```

List processing and pattern matching

sum of a list

Consider the function `sum` which adds all the elements of a list.
```
sum [2,3,1] = 2 + 3 + 1 = 6
```
There are two basic patterns for a list - that is there are two list constructors, :: and nil. The symbol :: is called cons, it has two components, nil is the empty list We can write equations for each of these constructors with completely general components. The empty list is easy - sum of all the elements in the empty list is zero.
```
        sum nil   = 0
```
In the cons case we need to consider the value of `sum(h::t)`. Where `h` is the head of the list - in this case an integer - and t is the tail of the list - i.e. the rest of the list. In constructing recursive functions we can assume that the function works for a case which is in some sense "simpler" than the original. This leap of faith becomes easier with

practice. In this case we can assume that function sum works for t. We can use the value `sum t` on the right hand side of the definition.

```
        sum(h::t) = ??? sum(t);
```

We are looking for an expression which is equal to `sum(h::t)` and we may use `sum t` in that expression. Clearly the difference between `sum(h::t)` and `sum(t)` is `h`. That is, to get from `sum(t)` to `sum(h::t)` simply add `h`

```
fun     sum nil   = 0
|       sum(h::t) = h + sum t;
```

appending (joining) two lists

The infix append function @ is already defined however we may derive its definition as follows The append operator joins two lists, for example

```
[1,2,3] @ [4,5,6] = [1,2,3,4,5,6]
```

The definition of an infix operator allows the left hand side to be written in infix. Given two parameters we have a choice when it comes to deciding how to recurse. If we choose to recurse on the second parameter the equations will be

```
fun     x @ nil    = ??
|       x @ (h::t) = ??;
```

It turns out that this does not lead to a useful definition - we need to recurse on the first parameter, giving

```
fun        nil @ x = ??
|       (h::t) @ x = ??;
```

The first equation is easy, if we append nil to the front of x we just get x. The second equation is more difficult. The list h::t is to be put to the front of x. The result of this is h cons'ed onto the list made up of t and x. The resulting list will have h at the head followed by t joined onto x. We make use of the @ operator within its own definition.

```
fun        nil @ x = x
|       (h::t) @ x = h::(t @ x);
```

Of course the @ operator is already defined. Note that the actual definition used is slightly different. Example: doublist Consider the function doublist which takes a list and doubles every element of it.

```
doublist [5,3,1] = [10,6,2]
```

Again we consider the two patterns nil and (h::t). The base case is nil

```
        doublist nil   = nil
```

A common mistake is to think doublist nil is 0. Just by looking at the type we can see that this would be nonsense. The output from doublist must be a list, not an integer. In considering the cons case an example may help. Imagine the execution of a particular list say doublist [5,3,1]. We rewrite [5,3,1] as 5::[3,1] and consider the second equation.

```
        doublist(5::[3,1]) = ??? doublist [3,1]
```

Thanks to our faith in recursion we know that doublist[3,1] is in fact [6,2] and so we ask what do we do to [6,2] to get our required answer [10,6,2]. We answer "stick ten on the front".

```
        doublist(5::[3,1]) = 10::doublist [3,1]
```

Returning to the general case with h and t instead of 5 and [3,1]:

```
        doublist(h::t) = 2*h :: doublist t
```

if .. then .. else ..

Sometimes pattern matching is not convenient. We may wish to compare values for example, in these cases the `if .. then .. else ..` structure is useful.

The expression `if` *`B`* `then` *`S1`* `else` *`S2`* tests the boolean expression *`B`*, it returns the value of *`S1`* or the value of *`S2`* depending on the value of *`B`*.

For example

```
if 1 = 0 then "I am the pope." else "someone else is the pope.";
```
Returns the string `"someone else is the pope."`

The following function "tells us" about a string s. A palimdrome is a word which is the same backwards as forwards.

```
fun pali s = if explode s = rev(explode s) then s ^ " is a palindrome."
                                          else s ^ " is not a palindrome.";
```
We can go further - the sentence is the same in both cases, except the substring "not " is missing in one case - this allows..
```
fun pal2 s = s^" is "^(if explode s = rev(explode s) then "" else "not
") ^
                        "a palindrome.";
```
In some languages the `else` part is optional - that would make no sense in ML as the expression must return a value.

The @ operator

The append operator is defined in the file "/usr/local/software/nj-sml-93/src/boot/perv.sml" and is given as:
```
  infixr 5 :: @
  fun op @(x,nil) = x
    | op @(x,l) =
    let fun f(nil,l) = l
          | f([a],l) = a::l
        | f([a,b],l) = a::b::l
        | f([a,b,c],l) = a::b::c::l
        | f(a::b::c::d::r,l) = a::b::c::d::f(r,l)
     in f(x,l)

    end
```
This version may be shown to be equivalent to the simpler:
```
  infixr 5 :: @
  fun nil   @ l = l
  |   (h::t)@ l = h::(t@l)
```
but it will run faster.

Pattern matching and recursion

When defining a function over a list we commonly use the two patterns

```
fun     lfun nil        = ...
|       lfun(h::t)      = ... lfun t ...;
```

However this need not always be the case. Consider the function last, which returns the last element of a list.

```
last [4,2,5,1] = 1
last ["sydney","beijeng","manchester"] = "manchester"
```

The two patterns do not apply in this case. Consider the value of last nil. What is the last element of the empty list? This is not a simple question like "what is the product of an empty list". The expression last nil has no sensible value and so we may leave it undefined. Instead of having the list of length zero as base case we start at the list of length one. This is the pattern [h], it matches any list containing exactly one item.

```
fun     last [h]        = h
|       last(h::t)      = last t;
```

This function has two novel features.

Incompleteness

When we enter the function as above ML responds with a warning such as

```
std_in:217.1-218.23 Warning: match non exhaustive
        h :: nil => ...
        h :: t => ...
```

The function still works, however ML is warning us that the function has not been defined for all values, we have missed a pattern - namely nil. The expression last nil is well-formed (that is it obeys the type rules) however we have no definition for it. It is an incomplete or partial function as opposed to the complete or total functions that we have seen thus far. You will naturally want to know how ML does treat the expression last nil. The warning given is a mixed blessing. Under certain circumstances a partial function is very useful and there is no merit in making the function total. However if we manage to compile a program with no warnings and avoid all partial functions we are (almost) guaranteed no run-time errors. The exhaustive checking of input patterns can be non-trivial, in fact the algorithm which is used in non polynomial.

Overlapping left hand sides

As the pattern [h] is identical to the pattern h::nil we might rewrite the definition

```
fun     last(h::nil) = h
|       last(h::t)   = last t;
```

Examining the patterns of the left hand side of the = we note that there is an overlap. An expression such as 5::nil will match with both the first equation (binding h to 5) and the second equation (binding h to 5 and t to nil). Clearly it is the first line which we want and indeed ML will always attempt to match with patterns in the order that they appear. Note that this is not really a novel feature as all of our first examples with the patterns x and 0 had overlapping left hand sides.

Conditions

Where possible we use pattern matching to deal with conditions, in some cases this is not possible. We return to the function to convert present to past tense. The general rule - that we append "ed" does not apply if the last letter of the verb is "e". We can examine the last character of the input by applying explode then rev then hd. The improved version of past should give

```
past "turn" = "turned"
past "insert" = inserted"
past "change" = "changed"
```

The special case irregular verbs are dealt with as before:

```
fun     past "run" = "ran"
|       past "swim" = "swam"
|       past x = if hd(rev(explode x))="e" then x^"d"
                        else x^"ed";
```