Lecture Notes, October 1989

A. C. Norman.

1 Introduction

2 Course objectives, prerequisites and outline

This course introduces the Computer Science option within the Mathematics Tripos and the first year of the Computer Science Tripos. It leads on towards parts IB and II of the Computer Science Tripos. Its main aim is to prepare the ground for the more detailed material that follows by teaching a collection of ideas about Computer Science and a variety of programming skills that can be used later on in the CST. It is understood that different members of the audience will have had very different amounts of contact with computers: some will have had hardly any, while others will already be accomplished programmers. There are two reasons why the novices need not feel intimidated by this course while the experts will still be challenged. The first is that the programming language used here is called ML, which is fairly new and which will almost certainly be strongly unfamiliar to all. The second is that this course is not principally about writing programs for the sake of running them on computers (that comes later!) - it is much more about the use of programming languages as a way of providing clear (to people) expositions of the various ways of performing computational tasks.

The structure of this course follows the book "Structure and Interpretation of Computer Programs" (Abelson and Sussman, MIT Press, 1985, ISBN 0-262-01077-1/0-07-000-422-6) fairly exactly, and in the sixteen lectures about the first half of that book will be covered. Abelson and Sussman express themselves in a language called Scheme, not the ML that is being used here, and so all the examples that they give will need superficial re- writing to put them in the notation required by ML. Although this is a slight inconvenience, it gives me a chance to stress here that the view taken in this course is that the fine details of syntax in programming languages are unimportant frivolities. Language constructs in ML will be introduced by giving examples, and the practical classes will provide opportunities to try them out, but it is not the purpose of this course to provide either a full or a precise description of what the particular language ML is or does. Two technical reports describing ML is available from the Computer Laboratory Bookshop, and a book describing it has been published: A Wikstrom, "Functional

Programming using Standard ML", Prentice Hall, 1987, ISBN 0-13-331661-0. Both the technical reports and Wikstrom's book provide formal descriptions of the ML language, and they will certainly be useful for reference, although the topics covered and the style used in Wikstrom differ significantly from this course.

The parts of ML required initially correspond to using the system as a glorified desk calculator. In response to ML's prompt you can type in an expression, terminated with a semicolon, and ML will respond by calculating and displaying its value.

e.g.

```
2+2;
22.0/7.0;
22 div 7;
~1;
```

Whenever ML evaluates an expression it works out what type of value is being used - in the above examples the whole numbers are typed as 'int' and the ones with a decimal point as 'real'. Observe that division in the two cases will behave differently, and so different operators are used to indicate it. Also note negation is indicated with a ' ' sign, rather than by using a '-' to mean both the difference between two numbers and the negation of a single one, as is common in other programming languages. ML also knows about strings. To get a string that has a double- quote mark as one of its characters you put a backslash (
) before it. Strings can be appended using the operator '^', thus

```
"String one " ^ "string two!";
"string with \"what looks like a string\" within itself";
```

There are also truth values, which are the values returned by arithmetic (and other) comparisons, and which will be needed when tests are to be made:

```
true;                             true : bool
not true;                         false : bool
2 > 3 orelse 5 < 7;               true : bool
```

The connective words for building elaborate conditional expressions are 'andalso' and 'orelse'.

At this level ML's insistence on decorating results with a description of their type can be seen as fairly frivolous, but it is in fact the externally visible manifestation of some complicated type-consistency checking that ML is capable of which will only just be touched on in this course, but which will be returned to later in the CST. It corresponds to widely held views within Computer Science first that the organisation of data is at least as important as that of code, and secondly that analysis of code (such as is done by ML when it deduces the types of

your expressions) is important for code comprehensibility, reliability and performance. ML will complain if you mix types up in ways it is not prepared for, and so expressions such as (not 1), (1 - 1.0), exp(”one”) and so on will be rejected by it.

Names can be associated with ML values by a statement such as

```
val pi_approx = 355.0 / 113.0;
val heading = "The start of my program";
```

and the names thus introduced can be used freely in ML expressions:

```
pi_approx - 4.0 * arctan(1.0);
val subheading = heading ^ " number 1";
```

but a characteristic feature of ML (and of the parts of Scheme used in Abelson and Sussman) is that associations between names and values set up in this way can not be changed later in the program. [Note that this is very different from the use of assignment statements in languages such as BASIC, where it is generally expected that names will have many different values associated with them during the course of running a program].

2.1 Exercises

(1.1) Find out how to connect yourself to a computer and start the ML system running. Try typing in simple arithmetic expressions, using ML as an overexpensive and clumsy desk calculator. Find out how to exit from ML and disconnect yourself from the computer. Find the Computing Service bookshop and but a copy of the technical report that contains an introduction to ML.

(1.2) Check that you can send and receive computer-mail from your supervisor and some of the other people on the course so that you have a convenient way of asking for help when you get stuck. Investigate the ‘help’ system on the computer.

(1.3) Type in various expressions for ML in integers, real values, boolean values and strings. Observe how ML replies with the value and type of the expression that you enter. See what sort of error messages you get when you mix up the types.

(1.4) Observe that some terminals have printers attached. Check that you can enable one of these to get a transcript of exactly what you typed in a session with ML, and exactly what it responded. This will be invaluable when you go for help!

(1.5) Double-check the schedule of practical classes associated with this course, and try examples suggested there.

3 Building abstractions with procedures

3.1 Procedures

The way so far explained for introducing names for values in ML is global, in that after the assertion val ¡name¿ = ¡value¿ the name ¡name¿ persists until then end of the ML session. This is frequently unnecessarily long, and so there is provision for making shorter term declarations:

```
let
   val <name1> = <value1>;
   val <name2> = <value2>
in
   <expression using name1, name2>
end;
```

E.g. to compute the fourth power of two one could write

```
let val two_squared = 2*2
in two_squared*two_squared end;
```

and the name two_squared would not be at all visible outside the two lines where it is required. Local declarations can temporarily shadow previous global (or indeed local) ones, but the old binding becomes visible once the local one is finished with. Thus

```
val one = 1;
let val one = 2
in one + one end;                   4 : int
one + one;                          2 : int
```

Almost all of this course will be based on a view of programming style that the secret service would have great sympathy with: no definition or value should be available anywhere in a program that it is not strictly needed. This 'need-to-know' principle can be applied by preferring local ML bindings (with let) to global ones, and keeping the scope of these let clauses as small as possible. I hope that the demonstrators at the practical classes will help encourage you to follow this line...

Procedure declarations also introduce local bindings; functions can be defined as in

```
fun square x:int = x * x;
fun sumsquares(x:int, y:int) =
   square x + square y;
```

and then used

```
square 1 + square 2 + square 3;                    14 : int
sumsquares(3, 4);                                  25 : int
```

The decorations ':int' after the names of the formal parameters for the function limit the types that they can accept, so that in the cases given above integer (but not for instance real) numbers can be used. These explicit type constraints are not needed in cases when ML can deduce what was required, but it is always safe to put them in.

ML displays the type of a function with a description of its arguments' types (if there are several it puts '*'s between them), then an arrow (-¿) then the result type, so in the above example square is of type int-¿int and sumsquares is of type int*int-¿int. Observe that functions with just one argument can be applied by just writing the function name followed by the argument (parentheses around the argument are not required, but you can put them in if you like: sin 1.0 and sin(1.0) have exactly the same interpretation). The 'formal parameters' given in the function definition behave exactly like local variables when the function is called - for the duration of evaluating the body of the function (the part after the '=') they exist and stand for the arguments passed by the function's caller, after that they can not be accessed.

The effect of calling a procedure can be explained in terms of a substitution model. The value of the sumsquares(7+1,7-1) can be obtained by first evaluating the argument expressions to obtain 8 and 6, then looking at the body of sumsquares, i.e.

```
square x + square y
```

When we substitute 8 for x and 6 for y in this (x and y were the formal parameter names for sumsquares) we obtain

```
square 8 + square 6
```

Now the body of square is x * x, and since x was its formal parameter we substitute into this expression to obtain

```
8*8 + 6*6
```

which reduces gradually as

```
64 + 36
100 : int
```

In this description in each case the arguments for a function were evaluated before the function was called, but it is quite possible to carry through the substitution process by substituting argument expressions into function bodies before they have been fully evaluated. In this case sumsquares(7+1,7-1) would have expanded into

```
square (7+1) + square (7-1)
```

and thence into

```
(7+1)*(7+1) + (7-1)*(7-1)
```

which would finally have been evaluated arithmetically, again to yield the integer answer 100. A question that can be raised now (but only properly answered a lot later) is what effect can the order in which substitutions are selected have on the result finally produced? A particular strength of the subset of ML that we are using at present (and of the equivalent part of Scheme) is that provided some pitfalls related to name-clashes are side- stepped all evaluation orders that lead to a result at all lead to the same result, and so when working through function definitions by hand it the expansion can be done in whatever order is most convenient. It is perhaps worth noting that this result is quite unexpectedly difficult to prove, and indeed several incorrect proofs were put published before a correct one was found: the details are another of the topics that you can look forward to (much) later in the CST.

Most interesting procedures involve behaviour that depends in a less smoothly analytic way on inputs than the above one, and the ML way of coping with this is to make the body of the function a conditional expression:

```
fun abs n = if n < 0 then ~n else n;
```

[In this case the comparison between n and 0 would only be valid when n was an integer, so it is not necessary to write ‘n:int’.]

Of course conditional expressions can be used anywhere that any other sort of expression can - this fact has to be allowed for when tracing out the sequence of substitutions involving use of a function such as abs.

```
(if 2 > 3 then 1 else ~1) + 7;             6 : int;
```

It is essential that the expressions before and after the ‘else’ in a conditional form both have the same type, an in particular this means that both must always exist!

Now conditional expressions and function definitions can be brought together to perform some useful(?) calculation, in this case the extraction of the square root of a number. The code given here follows that in Abelson and Sussman fairly directly, but for use with ML it turns out to be vital to define all functions before making any attempt to use them, so the order in which definitions are given here is almost precisely the opposite to that used in the book. On paper or using a text-editor on the computer it is easy to write code starting at the bottom of the page, so this does not matter too much! Note that for this example versions of square and abs that work with real arguments will be required:

```
fun abs x = if x < 0.0 then ~x else x;
fun square x:real = x * x;
fun average(a, b) = (a + b) / 2.0;
fun improve(guess, x) =
    average(guess, x/guess);
fun good_enough(guess, x) =
    abs(square guess - x) < 0.001;
fun sqrt_iter(guess, x) =
    if good_enough(guess, x) then guess
    else sqrt_iter(improve(guess, x), x);
fun sqrt x = sqrt_iter(1.0, x);
sqrt 2.0;                       1.414...  : real
```

There are several things that I might like to observe about this sequence of statements:

1. The definition of (square (x:real)) can not co-exist with that of (square (x:int)). In ML only built-in operators such as +, - and * (and a special built in function abs) exist in versions for both reals and ints.

2. Most of the above function definitions did not need the formal parameters decorated with type information: ML could deduce what was intended. This very convenient facility is paid for in the way ML forces you to define procedures before using them.

3. The test good_enough shown above is rotten numerical analysis, as is the fixed initial guess of 1.0. More cautious code would also detect and complain about negative arguments to sqrt. A much better termination condition to apply would be to detect when the error (square guess - x) stops decreasing: please rework my code to install this improved test.

4. When you have bodies of code like the above it is usually convenient to keep them on a file on the computer and read it into ML, rather than having to type it in to the computer each time you want to use it. The practical classes should provide some help about creating, editing and reading in files of definitions. An ML function call

   ```
   use ["filename"];
   ```

   reads in definitions and tests from the named file (yes the brackets used are square in this case).

5. In the above, sqrt_iter calls itself, but the other functions can be introduced in a neat hierarchy. When several function need to refer mutually to one

another they can be defined in a group, as with the following nonsense example:

```
fun f x = g(x + 1)
and g x = if x > 10 then x else f(x+1);
```

6. The code is presented in lots of very tiny functions, but each function serves a (more or less) clearly distinct purpose. This style is again part of the programming strategy I want to encourage. Last time I talked about the 'need-to-know' as a secret service policy - here I will go to the other end of the political spectrum and view each separate function an active cell in a (subversive?) organisation - keeping cells small and with clearly limited links between them limits the damage sustained when one of them is found to have become unreliable.

The square root example is expressed in terms of a fairly large number of sub-functions, but each of these performs a sufficiently clearly specified task that it is useful to think not just about the mechanisation (i.e. programming) of these procedures, but about the abstractions that they represent. Supposing that its numerical accuracy is adequate, the sqrt function we have defined ourselves (thereby hiding the one that ML originally provided) is just as valid as the built-in ML one, and a further one defined by

```
fun sqrt x = exp(0.5 * log x);
```

is also perfectly proper. When looking at sqrt as a procedural abstraction it is only possible to ask about its behaviour, not about what its internal actions are that lead to that external behaviour. This leads to the idea that what happens inside and what happens outside a procedure should be kept as separate as possible. The example given above does not capture the essence of this aim, in that good_enough and improve are available for direct use by the misguided. Energetic use of 'let' within procedure bodies remedies that, leading to the following version of the original code, which although superficially harder to read captures more of the proper intent of the programmer.

```
fun sqrt x =
   let
      fun improve(guess, x) =
         average(guess, x/guess);
      fun good_enough(guess, x) =
         abs(square guess - x) < 0.001;
      fun sqrt_iter(guess, x) =
         if good_enough(guess, x) then guess
         else sqrt_iter(improve(guess, x), x)
   in sqrt_iter(1.0, x) end;
```

With this definition of sqrt in place the internal functions improve, good_enough and sqrt_iter are kept hidden away from the end user. A formal parameter can be referred to anywhere inside a function, and this includes within sub-functions, and a yet cleaner version of the sqrt function can be given by exploiting this.

```
fun sqrt x =
   let
      fun improve guess =
          average(guess, x/guess);
      fun good_enough guess =
         abs(square guess - x) < 0.001;
      fun sqrt_iter guess =
         if good_enough(guess, x) then guess
         else sqrt_iter(improve guess)
   in sqrt_iter 1.0 end;
```

[I have taken the view that abs and average are sufficiently generally useful to be properly defined for all to use, and so have assumed that they have been defined globally somewhere.]

Given an understanding of just the parts of ML introduced so far it is possible to write quite large and useful programs, and the careful use of nested local definitions can keep the structure of these programs clear and tidy.

3.2 Exercises

(2.1) Experiment with local and global 'val' definitions of variables, particularly to discover what happens if you have several declarations of variables all with the same name. For instance, what does the sequence

```
val i = 1;
val i = i + 1;
```

do?

(2.2) Find out how to create (and inspect, alter and discard) files on the computer. Put one of the above definitions of sqrt into a file and use 'use' to make ML read it.

(2.3) A much better end-test for the square-root procedure is where the error term abs(guess 2 - x) ceases to decrease. Alter the code that you have to incorporate this improved test.

(2.4) Look in Abelson and Sussman's book. Compare the syntax of Scheme with that of ML (in pretty well all cases the meaning of the example programs that they give is exactly the same as ones given here in ML). Investigate their exercises.

(2.5) Make a variety of syntax errors in your ML code (e.g. miss out the word 'end' at the end of a 'let' clause, get brackets badly matched, ...) and get accustomed to the messages that the system gives you.

(2.6) The Student Edition of 'PC Scheme' (prepared by Texas Instruments and published (1988) by the Scientific Press with ISBN 0-89426-114-2) comes complete with an implementation of the Scheme language ready to run on an IBM-style personal computer. Extreme enthusiasts might like to look into this and see if a combination of it and Abelson and Sussman provide yet more things to try! Please note, however, that lectures and examinations here are in terms of ML and that there is no departmental backup or support for PC Scheme.

(2.7) Buy and read other books from the CST booklist. Find your College Library and put in request slips for books it has missing. Find the Computer Lab book-locker and work out how to make use of it. Discover the tea room in the University Library.

3.3 Iteration and Recursion

Procedural abstraction encourages us to look at the external behaviour of functions, that is the results obtained when they are given various arguments. Looking inside procedures at their internal behaviour there are characterisations that can be made that independent of the finer detail of the operations performed. Simple examples of patterns of behaviour will be given here. Later in the Computer Science Tripos it will be seen that understanding such patterns of behaviour and selecting computational methods that lead to desirable patterns can be both very complicated and of significant practical importance.

Consider the factorial function, defined by

```
factorial n = n * (n-1) * ...  * 3 * 2 * 1;
```

which, with a little regrouping can be seen to give

```
factorial n = n * factorial(n-1);
```

This can be expressed as an ML function definition by providing a value for factorial 0, as in

```
fun factorial n =
   if n = 0 then 1 else n * factorial(n-1);
```

Applying the substitution model for procedure evaluation allows us to see the effect of a request for the value of factorial(6).

```
factorial 6
6 * factorial 5                                   (A)
6 * (5 * factorial 4)
6 * (5 * (4 * factorial 3))
6 * (5 * (4 * (3 * factorial 2)))
6 * (5 * (4 * (3 * (2 * factorial 1))))
6 * (5 * (4 * (3 * (2 * (1 * factorial 0)))))
6 * (5 * (4 * (3 * (2 * (1 * 1)))))
6 * (5 * (4 * (3 * (2 * 1))))
6 * (5 * (4 * (3 * 2)))
6 * (5 * (4 * 6))
6 * (5 * 24)
6 * 120
720
```

Now consider an alternative scheme based on the idea of keeping a running product, and multiplying it by 1, 2, 3, ... and so on up to. That leads to code of the form

```
fun factorial n
   let
      fun sub_function(i, so_far) =
         if i > n then so_far
         else sub_function(i+1, i*so_far)
      in sub_function(1, 1) end;
```

This new definition seems bulkier (and on that basis alone less desirable) than the first one: is it describing the same calculation? To find out we can trace through it using the substitution rules to yield the table

```
factorial 6
sub_function(1, 1)                                (B)
sub_function(2, 1)
sub_function(3, 2)
sub_function(4, 6)
sub_function(5, 24)
sub_function(6, 120)
sub_function(7, 720)
720
```

which is conspicuously different in layout to the previous example. The first of these behaviour patterns (A) is referred to as ‘recursion’, while (B) is known as ‘iteration’. In scheme A any computer implementation must keep track of operations that need to be performed during the unwinding of the recursion. In

this case the amount of such information that has to be kept clearly grows directly with the value of the input parameter n, and so the recursion is a linear one. In the iterative case only a fixed amount of status information is needed by the computer, even though sub_function calls itself over and over again. The ML system has the property that functions that are iterative in the sense discussed here will only consume a finite amount of space while running (other implementations of other languages may not be so well behaved!).

Another common pattern of computation is called Tree Recursion. Consider the pattern of expansion when evaluating one of the Fibonacci numbers using the program

```
fun fib n =
   if n = 0 orelse n = 1 then 1
   else fib(n-1) + fib(n-2);
```

where at each level in the calculation there are two branches that can be expanded out. The cost of running tree recursive programs can grow at an exponential rate as the depth of the tree being traversed increases, and this can lead easily lead to code that it is not feasible to execute. In general tree recursion can not be replaced by iteration, but in this case it can: the Fibonacci numbers can be computed as follows:

```
fun fib n =
   let
      fun fib_iter(fi, fj, j) =
         if j = n then fj
         else fib_iter(fj, fi+fj, j+1)
   in fib_iter(0, 1, 0) end;
```

3.4 Exercises

(3.1) Is there any noticable difference in the time taken by the various versions of the factorial function given here when run on arguments small enough that their result does not exceed the range of arithmetic that ML supports?

(3.2) Is there any noticable difference in the time taken by the versions of the Fibonacci series calculator given here?

(3.3) Find out at least something about the way in which your use of computer resources will be controlled/limited. Discover how many people share access to the computer you are using, and watch to see how heavy the pressure on the machine is at different times of day. Find out how to abort an ML calculation that you have started but that is going to take an unreasonable amount of time, or perhaps not ever complete at all.

(3.4) Devise a recursive program to compute the number of ways of giving change for a particular amount in terms of combinations of UK coins of value 1, 2, 5, 10, 20 and 50.

(3.5) Can you produce an iterative solution to problem 3.4? (3.6) Was the sqrt function defined earlier iterative or recursive?

3.5 Rates of Growth

The simple version of the Fibonacci number program exhibited a behaviour where the amount of work done grew explosively with the size of the input value, while some of the other code fragments we have seen have done amounts of work that have grown linearly with the value of the numeric input parameter. This idea of 'rate of growth' is a good abstraction to focus on. It often corresponds to a much more useful idea of 'efficiency' than anything that can be found by obtaining absolute time measurements of any particular real computer, since comparisons of rates of growth tend to remain valid despite changes in the technology of computer hardware.

The usual notation for rates of growth is the 'Big O' one. If some process does an amount of work R that depends on an input parameter n, then we say that R = O(f(n)) if f is some function so that R(n) ¡= K f(n) for some fixed constant K and all sufficiently large n. In this notation the statement that a process is O(n) indicates that its costs grow no faster than linearly with the size of an input parameter. In a similar way O(1) is a way of saying 'bounded by some fixed constant, whose value is not specified here'.

Two things are work noting about Big O notation. Firstly it allows a finite number of exceptions to the inequality that it expresses. This is helpful otherwise any O(n) process would need to have zero cost when n=0! The other is that it expresses an inequality, so anything that is O(1), for instance, is certainly also O(n) and O(n 2). It is easy to fall into a the trap of thinking that something that is described as O(f(n)) must get close to using f(n) resources for some n - that is not what the notation says: when it is true it means that the function f(n) gives a sharp bound on the costs of the process.

Big O notation can be used to describe both the amount of space needed by a computation and the number of computational steps involved. Thus the recursive factorial function given earlier used O(n) space and O(n) time, and the recursive Fibonacci number code used O(n) space and O(2 n) time, while the iterative Fibonacci code used O(1) space and O(n) time.

To illustrate how the use of this notation makes it possible to formalise (crude) estimates of computational cost, consider the problem of raising some number to an integral power. Probably the shortest and neatest code is a straightforward linear recursion (this statement is almost always true!)

```
fun expt(a, n) =                                  (A)
   if n = 0 then 1
   else a * expt(a, n-1);
```

[At a later stage in a Computer Science course (and even more if this was a course on programming rather than computer science!) I would express a lot of worry about some of the programs here - for instance what happens if expt(1, 1) is called, and what is the correct response to give to the call expt(0,0)? In this course my attention is focused elsewhere, and so I will ignore these issues. If you look at a full description of ML you will find that it provides facilities for handling exceptions - these will be ignored for now].

This procedure is O(n) in both space and time. Observe how very similar the code is to the recursive factorial program given earlier - this suggests that an iterative version will be easy to construct, and it will then yield a growth rate of O(n) in time (still), but we can tell that its space usage will be O(1).

```
fun expt(a, n) =                                  (B)
   let
      fun expt_iter(i, result) =
         if i > n then result
         else expt_iter(i+1, a*result)
   in expt_iter(1, 1) end;
```

This may be a very valuable improvement on some computers, but it has been achieved at the cost of making the code longer and probably less clear. The next advance continues this trend, and relies on the fact that

$$x^{2n} = (x^n)^2 ,$$
$$x^{2n+1} = x * x^{2n} .$$

which leads to the code

```
fun evenp n = ((n mod 2) = 0);

fun expt(a:int, n) =                              (C)
   if n = 0 then 1
   else if evenp(n) then
      square(expt(a, n div 2))
   else a * expt(a, n-1);
```

where it is supposed that square has already been defined for integer arguments. Version (C) of the expt function is recursive again, but since on at least every other successive call the parameter n gets halved, the greatest possible depth of recursion is proportional to log(n). [Note that in order of magnitude estimates such as are used here the base for logarithms is unimportant, but if it worries you

please assume logs to base 2]. Thus it can be seen that this version is O(log n) in both time and space. A version which follows the idea of squaring but which uses O(1) space (i.e. is iterative) is left as an exercise.

Raising things to large powers is a mildly eccentric occupation, but the pattern of operations used to do it can be re-used to show how to perform repeated additions to achieve the effect of multiplication. A time-efficient iterative scheme for doing this can form the basis of hardware inside a computer that supports the illusion that the computer can multiply. The patterns used in paper and pencil long multiplication may also help when devising the iterative fast exponentiation routine. In summary, for the expt procedures considered, we have

method	time	space
(A) linear recursive	O(n)	O(n)
(B) linear iterative	O(n)	O(1)
(C) recursive squaring	O(log n)	O(log n)
(D) <code not given here>	O(log n)	O(1)

Now consider a variant on method (C), where instead of calling the square function the indicated multiplication is written out explicitly

```
fun expt(a:int, n) =                              (C')
   if n = 0 then 1
   else if evenp(n) then
      expt(a, n div 2)*expt(a, n div 2)
   else a * expt(a, n-1);
```

This has turned the code into a tree recursion, and in fact the time it takes to run goes back to being O(n). This example illustrates that although in ML replacing the call of a procedure by the expanded out body of that procedure does not effect the value that is eventually produced by a computation, it can have a big effect on the amount of time taken. Here greatest efficiency was attained because ML (in effect) performed substitutions to evaluate the argument to square before investigating the body of square. In other cases (that we will see later) other strategies give the best performance.

3.6 Exercises

(4.1) Code and test an iterative O(log n) version of the function expt.

(4.2) If a and b are two integers with a ¿ b ¿ 0 then the greatest common divisor of a and b is also the greatest common divisor of b and (a mod b). Use this to produce a function that can find the greatest common divisor of any two positive

integers. Is your code recursive or iterative? Can you establish an estimate for the rate at which its computing time grows as the values of a and b do? (4.3) Investigate the various tests for prime numbers given by Abelson and Sussman, again considering rates of growth of both space and time for each method discussed.

(4.4) Try typing (on one line) val t = CpuTime(); expt(1,1000); CpuTime()-t; and similar things with different calculations in the middle. Discover if the values reported at the end behave plausibly like measures of the time taken to run the test code in the middle, and if so what units they report time in. On some computers if the above code is not all typed on one line the time the computer spends waiting for you to push buttons on the keyboard will be included in the final time reported - is this true on the system you are using?

(4.5) Define the smallest and simplest ML function that you can think of that will have time that grows linearly with the value if its argument. Time it for various argument values and see if the observed growth rate seems linear. If so, estimate the constant of proportionality relating the argument to real computer time used.

(4.6) Exploit the identities n C r = n-1 C r-1 + n-1 C r , n C 0 = n C n = 0 to define a function that calculates binomial coefficients. What sort of recursion does it use? How does its cost grow as n and r grow? Estimate the time that would be needed if it were used to computer 60 C 30 . Are there ways of improving the code?

3.7 Higher Order functions (control structure)

It was indicated last time that a single shape of code could express either the idea of exponentiation (if it multiplied) or of multiplication (if addition was the underlying step). The code involved is

```
fun expultiply(a:int, n) =
   let fun squouble(a) =
      base_operation(a, a)
   in if n = 0 then base_identity_value
      else if evenp(n) then
         squouble(expultiply(a, n div 2))
      else base_operation(a,
              expultiply(a, n-1))
   end;
```

where base_operation stands for either addition or multiplication and base_identity_value for 0 or 1. Rather than having to write this code out twice, once to implement a product function and once for exponentiation, it is possible to code it once and make base_operation and the identity value for it into parameters.

```
fun expultiply(a:int, n, opr, identity) =
   let fun squouble(a) = opr(a, a)
   in if n = 0 then identity
      else if evenp(n) then
         squouble(expultiply(a, n div 2,
                             opr, identity))
      else operation(a,
           expultiply(a, n-1, opr, identity))
   end;
```

The third argument to this function must itself be a function, and in ML this is perfectly respectable. The text written here to indicate the argument is just the name of the function that is to be passed. The following code illustrates this, defining a version of expt that uses a further layer of functions to perform the multiplication that it requires internally.

```
fun expt(a, n) =
   let fun prod(x, y) =
      let fun sum(p:int, q:int) = p + q
      in expultiply(x, y, sum, 0) end
   in expultiply(a, n, prod, 1) end;
```

A second example of an problem where functions as arguments provide a good abstraction arises when considering the derivative of simple functions. The derivative of a function f(x) at a x=a can be estimated as

```
(f(a + h) - f(a)) / h
```

for small offset values h. In ML we can capture this formula directly

```
fun deriv_approx(f, a, h:real) =
   (f(a+h) - f(a)) / h;
```

which can then be tested, for instance with

```
fun quadratic(x) = 3.0*x*x - 17.0;
deriv_approx(quadratic, 2.0, 0.001);
deriv_approx(cos, arctan(1.0), 0.001);
```

The issue to be stressed here is that permitting functions to be passed as arguments to other functions makes it possible to write one procedure (expultiply, or deriv_approx) that captures the essence of some computation, rather than copying out all the possible special instances. This clearly helps us to produce clearer procedural abstractions for the processes we are designing, and encourages us to produce code fragments that will have many uses rather than just a single use.

3.8 Exercises

(5.1) Assuming a function f(i) that returns real values and an integer n, produce a function that will form the sum f(1)+f(2)+...+f(n). Test it with f(i) the reciprocal of factorial i (this gives approximations to the number e that is the base of natural logarithms), f(i) = 1/((4i-3)(4i-1)) (which leads to pi), and f(i) = 1/i. Note the existence of useful functions in ML: real(n:int) returns a real number with value the same as the integer n, floor(x:real) returns an integer value no larger in value than the real number x.

(5.2) Generalise your summing function so that it accumulates values using an operator and an initial value that are handed to it as arguments. Demonstrate it with a call something like fun prod(a:real, b:real) = a*b; accumulate(50, real, prod, 1.0); to compute factorial 50 using real arithmetic;

(5.3) Collect some of the more general and potentially re-usable functions that you have written into a single file, so that it can be loaded at the start of future sessions with 'use'. Comments in ML are written as (* text *), so you might reasonably document your private library, explaining what each function does.

(5.4) Suppose that the function of (5.1) is being used to sum series to n terms as a way of evaluating approximations to useful values (Abramowitz and Stegun's big book of mathematical tables can provide lots and lots of example series to try summing...). The partial sums of a sequence taking in 1, 2, 3, ... terms form a sequence which it is hoped will converge towards some limit - unfortunately quite often this convergence is slow. If s n-2 , s n-1 , s n are three consecutive values in such a sequence, define S n as s n - (s n - s n-1) 2 / (s n - 2s n-1 + s n-2). Then the sequence S n often converges much faster than the original one. Implement a higher order function that can perform this transformation, and try it on (among other things) the sequence 1 - 1/3 + 1/5 - 1/7 + 1/9 which converges to pi/4. The transformation works by assuming that the sequence concerned is really a geometric progression of the form s n = p + q*k n with limit p: it uses the three sample values from the sequence to allow it to find p, q and k. What happens if you try to accelerate convergence yet further by applying the transformation twice?

(5.5) Exchange your version of the accumulate function with somebody else, and read their code. If # denotes the operator being used, do they compute f(1) # (f(2) # (f(3) # ...)))) or ((((f(1) # f(2)) # f(3) # ... or possibly something different from both the above. Starting from their code produce an accumulate function that groups things the other way. Give a test use of the two versions of accumulate that illustrates that they can sometimes give different effects. In what cases can it be guaranteed that they will behave indistinguishably?

(5.6) Produce yet another version of accumulate that splits the range from 1 to n into two roughly equal sub-ranges and accumulates values from each sub-

range before combining these to produce a final value. Characterise the space and time requirements of this function. Although from an ideal mathematical point of view all your versions of an accumulator function will be equivalent, when using computer real arithmetic (i.e. floating point) there will all give slightly different answers: investigate!

3.9 Higher Order functions (as values)

If functions can be passed as arguments, can they be returned as results? If so, does that enrich our programming style by allowing further separation between the 'what' and the 'how' of typical procedures? In ML it is possible to return functions as results, and although many older languages do not support this (or put peculiar limitations on how it may be used) it can provide good ways of modelling behaviours. Looking back to the code previously given for estimating the derivative of a function. With minor adaptation it can accept a function as an argument and deliver its derivative (again as a function) as its result. For simplicity here it is coded with a fixed offset of h=0.001

```
fun differentiate f
   let val h = 0.001;
      fun df(a) = (f(a+h)-f(a))/h
   in df end;
```

and now this can be used to compute functions that are to be applied, such as

```
(differentiate cos)(1.0) + sin(1.0);
(differentiate (differentiate sin))(0.0);
```

The point to be made here is that defining differentiate in this way puts all the information and mechanism about differentiating inside the procedure, and makes it possible to use it in a general way.

When functions are being passed backwards and forwards it is useful to be able to write one without having to invent a name for it. For instance in using expultiply it was necessary to have a function that added (or multiplied) two integers, and in the code shown earlier this was named using a 'fun' definition. An alternative would have been to use an 'anonymous function'. These are written

```
fn <formal arguments> => <body>
```

as in

```
     expultiply(2, 10, fn(x:int,y:int)=>x*y, 1);
or   (derivative (fn x => 3.0*x*x-17.0))(2.0);
```

[There are some who suggest that ML was wrongminded in using the keywords 'fn' and 'fun' for the two different ways of introducing functions, and they tend to pronounce 'fn' as 'lambda', the Greek letter used elsewhere for introducing anonymous functions].

Further uses of this idiom would be to recode expultiply so that instead of accepting arguments (a, n, operation, identity) and actually performing a calculation, it accepted just the last two of those values and returned a function of two arguments which would be either multiply or expt. This moves strongly towards viewing programs as things with large numbers of levels of abstraction, starting at the top with recipes for general methods, preparing specialisations of these to get functions to perform particular operations and then using these to solve problems. Keeping this layered structure in place so that code written at one level is not corrupted by issues that should only be visible at another is a major part of the challenge of understanding the proper development of large packages of computer code.

3.10 Exercises

(6.1) In ML, provided f does not try to call itself, there is no difference between the effect of a function definition fun f arg = body; and a value definition involving an anonymous function to the right of the equal sign val f = fn arg =¿ body; Try some of the previous examples using the second version of the syntax to become convinced that all still works. ML does not make it possible to write anonymous functions that call themselves - some of the technical issues that led to this decision may emerge during the final year of the CST courses. As well as function definitions, 'let' statements can be expanded in terms of uses of anonymous functions: let val x = A in B end; can be replaced by (fn x =¿ B)(A); [note that this form is actually a little shorter to type] Convert some previous code to avoid the use of 'let'. This demonstrates that 'fun' and 'let' do not contribute much to the semantic power of ML, although they do help make code more readable.

(6.2) Define a function called compose which can take two one- argument real functions and return the function that is their composition. Thus one would expect compose(sin, sqrt)(2.0); to have the same value as sin(sqrt(2.0)); [warning: it may be necessary to decorate arguments to compose with type information. See how ML displays the types of things that would be suitable as arguments to check what should be written].

(6.3) Double is to add as square is to multiply. Produce a function that will derive double from add and square from multiply. What effect does if have if given (a) difference and (b) quotient?

(6.4) Comment on the relationship between 0; fn x:int =¿ 0; fn x:int =¿ (fn y:int =¿ 0); [hint: the type information that ML displays when you enter one of

the above is closely related to the answer to this question] Now what about (fn x:real =¿ 0);

(6.5) Adjust the differentiation code so that rather than using a single fixed value of h to estimate a derivative it tries values h = 0.1, 0.01,... and so on until two estimated derivatives are within 1code arranged so that the parts of it that have different concerns are well separated, using block structures, functions as arguments and results etc where such things lead to expositions of your solution that are easy to untangle.

(6.6) Reorganise your files on the computer so that you can distinguish between important and frivolous files, and so that files that you want but are not going to update on a daily basis are put away somewhere tidy and safe. Find out how limits on your use of file space are enforced, and how you can discover if you are close to any such limits.

4 Building abstractions with data

4.1 Constructor and selector functions

This section of the course is concerned with combining together pieces of elementary data to build complex structures in much the way that the first section of the course was about combining small operations and expressions to create procedures that could have complicated (and hence interesting and useful) behaviours. As was the case when discussing procedures there will be an underlying emphasis here on arranging the presentation of data so that a clear distinction can be drawn between those aspects of it which are essential for the purposes in hand and those that are purely incidental affect of the particular way that it has been represented on the computer.

A first example of a class of objects best represented by composite data will be the rational numbers. A rational number will have a numerator and a denominator: for instance (13/97) has 13 as its numerator and 97 as its denominator, but behaves as a single value composed of those two parts. Without making any statements in advance about how the numbers will be represented it is reasonable to ask for functions that create and inspect rationals. A constructor function and two selectors will be needed, such that

```
make_rat(p,q)       returns a representation of p/q,
numer z             selects the numerator from a rational,
denom z             selects the denominator.
```

Note that using just these operations it should be possible to build procedures to perform arithmetic on rationals, for instance

```
fun add_rat(x, y) =
   make_rat(numer x*denom y+numer y*denom x,
      denom x*denom y);
```

ML provides secure and automatic ways of introducing abstract data types such as the rational numbers used here, but for now these will be ignored, and make_rat, numer and denom will be implemented using a primitive data constructing facility that ML has. In ML a list of objects, separated by commas and enclosed in parentheses, can be considered as making up a single composite object. Thus we can have

```
fun make_rat(a:int,b:int) = (a, b);
```

Extracting components from this form of composite object is achieved by defining a function that has as its argument a template that will match the object. The object's components can then be accessed in the function body. Thus we can write

```
fun numer (a:int,b:int) = a;
fun denom (a:int,b:int) = b;
```

[At this stage it can be revealed that all functions in ML always take a single argument, and all the cases that have been seen where it seems as if several arguments are involved really just reduce to packing the arguments into one composite object at the call and unpacking them again at the start of the procedure. If this is fully understood it can be seen that make_rat could equally well have been defined by

```
fun make_rat x:(int*int) = x;
```

without that calling for any re-implementation of numer and denom].

If functions for addition, subtraction, multiplication and division are complete for rational numbers the above implementation of the abstract datastructure for rationals can be tested. Experimentation will rapidly show that it is unsatisfactory in that, for instance, denom(make_rat(2,6)) comes out as 6 rather than the value 3 that was probably really wanted. The most obvious way of correcting this will be to re-implement make_rat as

```
fun make_rat(p, q)
   let val g = gcd(p, q)
   in (p div g, q div g) end;
```

which ensures that all rationals are kept in lowest terms throughout the system. But note that since we want to think of rational numbers as implemented by an abstract datatype there may be alternative implementations. One would be to leave make_rat in its original form and replace numer and denom by

```
fun numer(p, q) = p div gcd(p, q);
fun denom(p, q) = q div gcd(p, q);
```

[I have assumed that a function gcd has been defined, and that it can only accept integer arguments, since this allows me to leave out the ':int' restrictions on the structure components in these definitions].

It is important to note that whichever solution is selected it will not make it necessary to make changes to add_rat and the higher level functions. We have a hierarchy of levels of abstraction again, and should strive to keep the concerns of these different levels separate.

4.2 Exercises

(7.1) Complete the coding of procedures for the four basic operations applied to rational numbers. Using the initial datastructure (that did not reduce fractions to lowest terms) add up 1/1 + 1/2 + 1/3 + ... + 1/10. What is the denominator of your answer?

(7.2) Install a datastructure that does reduce fractions to their lowest terms and recompute the sum of the first ten reciprocals. What denominator do you get now? Does it make much difference to efficiency if the gcd calculations are done in make_rat or on numer and denom? Would it be proper to be double-certain and make both make_rat and numer and denom reduce fractions to lowest terms? Does your code need adjustment if you are going to have to cope with negative rational numbers?

(7.3) Change your rational arithmetic package so that the rational value (p/q) is represented by the ML-level structure (q, p) (rather than the (p, q) used before).

(7.4) Define suitable datastructures for representing line- segments in the plane in terms of their endpoints. Adapt it to deal with line-segments in 3-space. Produce a function that will find the mid point of a line segment.

(7.5) Constructor and selector functions can still form useful purposes by expressing abstraction boundaries even when the abstract object being handled can be represented using a single item of primitive data. Write suitable constructor and selector functions to support a data type 'number in the range 0 to 4'. Write functions to add, subtract and multiply such quantities, using the rule that the required result is the remainder when the natural integer result is divided by 5.

4.3 Alternative representations of datastructures

One thing that can be asked about a programming system is 'which of the facilities in it are really necessary'. Here it will be demonstrated that the ability to cluster several integers into a single item of composite data (as used for building data

structures, and also to allow functions to seem to accept several arguments) is, contrary to plausible intuition, not strictly needed.

This will be done by showing how the use of functions as result values makes it possible to achieve a remarkably similar effect, and the example taken here will be to recreate a version of make_rat and its friends. Consider

```
fun make_rat(p:int, q:int) =
   let
      fun extract n =
         if n = 0 then p else q
      in extract end;
```

and

```
fun numer x:int->int = x(0);
fun denom x:int->int = x(1);
```

then we can try using the substitution model for procedure elaboration to discover the value that will be returned by numer(make_rat(u, v)). The call to numer expands into

```
(make_rat(u, v))(0)
```

which is

```
extract(0)
```

in an environment where p=u and q=v. This quickly reduces to the value u. Similarly denom(make_rat(u, v)) = v, and so the above definitions behave as is required to simulate the pair datastructure.

It can properly be complained that in ML the passing of two arguments to make_rat implies an implicit use of the built-in mechanism for creating and unpacking pairs of objects. This too can be avoided if we need to, see exercise 8.4 below. In languages like ML it is even possible to dispense with built-in integer arithmetic, simulating that in terms of function values. Exercise 8.5 provides a start towards an explanation of how this can be done.

A more practically useful feature of ML is that it provides a way of packaging data that is similar in its idea to the constructor/access function ideas shown above, but which allows ML to check that access to the data is only made through the defined interface. To use this facility we would tell ML

```
datatype rational_number =
   make_rat of int*int;
fun numer(make_rat(p,q)) = p;
fun denom(make_rat(p,q)) = q;
```

where the first statement introduces and names the new datatype and causes a constructor function make_rat (that will expect two integer arguments) to be brought into existence. Of course if our programs were always perfect this feature of ML would be no different from any of the previously mentioned implementions of make_rat and its friends. As we live in a real and imperfect world the extra checking that is made possible by introducing a new datatype in a way that makes ML fully aware of its separate identity is very valuable indeed.

4.4 Exercises

(8.1) Code and test the functional representation of the rational number datatype. Inspect and try to make sense of the types of all the functions and expressions involved as ML displays them.

(8.2) Any positive integer can be expressed (in binary notation) as a string of bits bn, ..., b2, b1, b0. Write a function that constructs from a number the value that has binary representation bn, 0, ..., 0, b2, 0, b1, 0, b0. [The bits have been spread out and zero bits interleaved between them. Hint: decompose the number by dividing by two, rebuild by multiplication by four]. Hence produce a version of make_rat that packs the numbers a and b together as a binary number ending with the bits b2, a2, b1, a1, b0, a0.

(8.3) If p and q are positive integers, the value 2 p multiplied by 3 q can be used to encode the ordered pair (p, q). Write constructor and selector functions based on this packing scheme. What order of growth is there in the costs of the various implementations of make_rat and numer that have been discussed in this section? In a similar way 2 p (2q+1) also encodes the pair (p, q) - code up the constructor and selector functions and work out what their costs are. A yet further packing scheme would imagine p and q represented as binary (say) numbers and would create a combined number by interleaving the bits of p and q. Investigate that sceme too.

(8.4) Define a function that adds 1 to an integer. Now define a function that adds two to an integer. And one that adds 37 to an integer. Generalise the creation of these by writing a function that could create them all:

```
fun incrementby n:int =
   let fun addn a = a + n
   in addn end;
```

then

```
val add1 = incrementby 1;
val add37 = incrementby 37;
```

Test this function and several variants on the idea. Now what is the interpretation of and value returned by

```
incrementby 2 3;
```

Compare the types, as displayed by ML, and behaviours of incrementby and (fn (x:int,y:int)=¿x+y).

(8.5) In ML I might like to define a series of functions as follows:

```
fun not_at_all f a = a;
fun once f a = f(a);
fun twice f a = f(f(a));
fun three_times f a = f(f(f(a)));
```

and then try to provide a successor function to help me generate further functions in the sequence

```
fun successor n f a = n f f(a);
```

Are there any problems that prevent me from building up a full simulation of integer arithmetic in this way? [Note: this exercise makes heavy use of the idea introduced in 8.4, and it is critical, for instance, that twice was defined as 'twice f a =' rather than 'twice(f,a) ='. As written above it also relies on the fact that a sequence of names 'a b c d' in ML will be treated as if they were bracketed in the order '((a b) c) d'. Even if this is unspeakably confusing you can still type the examples in and try them with, for instance

```
three_times (fn x=>x+1) 2;
```

The types that ML assigns to the above functions will also look odd, containing symbols 'a, 'b and so on. These are a manifestation of ML's capability of type-checking functions even when it does not know exactly what type the arguments will end up having, and 'a, 'b stand for arbitrary types].

(8.6) After 8.5 consider

```
val q86a = twice three_times;
val q86b = three_times twice;
```

How can you interpret the two values just defined?

(8.7) Set up ML 'datatype's for line segments, numbers modulo 5, ranges of real numbers (e.g. (1.0 to 1.5)). Work through the interval arithmetic examples in Abelson and Sussman.

(8.8) 'abstype' in somewhat similar to 'datatype', but it provides extra security by arranging that only a limited number of functions have access to the internal components of a compound object. Check the full ML manual to find out how to use 'abstype' and use it wherever possible (or at least reasonable) in future exercises.

4.5 Lists, trees and sets

The datastructures introduced so far have all been of fixed size and format and have been capable of binding together nothing more than a few numbers or strings. This section of the course introduces variable sized datastructures. The first of these is the list . In ML lists are written in square brackets:

```
[1, 2, 3, 4, 5];
```

is a list of integers (of length 5). There is obviously some strong low-level relationship between a list in this sense and a tuple as seen earlier:

```
(1, 2, 3, 4, 5).
```

The distinction that ML draws between the two concepts requires that all the items in a list have the same type, but then treats all lists of (say) integers as being compatible with one another, whereas with tuples the entries can be of different types but the type of the whole tuple includes information about how many items there are in it. Thus [1,2] is typed as 'int list' while (1,2) is 'int*int'.

Lists in ML can be extended using an operator '::'. This puts a single extra item on the front of an existing list, and long lists can be built up by using it repeatedly

```
1 :: (2 :: (3 :: (4 :: (5 :: [])))));
```

is another way of writing

```
[1, 2, 3, 4, 5];
```

that stresses this structure, and that the front of a list is much more accessible than the end. The empty list [] can also be written as 'nil', which some people prefer.

I will use archaic, arcane, universally vilified but universally known names for the selector functions on lists, defining them by

```
fun car (a :: b) = a;
fun cdr (a :: b) = b;
```

[When these definitions are introduced to ML it will moan. The complaint it has is because these definitions do not explain what the behaviour of car and cdr should be if given empty lists. I view it as an error to try taking car or cdr of an empty list, and will not worry further about this issue].

If x stands for any nonempty list, then we have

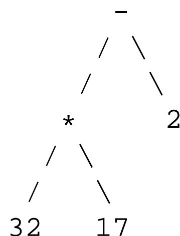
```
x = car(x) :: cdr(x)
```

and if q is an object, and l a list of similarly typed objects,

```
    q = car(q :: l)
and  l = cdr(q :: l).
```

Given the idea of lists, which can be though of as (finite) sequences, there are a fairly large number of useful operations that can be imagined, such as finding the length of a list, appending two lists together or reversing the order of items in a list. Higher level operations involving lists include building a new list which has elements obtained by applying some given function to the elements of an initial list. See exercise 9.2 for an observation about the ML syntax for coding these.

Lists provide a useful low-level representation for data that comes in varying amounts but which is naturally rather flat and list-like. For some applications it is necessary to look at yet more flexible classes of structure. For instance consider the representation of algebraic formulae. Something like 32*17-2 fits in with an abstraction in terms of tree-like structures, where there are leaves containing operand values and internal tree nodes representing operators



ML makes it possible to define a new datatype that will permit us to work with trees of this sort:

```
datatype tree = leaf of int
              | node of string*tree*tree;
```

where the vertical bar indicates that the type permits two variants, one for leaves and on for internal nodes. A leaf node can the be constructed by using the constructor function

```
leaf(2)
```

and the whole of the above tree could be built by saying

```
val lhs = node("*",
               leaf(32),
               leaf(17));
node("-", lhs, leaf(2));
```

Functions to use trees can decompose them using pattern matching in formal parameter specifications in an extension of the mechanism used in the definitions of car and cdr. Thus to add up all the integers in the leaves of a tree,

```
fun addup(leaf(n)) = n
  | addup(node(op,lhs,rhs)) =
        addup(lhs) + addup(rhs);
```

Trees represent a richer class of extensible datastructures than lists: it may not be obvious that it is also useful to have weaker abstractions. A good example of one such is the 'finite set'. Lists can easily be used to represent finite sets (at least sets where all the members have the same type), but there are two things that are wrong with this from the point of view of abstract datastructures. Firstly use of lists reveal an order in which their elements are present, and so the lists [1,2] and [2,1] would be considered different. With sets there should be no ordering of the items included. Then an item is either present in a set or it is not: there is no interpretation to having an item in a set 'twice'. This means that lists such as [1,2,1] are not good representations of sets (because of the repeated entry). See exercise 9.7 for suggestions that show that the abstraction of a set datatype can be supported in several different ways.

4.6 Exercises

(9.1) Write and test functions to find the third element of a list (supposing the list is long enough), to find the last element of a list, and to append two lists.

(9.2) ML provides a neat way of defining functions that work on lists, making the test that is almost always needed for the special case of an empty list fit in neatly with picking out the car and cdr components of non-empty lists. Here is a sample of it

```
fun map(f, nil) = nil
  | map(f, a::b) =
       f(a)::map(f,b);
```

where vertical bars are used to separate collections of disjoint cases to be considered in the function definition. Given the above, what are the values computed by

```
    fun square a:int = a*a;
    mapcar(square, [1,2,3,4,5]);
and mapcar(fn x=>[x], [1,2,3]);
```

(9.3) Write a function that takes a list and returns a list of all possible permutations of the original list. Thus permutations [1,2,3]; could hand back

```
[[1,2,3], [1,3,2], [2,1,3], [2,3,1],
   [3,1,2], [3,2,1]]
```

but any other ordering of the permutations in the result list would be considered acceptable.

(9.4) Here are a pair of ML functions (called cryptically a and r), defined in terms of one another. What do they do? How do they do it? What is the computational complexity of the processes that they generate? Can you produce functions that compute the same results but which have lower order computing time and/or space?

```
fun r(nil) = nil
  | r(p::q) = a(r q, [p])
and a(nil, v) = v
  | a(u, v) = a(r(cdr(r u)),
         car(r u)::v);
```

[Note: in the definition of a the first clause is activated if the first argument to a is an empty list, the second one in all cases when u is not an empty list].

(9.5) Write code to evaluate an arithmetic expression represented by a tree datastructure. Change your definition of tree and your evaluation code so that the items in leaf nodes are rational numbers, and get it working again.

(9.6) Investigate the symbolic differentiation example and code in Abelson & Sussman, and produce a version of it in ML. Keep a clear separation between the parts of your program that are to do with the representation of the algebraic forms, those that implement differentiation rules and any that are concerned with trying to keep expressions reduced to simple form.

(9.7) For an abstract datatype to represent sets we need to support the following operations:

```
create an empty set,
adjoin a new element to a set,
test if a given object is a member of a set,
remove an element from a set,
exhibit one of the members of a non-empty set.
```

Given these, we can define set intersection and union, functions to apply operations to all members of a set and so on. Implement the above to support sets of integers, using as a base representation

1. Lists, where the set constructor operation ensures that no list ever has repeated entries in it,

2. as above, but with the numbers in the list maintained in ascending order (which may make set inclusion tests slightly faster),

3. Binary trees with the integers stored in the leaves, as discussed in Abelson & Sussman.

4.7 An example datatype: complex numbers

On several occasions so far it has been pointed out that some particular datatype could be implemented in a number of different ways. So far the decision about which representation should be used has been made once and for all when the datatype was implemented in terms of constructor and selector functions. There are occasions when the implicit insistence that this makes that data should have a single format of representation is unreasonable. As an illustration of what can be done about this we will follow through code to work with complex numbers, introducing three levels of abstraction. At the top we will just perform arithmetic on complex numeric values. The middle one will be the part of the code that knows that there will be two distinct representations used for these numbers, and beneath that there will be two parallel packages, one implementing the numbers using rectangular (x + iy) co-ordinates and the other using polars (r, theta). It will be useful to give recipes for arithmetic on the two particular representations first.

```
datatype rect = make_rect of real*real;
fun rect_x(make_rect(x,y)) = x;
fun rect_y(make_rect(x,y)) = y;

fun rect_plus(u, v) =
   make_rect(rect_x u + rect_x v,
             rect_y u + rect_y v);
fun rect_times(u, v) =
   let val ux = rect_x u;
       val uy = rect_y u;
       val vx = rect_x v;
       val vy = rect_y v;
   in make_rect(ux*vx-uy*vy,ux*vy+vx*uy)
   end;
fun rect_r u =
   sqrt(square(rect_x u) + square(rect_y u));
fun rect_theta u =
   (* use arctan to compute angle *);
```

which provides addition (easy) and multiplication (somewhat harder) in rectangular co-ordinates. It also provides functions that compute the absolute value of a complex number and the argument of it (i.e. the angle between the real axis and a line joining the origin to the given number). Now for the corresponding code that works in polar co-ordinates:

```
datatype polar = make_polar of real*real;
fun polar_r(pol(r, theta)) = r;
fun polar_theta(pol(r, theta)) = y;
```

```
fun polar_times(u, v) =
   let val ur = polar_r u;
       val utheta = polar_theta u;
       val vr = polar_r v;
       val vtheta = polar_theta v;
   in make_polar(ur*vr,utheta+vtheta)
   end;
fun polar_quotient(u, v) =
   let val ur = polar_r u;
       val utheta = polar_theta u;
       val vr = polar_r v;
       val vtheta = polar_theta v;
   in make_polar(ur/vr,utheta-vtheta)
   end;
fun polar_x u =
   polar_r(u) * cos(polar_theta u);
fun polar_y u =
   polar_r(u) * sin(polar_theta u);
```

Observe, as could have been predicted, that both the polar and rectangular representations can support functions that extract the x and y co-ordinates of complex values and also the polar co-ordinates, but the expenses of the accesses differ significantly across the two representations. Observe also that addition and subtraction are easy in rectangular co-ordinates, while multiplication and division are easy in polars. This suggests that it might be useful to produce functions to convert between the representations:

```
fun to_polar(u) =
   make_polar(rect_r u, rect_theta u);
fun to_rect(u) =
   make_rect(polar_x u, polar_y u);
```

and it then becomes easy to implement some operations that previously looked tricky:

```
fun polar_plus(u, v) =
   to_polar(
      rect_plus(to_rect u, to_rect v));
fun rect_quotient(u, v) =
   to_rect(polar_quotient(to_polar u,
                          to_polar v));
```

That allows us to complete our two separate implementations of complex arithmetic. One will be more efficient when most of the operations performed are additive, the other if most are multiplicative. A way of getting close to the

best of both worlds will be to provide the user with complex numbers that tend to select for themselves the representation that will be most sensible. Define the datatype by

```
datatype complex = case1 of rect
                 | case2 of polar;
```

which just states that a complex number is either a rect or a polar, these types being labelled as case1 and case2. We can then introduce constructor functions that create complex values given x,y or r,theta representations:

```
fun complex_from_xy(x,y) =
   case1(make_rect(x,y))
fun complex_from_rtheta(r,theta) =
   case2(make_polar(r,theta))
```

and all possible selector functions

```
fun complex_x(case1(u)) = rect_x(u)
  | complex_x(case2(u)) = polar_x(u);
```

and so on.

Then complex addition and subtraction can be arranged to leave their results in rectangular form, while multiplication and division give back polar-represented answers. Of course the user of the package is prevented by the barriers of abstraction from knowing this, and will at most be able to deduce that something of that sort is going on by making detailed timing measurements on the implementation of complexes that has been provided. Use of the ML datatype scheme described so far goes some way towards keeping the internal and external details of structures well separated. A further ML feature, called 'abstype' provides yet stronger support for ensuring that all access to data is through the officially associated support functions. Details of this can be found in the ML technical reports and in Wikstrom's book.

4.8 Exercises

(10.1) Code and test complete versions of both the rectangular and polar representations of complex numbers.

(10.2) Glue them together as suggested above to provide a single unified datatype called complex.

(10.3) Create a further layer to your code to provide a special representation for complex numbers that happen to have a zero imaginary part.

(10.4) Do a similar job for rational numbers, making a special case for ones with a denominator that is 1. Then create a layer of code that implements a

datatype called ‘number’ that can have instances that are either rationals or complexes. [Note: the code that you write will require that whenever numbers are to be combined they have the same type, and so is not concerned with adding rational to complex values and so on].

(10.5) Large integers can be represented by lists of digits, so that for instance the number 199731 might be stored as the list [1,3,7,9,9,1]. I have stored the units digit as the first element of the list since this will probably make the rest of the code easier. Write code to add, subtract and multiple big numbers, and implement a layer of datastructure that allows numbers up to 10000 to be stored as normal ML ints, while representing larger numbers in list form.

4.9 Approaches to the implementation of generic operators

The examples given above show that it can be useful to have operators that can be applied to many different classes of data, for instance something to perform addition should have an interpretation for integers, reals, rationals and complex numbers, and possibly other classes of object. ML can deal very cleanly with certain classes of generic operators - ones that are known as ‘polymorphic’. These are ones where the operations to be performed in the implementation of the operator are independent of the full specification of the type of the operands. Good examples of this sort of generality will be found in the procedures that work on lists - most of these are valid regardless of what the type of the elements in the lists are. If you have tried out some list processing examples you will have seen ML describe the types associated with these functions with symbols ’a, ’b and so on in them, where these markers stand for arbitrary type expressions.

The support required by arithmetic is of a quite different character, in that it is clearly of the essence that the insides of a package to perform (say) complex arithmetic will be different from that which does rational arithmetic. We are still used to the abstraction of being able to use a single set of symbols (+, -, * and /) to denote the range of operations involved, with the detailed code to be activated depending on both the operator involved and the type of its operands.

The next short section of this course introduces one particular way of addressing the problem of providing convenient and flexible support for large numbers of related classes of datastructures, while preserving good abstraction barriers so that no piece of code need be aware of information that not properly relevant to it. The scheme is known as ‘message passing’.

Message passing represents an alternative attack on the problem of controlling datatypes to the one embodied in ML, one where typechecking can not be performed on the basis of a static analysis of the code involved. It is quite possible to express message passing code in ML, but the proper and general version of the code involved will have to reveal all the mechanism for coping with para-

meters which have types that can only be determined at run-time. For an initial discussion of the message passing idea this detail will be suppressed, and the examples that follow, although they look like ML, can not be used with the ML system. To indicate this the notes here will have vertical bars to the left of any such pseudo-code.

Back to consideration of the problem of supporting arithmetic if the face of the problem that +, - and so on may need to work with integers, reals, rationals, complex numbers, and indeed they have quite natural interpretations when applied to elements from abstract rings and fields, to polynomials, power series, matrices and so on. The structuring techniques we have introduced so far make it possible to implement each of these arithmetic domains in a tidy way, providing the outside world with a procedural interface that successfully hides internal structure. But the user-level function for addition had to be coded as a long-winded list of cases, dispatching into whichever underlying package was relevant. [I am still setting my face firmly against any discussion of mixed mode arithmetic, things are quite complicated enough without that extra worry].

The most objectionable feature of this way of organising things is that if (when!) a new arithmetic type is introduced, all the user-level interfaces have to be re-coded to cope with it. Thus implementing the new type involves more than just coding the required computations (which will generally form a neat module of code), it involves distributed adjustment of an interface layer.

The first step towards tidying things up is to recognise that the top level dispatch functions are really just implementing a big table lookup from operation and datatype to the detailed code that is relevant for, for instance, multiplying complex numbers. This leads to the suggestion that objects should be represented with manifest type:

```
||   datatype object = thing of string*<any>;
||
||   fun type_of(thing(t,o)) = t;
||
||   fun representation_of(thing(t,o)) = o;
```

Then all operators can be invoked in a uniform way using a new function apply, as in

```
||   apply("~", thing("int", 1));
||
|| where
||
||   fun apply(op, arg) =
||
||      lookup_method(op, type_of arg)(
```

```
||
||          representation_of arg);
```

and now lookup_method can be a single general purpose function that inspects some tables to find the body of code needed, in this case the one associated with the operator name ” ” and the type name ”int”. The table inspected by lookup_method will need updating whenever a new datatype or operator is invented, but it will be regular, static, stylised and hence relatively easy to work with.

Message passing is an idea that corresponds to viewing the lookup_method table as a matrix, and recognising that if we concentrate on rows we get the original dispatch functions corresponding to each user-level operator, but that each column captures exactly the information associated with a particular datatype. This can be implemented by representing all data objects as functions (fortunately we have come across this trick before), which take an operation to be performed as their first argument. A small example illustrating this idea can be given as code to create a representation of a simple integer that responds to requests for its successor and predecessor, and to enquiries as to whether it is positive, negative or zero:

```
|| fun make_int n =
||
||    let fun rep_of_int op =
||
||       if op="inc" then make_int(n+1)
||
||       else if op="dec" then make_int(n-1)
||
||       else if op="=0" then make_bool(n=0)
||
||       else if op="<0" then make_bool(n<0)
||
||       else make_error("Unknown op on int")
||
||    in rep_of_int end;
```

which could then be used by making calls such as

```
||   val one = make_int 1;
||
||   (one "dec")("=0");
```

which ought to return a value standing for true. [make_bool and make_error have not been shown: their behaviour is left to the imagination, as is the problem of interpreting displayed results to determine if they are correct].

Even though the above code may look just as ugly as the type-dispatch that we had earlier in the definition of an addition function, it is not. The reason I assert this is that it is less in breach of the modularity that we want when implementing a new datatype.

4.10 Exercises

(11.1) Message passing does not give any trouble to ML provided all operations implemented for a datatype take the same number of arguments of consistent types, and provided the results returned as responses to all messages are of a single type. Thus it is possible to produce a message- passing model of integer arithmetic supporting the operations ”+”, ”-”, ”*”, ”div”, ”=” and ”¿” provided that the boolean results from the last two requests are encoded as integers (say 0 and 1 for false and true). Write code to demonstrate this.

(11.2) Write a function that accepts a ‘message passing integer’ as from 11.1 and derives from it the regular ML integer that it represents. [Hint: if all else fails positive number’s values can be determined by decrementing them until they reach 0 and seeing how many steps were needed].

(11.3) Read the section in Abelson & Sussman relating to mixing types. [Note: ML does not provide ‘put’ and ‘get’, and for that reason as well as ML’s type-checking and shortness of time further discussion of fine control over datatypes is deferred until a later time in the CST].

(11.4) Design an ML datatype that comes as close as possible to being able to cover all possible ML objects. For instance consider the initial attempt datatype U = i of int — b of bool — prod of (U*U) — fun of (U -¿ U); Exploiting such a universal datatype implement more complete examples of message passing systems.

5 Modularity, Objects and State

5.1 Mutability - advantages and disadvantages

None of the ‘variables’ that have been used so far have in fact ever varied in their values. All the functions that have been written behave in a way that depends solely on the arguments that they are given, and not on the history of their previous use. Datastructures have been created once and for all, and have remained immutable once built. The illusion of change has come about through having many instances of variables (e.g. corresponding to the many invocations of a procedure that is called repeatedly), and by creating edited copies of data.

In this section of the course we investigate the ability to alter things. The main motivation for introducing this is not to increase the power or capability

of our programming language, but to make it possible to produce more natural computational models of certain classes of real world behaviour. In particular it is sometimes quite unreasonable or inconvenient to have two successive identical calls to the same function giving the same result, e.g. the a random number generator might be wanted or the operation being abstracted by the procedure could be withdrawing money from a bank account.

Take the second of these examples. A bank account has state which must be able to alter as transactions are made using it. Normal declarations in ML introduce immutable associations between names and values (even though these associations can be hidden by newer definitions of variables with the same name as the original one). To produce an updatable value in ML it is necessary to make an explicit request for one:

```
val balance = ref 100;
```

makes a fixed association between the name 'balance' and an updatable cell (a reference), which initially has the integer 100 stored in it. Now of course the type of balance is not just int, it is 'int ref', and to extract the int from an int ref it will be necessary to apply an explicit operator. In the case of ML this is written as exclamation mark, so we can now ask

```
!balance > 10;                true:bool
```

and so on.

Updating a reference is done with the ':=' operator.

```
fun withdraw amount =
   if amount < !balance then
      (balance := !balance - amount;
       amount)
   else 0;
```

[Note that this code does not permit overdrafts]. The semicolon operator introduces the idea of sequential steps in the computation, allowing the update operator to proceed for its effect and then returning the amount actually withdrawn. [ML provides a way of raising exceptions, and in a real version of this code it might be more useful to use this mechanism to complain rather than just returning 0 in the 'insufficient funds' case. There is not room for a survey of ideas about exception handling in this course, but those interested in it should certainly find out about ADA as well as ML and various recently designed experimental languages].

The above code does not satisfy proper requirements for code modularity. The variable called balance has its initial value set up in an inflexible way, and is not protected from interference by other pieces of code [for bank balances this is deemed a bad thing!]. The solution uses ideas that have been seen before - the variable to store the balance is made local, and so that it remains available, it is necessary to use a function-producing function to generate code to access it:

```
fun make_account initial_balance:int =
   let val balance = ref initial_balance;
       fun withdraw amount =
          <code as before>
   in withdraw end;
```

where it now is possible to maintain several accounts simultaneously:

```
val student = make_account 50;
val company_director = make_account 230000;
student 24; (* textbook *)
student 11; (* train fares *)
company_director 35000; (* fast car *)
```

It should be clear that updatable cells provide a new degree of flexibility and power when designing abstractions. The cost of this power is not instantly apparent - introducing the new operators !, ;, := and ref into ML looks like a simple extension. But use of these facilities has a global effect on the language. The substitution model of procedure invocation is no longer valid, and even to the extent that it is the order in which parts of a program are elaborated can now alter the meaning of the code. This cuts away the whole basis for formal reasoning about programs that has up until now been available to us. Although there are ways of providing precise explanations of how code behaves in the face of assignment and sequentiality, they are more complicated and include more and deeper pitfalls for the unwary than semantic models for side-effect free computation. Some of the details will be covered later in the CST.

5.2 Exercises

(12.1) Compare and contrast

```
fun make_inc1 x:int =
   fn y => x + y;
```

and

```
fun make_inc2 x:int =
   let val save = ref x
   in fn y => save := !save + y;
```

(12.2) Using a message-passing style of code, design a version of the bank-balance recorder that accepts both ”deposit” and ”withdraw” messages and updates the balance accordingly.

(12.3) Which of the following ML expressions can possibly be meaningful? For those that are devise initial settings for the variables involved to allow them to be evaluated.

```
!!!!exclaim;
!x := 33;
car(x) = x;
car(x) = y;
cdr(x) = x;
```

(12.4) In 'The Art of Computer Programming' by D. E. Knuth (vol 2) (Addison-Wesley, 1974), you will find a lengthy section on the generation of (pseudo-) random sequences of numbers. On the basis of information culled from there, implement a function random such that successive uses of it deliver pseudo-random integers in the range 0 to 999.

(12.5) A turtle starts its life at co-ordinates (0.0, 0.0) in the plane pointing due north. It accepts a sequence of requests which are each of one of the forms

```
turn <angle>
move <length>
```

and keeps as internal state information about where it is. Implement a turtle.

5.3 Sameness. Environments

Prior to the introduction of the idea of assignment there was no difficulty in deciding when two datastructures were identical. [It is impossible to produce a general method for deciding if two pieces of code compute the same function, and so the issues of equality between functions will not be considered here]. With 'ref' objects it things become more complicated, and in particular much harder to formalise. Using an example from the previous section, consider

```
val ac1 = make_account 100;
val ac2 = make_account 100;
```

where obeying the code in make_account does not involve any update operations, and in the two cases make_account is called with the same argument. Thus it seems that the definitions of ac1 and ac2 can be discussed within the framework of substitution semantics and they must be the same. But of course this is not so, in that ac1 and ac2 each independently record their own histories, and the effect above is quite different from that of

```
local val ac = make_account 100
in val ac1 = ac;
   val ac2 = ac end;
```

['local' is very like 'let' but just allows definitions to appear within the scope that it introduces].

This example is a manifestation that the presence of assignment operators anywhere in a program can alter the meaning of even assignment-free parts of the code.

When list and tree-like datastructures are being handled updates within one structure will effect any other structures that share sub-trees containing the overwritten node. There are occasions when the exploitation of this, coupled with careful control over which sub-trees are shared, can simplify and speed up algorithms, but note that it also denies the implementation of the programming language the flexibility of making copies of common datastructures (e.g. distributing them across several separate co-operating computing elements) or of commoning up structures observed to be the same shape (e.g. to save space).

As a piece of notation the programming language Lisp introduces the idea that two objects are eq if they are the same object (so the effects of updating a component of one are seen in the other), and equal if they are structures with identical types and components but with no consideration given to issues of sharing.

In a move towards understanding how imperative programs work a computational model for their evaluation will be sketched. This replaces the idea of substitution to cope with variables with a concept of evaluation relative to an 'environment'. An environment is just a record of the values of variables, though it should be stressed right from the start that during the course of any computation many different environments will be created, and so it is at best loose notation to talk of 'the' value of a variable. The use of environments will be illustrated by repeating the sumsquares example previously expanded using substitution:

```
sumsquares(7+1,7-1)           <empty env>
sumsquares(8, 6)              <empty env>
square x + square y           {x=8, y=6}
square 8 + square 6           {x=8, y=6}
x*x                           {x=8}
8*8                           {x=8}
64                            {x=8}
           x*x                {x=6}
           6*6                {x=6}
           36                 {x=6}
64 + 36                       {x=8, y=6}
100
```

where whenever a procedure is invoked its body is evaluated relative to an environment that shows an association between the names of the formal parameters and the actual arguments. In this model execution of a 'let' statement involves evaluating its body in an environment extended to include a binding for the newly introduced local variables. It is useful to display the extension part to an environment separated from its parent, since this will help when considering cases names

are re-used in an inner block, and it will be essential for an understanding of functional values as arguments and result values. The following is an example where two distinct variables each called x are used.

```
fun silly(x, y) =
   x + (let x = y + 3
        in x + y end);
```

Tracing the execution of a call to silly shows that the body of the 'let' clause is evaluated in an environment that binds x twice. When such things occur the inner binding takes precedence over outer ones.

```
silly(2,5);                 <empty env>
x + (let...)                {x=2,y=5}
2 + (let x = 5 + 3
     in x + y)              {x=2,y=5}
```

now concentrate on the let expression, which is evaluated as

```
x + y                       {x=8}{x=2,y=5}
8 + 5
13
```

and hence the final result is

```
2 + 13
15
```

The environment model for program execution turns out to provide a convenient basis for implementing computer languages. There are subtle issues that arise when functions are treated as first class objects that can be passed around as freely as (say) integers, but these will not be considered in this course.

5.4 Exercises

(13.1) Two versions of the factorial function were used to illustrate the use of substitution semantics for program evaluation. Work through the same examples using environments. [For code not using any imperative features an environment-based evaluator should always get the same results as a substitution based one].

(13.2) An environment for use in integer-only calculations could be represented by a '(string*int) list', with the strings naming variables and the integers indicating values. Using such a representation, and holding arithmetic expressions as trees, show how code to evaluate the expressions could be designed.

(13.3) Consider make_account as shown earlier in these notes. It produces a function-value (withdraw) as its result. The body of withdraw needs access to the

variable ‘balance’. In the environment model of computation this is achieved by demanding the the value of withdraw should consist of the code for withdraw together with the environment (established by make_account) that contains a binding for balance. Follow through some examples using make_account and the function it returns to see how this can work.

(13.4) The function

```
fun copy_list(nil) = nil
  | copy_list(a::b) = a::copy_list(b);
```

copies a list. What parts of the resulting list is eq to the corresponding parts of the original, and what parts are equal ?

(13.5) Compare and contrast

```
let tworefs1 a =
   let r = ref a
   in (r,r) end;
let tworefs2 a =
   (ref a, ref a);
```

5.5 Continuations

The environment way of modelling computation allows us to cope with some of the complexity of having mutable objects present in our universe by making it explicit that the state of all such objects must be carried along with the text that describes what computations are to be performed. We still do not have a fully satisfactory way of explaining the order in which calculations (and hence potential update operations) will get elaborated. The purpose of this section is to introduce one particular way of providing such an explanation. A big attraction of this scheme (which will be referred to as the ‘continuation passing’ style of computing) is that is actually reduces the number and complexity of the basic concepts that have to be built into our model of computation! So far the main building unit for programs has been the function - it is handed a set of arguments and in due course it yields a result. The simplification that the Continuation Passing Style (CPS) makes involves showing that there is no real need for functions to return to their caller. The only thing that will have to be modelled will be calling functions, and the issues about returning from them (maybe having to go back to implicitly saved environments, with all sorts of potential worries about the consequences of side effects on same) do not arise. How is this achieved?

The central idea is that a function which used to expect n arguments will now be given n+1, where the extra arguent will be a further function (the continuation) which will be called when the function has otherwise finished its work. To take a simple example, we would re-write

```
fun ff(x) = x + 1;
```

as

```
fun ff(x, continuation) = continuation(x+1);
```

The continuation will have as its argument the value delivered by the function. Thus returning a value has been replaced by invoking the continuation. Of course this willoften be an iterative-style function call, and so need not be thought of as expensive. The use of continuations makes it possible (and indeed necessary) to make the order of execution of code much more explicit than was previously possible - for instance nested function calls will require multiple continuations which indicate that the inner call is to be completed before the outer one is triggered:

```
fun f(x) = g(h(x));
```

has to become something like

```
fun f(x, cont1) =
    let fun cont2(n) = g(n, cont1)
    in h(x, cont2) end;
```

which might more succinctly be coded as

```
fun f(x, cont) =  h(x, fn n=>g(n, cont));
```

Note how the calls to g and h have been turned inside out so that g will only get processed when h gets around to invoking its continuation.

So far as calling functions is concerned there need be no special distinction between the argument representing a continuation and all the other arguments. Indeed conditional behaviour will be modelled by giving a function two (or more) alternative continuations; with the understanding that in this case the continuations do not need (useful) arguments one would model ML's if by a CPS function

```
CPSif(test:bool, cont1, cont2) =
    if test then cont1() else cont2();
```

and then the absolute value function might be expressed as

```
fun CPSabs(x, cont) =
   CPSif(x < 0,  fn () => cont(~x),
                 fn () => cont(x));
```

A delightful property of CPS is that it makes calling functions and returning results visibly the same operation, and so reveals a perhaps unexpected symmetry about computation. An insight that this can give us is that if we allow function calls to be expressed with multiple arguments (and in this entire section I am viewing multiple arguments as a primitive facility and not looking closely enough to

be able to see how it may be modelled in terms of tuples) then multiple results are naturally supportable. Most programming languages (including ML) do not provide totally satisfactory syntax for this, but in CPS we can show what is required as in a function which computes both the quotient and remainder of two numbers:

```
fun CPSdivide(p, q, cont) = cont(p div q, p mod q);
```

5.6 Exercises

(14.1)Show how an expression f(g(x,y), h(y,x)) would be translated into CPS, giving two versions, one corresponding to calling g first and the other to calling h first.

(14.2)Investigate the use of continuations as a way of modelling conditional constructions (e.g. a three-way if that takes one of three different actions depending on whether its control expression is ¡, = or ¿0), and repetitive constructs (e.g. styled after a BASIC FOR loop).

(14.3)Read the full ML manual to find out about exceptions, and the raise operator. Is the ML exception mechanism as powerful and general as the full use of continuations? Are there facilities it provides that are not easily modelled using continuations?

(14.4)Convert both the recursive and iterative versions of the factorial program (section 3) into CPS. What differences are visible and how do they relate to the distinction between iterative and recursive code?

(14.5)How easy does it look as if it would be to mechanise the transformation of code from ordinary style into CPS? In some informal notation can you give a set of rules that would do the job?

(14.6)Investigate the behaviour of and possible uses for a function fun what_do_I_do(x, cont) = cont(cont); [NB: the ML typechecker will not accept this example - you can still consider what might happen if it did]. What about fun another_oddity(x, cont) = x(cont); given that an identity operation is specified as fun do_nothing(x, cont) = cont(x);

5.7 Queues, tables, objects, streams

The first abstraction to be considered in this section is that of a queue. Queues can be thought of as objects subject to an enquiry 'are you empty' and two operations. The first of these picks off the first item from any non-empty queue, the second adds an item to the end of any queue (empty or not). Very often queues will be used on interfaces between separate bodies of code, with one partner adding items and the other consuming them. In such cases it is clearly necessary to think of the update operations as altering the internal state of the queue. [One could imagine

an attempt to model queues where the add_item procedure produced a new queue with the extra item included. This is all very well, but dodges the issue as to how this new queue is passed across to the consumer process. On a queue of queues?].

As always there are choices to be made when implementing queues, and these need to be made on the basis of code modularity, clarity and robustness as well as the rates of growth of computing time and space used by the implementation. Perhaps the simplest representation of a queue will be as a reference (so it can be updated) to a list. Then the implementation will be something like

```
fun is_empty q =
     !q = [];
fun take_from q =
   let (head_item :: tail) = !q
   in !q = tail; head_item end;
fun add_to(q, item) =
   !q = add_to_end_of_list(!q, item);
```

where the details of add_to_end_of_list are left as an exercise. [In ML it would be yet better to introduce a new datatype for queues, thereby protecting them from confusion with other objects that might be represented as references to lists]. In this implementation the first two functions are pretty well behaved, but add_to can be expected to use both time and space proportional to the number of items stored in the queue. Store use can be reduced by designing add_to_end_of_list to work not be creating a new list but by corrupting the end of an existing one. Then time can be brought down to O(1) by keeping, instead of the single head-pointer q used so far, a pair of pointers to represent a queue, one to the head and one to the tail. [Note: In ML the list datatype protects its users from updating the linkage between list elements. See exercise 15.2 for the ML way of making it explicit that mutability is needed in the datastructure used to implement a queue].

A second class of object that seems to provide a natural model of many useful behaviours, and which necessarily involves update operations, is the table. For the purposes of this course a table is a way of recording an association between keys and values. It is possible to add new entries to a table, delete existing ones and alter the values stored against currently- present keys. This sort of table can provide a model for the records that a bank should keep associating customer names with accounts. Equally it captures the essential features of file directories in a computer system, dictionaries, method-lookup tables for object oriented computing and many other forms of aggregate data. In special cases tables may be read-only, or may be known to have keys that are all integers taken from some restricted range, and of course in these cases special implementations may be possible to take advantage of the limited patter of use: here most emphasis will be given to the general case.

One natural way of implementing a table will be as an object of type

```
(<key> * <value>) list ref
```

where the ref makes it possible to replace the entire body of the table to reflect changes, and the items in the list pair together keys and values. If there are n items stored in the table both access and updating are O(n) processes, and this will frequently be considered unacceptably slow. [Here I am using the common abuse of notation that was mentioned earlier, and using O(n) to suggest that the worst case performance that grows proportionally to n is actually attained. In this example it will be].

A list structured as shown above is known (by the Lisp programming parts of the world, at least) as an 'association list'.

Two-dimensional (and higher) tables do not actually need any new implementation - they can be viewed as simple tables where the key is a composite data type. Thus a two-dimensional array in ML might be a simple tables indexed by keys of type int*int. Equally, however, multi-dimensional tables can be modelled as tables which themselves have sub-tables as entries. As well as being an implementation trick, this idea that a 2D table is really a table of tables has implications at the level of the abstraction that the table provides, in that it distinguishes between the two keys, making it easy to take a row (or possibly column) slice of the complete table.

Tables that use just a single integer key can be viewed as ones in many dimensions if the key is looked at as if it were a large number with each digit of it an index in some new dimension. If the radix for the big-number representation is made two then an integer value (from some pre-specified range) can be viewed as the recipe for a path towards the leaves of a binary tree. Such a tree with maximum depth n can have up to 2 n leaves, or put the other way a tree with n items in it need only have depth log(n). This leads to the idea of representing general tables as trees and attempting to make access and update time proportional to the tree depth (i.e. O(log n), one hopes). For keys that are integers in some known range (e.g. 0 to 31) it is easy to find a good way of deciding where in the tree various values should be stored, for less well specified classes of keys there are a variety of amazingly ingenious methods that guarantee to keep trees reasonably bushy and thereby guarantee to achieve O(log n) computing costs. Some of these will be covered in later years of the CST courses.

Queues and tables both represent attempts to capture the behavioural aspects of things that contain internal state that varies with time. In each case concentration has been on the object that owns the state. The final part of this section of the course provides a different way of looking at history- sensitive behaviour, not based on looking at the object that does the behaving but at the sequence of interactions it has with surrounding code.

A sequence of items of data received from such a body of code is known

as a stream. For the present a working model or example of a piece of stream generating code will be an object that responds to a single message (which might be called ”next”) by returning the next item of data from the sequence that it computes, or possibly a marker value indicating that it has transmitted all the data that it wants to. The next section will discuss both the implementation of stream-generating functions and the use of streams as connective tissue to hold together large bodies of code.

5.8 Exercises

(15.1) Produce the version of add_to_end_of_list that creates a whole new copy of the list in the process of adding an item to the end.

(15.2) The following datatype defines a basis for implementing queues of integers using update-the-tail queues. Produce a queue package using it.

```
datatype cdr_mutable_list = empty |
  item of int*(cdr_mutable_list ref);
```

(15.3) Implement two packages of code that provide support for tables indexed by integers and holding integer values, and which allow new entries to be added to the tables. Design your code so that access to the tables in your two packages costs O(n) and O(log n) steps (on average) respectively. You should also make provision for an enquiry as to whether a table holds an entry for some specified key.

(15.4) Based on one of the table-packages of 15.4 show how to construct a function memoify such that if one has a function fn of type int-¿int then memoify(fn) is also a function of type int-¿int, computing the same values as the original fn but which keeps a tabular record of all the values it ever computes and thereby avoids ever having to obey its body more than once for any particular argument value.

(15.5) In a previous exercise you wrote a function that generated a list of all the permutations of some set of items. Can this code be redesigned to generate a stream of permutations? Can the code to multiply two long integers (originally coded representing the integers as lists of digits) be adapted to accept streams as inputs and produce a stream output?

5.9 Streams - use and implementation

Streams provide a useful tool for building abstractions because they provide aspects of both data management and control structure bundled together in one neat concept. Also, looking at the sequence of values that make up a stream often gives a clearer view of what is happening as a process evolves than does trying

to inspect the changing internal state of the code that generates the stream. It is possible to think of the stream as if it were a static value with all its past and future elements visible, and thus possible to reason about the totality of these values.

Having introduced streams, it becomes natural to produce some functions which operate on streams and produce stream values. To start with it will not be necessary to know how streams are implemented, just how they behave, and in many respects streams behave like lists. Code to handle them will therefore be expressed in terms of a constructor :: and selector functions car and cdr, but with the understanding that the stream versions of these operators will not be exactly the same as the list ones. Note that this interface to streams is different from the message-based one introduced earlier, but is just a different view of the same underlying concept.

The first operation applicable to a stream is that of applying some given function to all the items in the stream, collecting the results into a new stream. This process is known as mapping. For instance the function that squares its argument could be mapped over a stream of integers. The second high level stream function to be mentioned performs filtering - it takes values from its input stream and applies a test function to each. The output stream is made up of just those input values which satisfy the test. The power of filtering can be illustrated by giving a concise program which describes the stream of all prime numbers. The function prime_stream below should be invoked with the stream of integers $\{2,3,4,5,\ldots\}$ as its input, and it generates as its output the stream of primes $\{2,3,5,7,\ldots\}$.

```
||        fun prime_stream (p::rest) =
||
||           let fun not_p n = (n mod p) <> 0;
||
||           in p :: prime_stream(
||
||              filter(not_p, rest)) end;
```

where filter is the stream function that generates an stream consisting of all items in its input that satisfy the given predicate (which in this case achieves the effect of removing all multiples of p from the stream). The vertical bars in the margin by this example are there as a reminder that the code given is not directly valid as ML. Using MLs list notation in the same informal way to denote stream operation the coding of map and filter should be easy exercises.

It is also possible to produce other general and useful functions applicable to streams. One will merge pairs of streams either by simple interleaving of elements or with the selection of which input stream to take an item from being controlled by a user-provided function applied to the two input streams. Others, only really useful for streams of finite length, can combine all the items in a stream using some user-provided function after the style of the function in exercise 5.4.

One way of understanding the relationship between streams and lists is to show how a variation on lists can be used to implement streams. This variation is based on an idea known as ‘lazy cons’. In ordinary ML there is a function, usually written using an infix ::, to create a new component that will be part of a list. One can imagine that inside ML this is a perfectly ordinary function, called (say) cons. In ordinary ML all calls to this function proceed in the regular manner, i.e. the arguments for cons are evaluated and then the body of cons is processed. That body performs system-level operations to allocate some store. The selector functions car and cdr are then do nothing more than to retrieve the two components that cons stored away. Lazy cons is different. The system must recognise lazy cons, and must not evaluate its (second) argument in advance. The system level code then stores away not the value of the second argument, but an unevaluated expression. To compensate for this change, the cdr function is also altered so that it knows that what it will find stored directly in the datastructure will not be a value but will still be an expression in need of evaluation. The overall effect is that the tail of a list (now a stream!) only gets evaluated when somebody tries to inspect it. A consequence is that it becomes quite proper to think in terms of infinite streams, since at any particular stage in the processing only a finite portion will have been evaluated, the rest will be represented by unevaluated expressions stored away in the cdr fields of stream cells.

Perhaps surprisingly it is perfectly possible to model this sort of behaviour in ML. ML defines that the arguments of a function must be evaluated before the body is processed, and so the system can be tricked into delaying evaluation of any expression by writing that expression as the body of some otherwise vacuous function. When it is time to find the value, the packaging function can be invoked (with an arbitrary argument), an operation which is known as ‘force’. Using this trick a stream of integers could be introduced as

```
datatype int_stream =
    cell of int*(int->int_stream);
```

and then the stream of integers starting at some given value n could be constructed using the function

```
fun make_ints n:int =
   cell(n, fn w=>make_ints(n+1));
```

and the car and cdr functions might be coded as

```
fun car(cell(p,q)) = p;
fun cdr(cell(p,q)) = q(0);
```

where the application of q to 0 in the definition of cdr is the force operation.

The ability that streams provide to cope with structures which are notionally non-terminating is very powerful, but it seems that the use of streams causes the order of evaluation of pieces of a program to become dependent on the patterns of access to data, and this can lead to grave confusion if assignment operations that make functions history or order-of-evaluation sensitive are used as well.

5.10 Exercises

(16.1) The sequence of characters typed at a terminal might appear to a computer as a stream of numeric codes representing those characters. Show how it is possible to convert this stream of characters (integers, actually) into a stream of lines (where each line is represented by a list of the codes of the characters involved). Make your design such that it can easily be adapted for use on different computers where the code generated by the ¡return¿ key will not always be the same.

(16.2) Is is possible (even in theory) to produce a stream that represents the digits in the decimal representation of pi?

(16.3) Enumerate, in ascending order and with no repetitions, all the positive integer that have no prime factors other than 2, 3 and 5. The enumeration should be in the form of a stream, which will start 2,3,4,5,6,8,9,10,12,15,..., and an efficient way of generating it produces this by merging together a number of streams each derived from the basic streams 1,2,4,8,16..., 1,3,9,27,... and 1,5,25,....

(16.4) Generate streams that yield that successive terms in infinite power series expansions of the sine and cosine functions. Is it possible to produce a stream-manipulating function that derives from these the infinite power series stream for the tangent function? Given a series stream for the tangent function can the coefficients in an expansion of arctan be derived?

(16.5) Show how, given two infinite streams (A and B), it is possible to construct a stream that has as its elements every possible way of pairing together items taken from A and B.

6 Concluding observations

6.1 From formalism, through implementation to application

This introduction to programming and datastructures has stressed two ideas. The first is that the specification of a procedure or a datatype can represent a firewall that separates the concerns of users of the procedure or type from those of its implementer. Making this separation of concerns explicit enables us to recognise that even simple datatypes may admit several very different implementations, and that

there can be substantial variations between the efficiencies of different procedural solutions to some given problem.

The second idea is that there can be formal models supporting many aspects of computing. These include ideas relating to the estimation of time and space use in a process (as expressed in big-O notation), substitution and environment explanations of the process of computation, analysis of the types of objects being manipulated and the demonstration of a relationship between the use of 'let', 'fn' and 'fun' in ML.

Overall the emphasis has been not on the values computed by some program, but on the way in which the structure of a program can be made to reflect the nature of the problem domain in which it is applicable. The view taken is that this approach will lead to bodies of code where each part has a clear purpose and specification, and hence can be designed and validated in (a reasonable degree of) isolation.

It might seem that this is a self-evidently reasonable goal and will have been adopted by everybody without the need for lecture courses. Particularly for large programs this does not seem to be the case: recognising the proper abstractions that lie beneath the surface of a computational task and deciding how to partition the complete problem into units is hard. Avoiding reliance on features of a sub-function or datastructure that are incidental results of a particular implementation rather than explicitly recognised parts of the specification is also very easy. Some of the ideas introduced in this course help keep interfaces simple and narrow and thus assist in the construction of good code:

Assignment-free functions guarantee to behave consistently whenever they are called: they are neither sensitive to history nor subject to the influence of variables or structures other than their arguments.

Objects thought of in a message passing style provide a way of encapsulating information that has to be updated in place, so that the only way in which the variable data can be accessed will be through messages sent to the object. Viewing the sequence of messages as a stream can sometimes provide a further way of isolating the updates and obtaining a more global description of the behaviour of the complete system.

Datatypes, enforced by your programming language, can both help in documenting the structure of code and impose a degree of consistency in the way in which functions are invoked. The design of type systems for programming languages is still an active area of research, and there are some delicate balances that have to be struck between type systems that are very secure, but which are clumsy and inflexible, and ones that compromise security in order to provide the user with additional capabilities. ML is fairly typical of an emerging generation of well type-checked programming languages.

These notes will no doubt contain errors of various sorts - ranging from simple

typing mistakes to places where brainstorms have led to muddled explanation or bad examples. When you detect any of these, please let me know in the hope that the notes issued in future years will be better. The first place to seek extra assistance with this course will be at the associated practical classes, and by discussing issues with others who are taking the course. College supervisors and Directors of Studies should also be able either to provide help directly or to organise group problem solving sessions. Good Luck!

6.2 Exercises

(17.1) Starting with the Physics of silicon and working upwards towards studies of the social impact of Robotics on society, identify levels of abstraction at which it is possible to talk about computers.

(17.2) Take one of the example programs that you have developed during the term, selecting one that is about a page long. Toss coins to select a number n in the range 2 to 4, and rework your code to introduce n bugs, which should be as subtle and non obvious as you can manage. Exchange your corrupted program with an analogous bug-seeded one prepared by a friend, and find the introduced errors. Do any unintentional bugs or infelicities get uncovered this way? How easy is it to manufacture bugs and how easy is it to find them?

(17.3) When the course giving an introduction to computer hardware is sufficiently under way, read the sections of Abelson & Sussman's book on the design of program and datastructures for modelling electronic circuits and adapt it for the particular case of digital (computer) circuits.

(17.4) Compare and contrast (as possible initial teaching languages) ML as used in this lecture course with the Scheme programming language used by Abelson and Sussman. If you have used BASIC or Pascal before, include them in the comparison.

(17.5) Consider the definition of a function Y given below. Is it possible for ML to assign a type for Y, and if so what is that type. Supposing Y could be used, what would the value (Y f) expand into?

```
fun Y f =
  let fun g h = f (h h)
  in g g end;
```

(17.6) Check the lecture schedules for the rest of this and the remaining years of the CST to see what further questions can be related to ML and the programming-related issues introduced in this introductory course. Start looking into algorithm design and analysis, the technology of constructing compilers for languages like ML, and formal methods for reasoning about programs and proving them to be (in)correct.