1 Objectives

We remind you that the main objectives of the course are to learn:

- To write functional programs
- To write specifications and prove program correctness
- To analyze the sequential and parallel running time of your programs
- To appreciate the potential for parallelism, and use it for efficiency
- To structure your code into modules, with clear, well designed interfaces

Today’s material is mostly concerned with the first two of these.

2 Today’s lecture

An introduction to:

- Functional programming
- Expressions and types
- Declarations, patterns, bindings and scope

This write-up may not cover the same material as in class. As usual, it is a supplement. Also there may be overlap with part of another lecture. No big deal, as you are expected to read all available documents and seeing the same information multiple times can help.

Material on specifications will be included with Lecture Notes 3.
3 Functional Programming

Computation as evaluation

Functional programming is a programming paradigm based on computation as evaluation, as opposed to imperative programming, in which computation causes state change. Programs in a functional programming language are expressions, which denote values, or declarations, which bind names to values. Expressions can be evaluated, and (if evaluation terminates) produce a value. Since evaluation causes no side effects or state change, repeated evaluation of the same expression always produces the same result. This means that it is much more straightforward to reason about functional programs than it tends to be for imperative programs; indeed, this feature is often cited as a key motivation for the development of functional languages. There has been a lot of quasi-religious tub-thumping about the virtues of “pure” functional programming and the perceived sins of “impure” features such as assignment and state. Nevertheless most modern “functional” languages include some impure constructs, for pragmatic reasons to do with efficiency and ease of use. We will begin with pure functional programs and explore later what happens when we allow controlled use of impure features. We will point out advantages of the functional style of programming, but we will also try to give a fair assessment of the alternatives.

Simplicity

A major advantage of functional programming is simplicity (in conceptual terms). Programs behave like mathematical functions, which can be applied to suitable arguments and produce a result. There is a close relationship between functional programs and the mathematical notion of function, and techniques from mathematics and logic are excellent tools for specifying and reasoning about the behavior of functional programs. In particular, principles of mathematical induction, which are used extensively in foundational math and logic, will be crucial for this course. We use induction in one form or another to prove termination of programs, and to prove that programs satisfy their intended specifications.
Referential transparency

A functional language obeys a fundamental principle known as Referential Transparency\(^1\): in any functional program you can replace any expression with another expression that has an “equal” value, without affecting the value of the program. We will clarify what we mean by “equal” shortly, but for the moment just note that integer expressions are equal if they evaluate to the same integer value. So the expressions \( 21 + 21 \) and \( 42 \) are equal. And you probably would agree that \((21 + 21) \times 2\) and \(42 \times 2\) are also equal, as predicted by this principle!

Referential transparency is a powerful principle that supports “equational reasoning” about functional programs. Roughly speaking, this is substitution of “equals for equals”, a notion so familiar from mathematics that we do it all the time without making a fuss. While this may sound obvious, in fact this principle is extremely useful in practice, and it can lend support to program optimization or simplification steps that help us to develop better programs.

It is often said (e.g. in Wikipedia) that imperative languages do not satisfy referential transparency, and that only purely functional languages do. This is inaccurate: we will see later that impure languages also obey a form of referential transparency, but that we need to take account not only of value but also side-effects, in defining what “equal” means for imperative programs.

For functional programs, because evaluation causes no side-effects, if we evaluate an expression twice we get the same value. And the relative order in which we evaluate (non-overlapping) sub-expressions of a program makes no difference to the value of the program, so we can in principle use parallel evaluation strategies to speed up code while being sure that this does not affect the final value.

\(^1\)Sometimes called Frege’s Principle, after the German philosopher Gottlob Frege, who is traditionally cited as the originator of the idea that the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them. Another term associated with this is the Compositionality Principle.
4 Programming Language

We use Standard ML, a "functional" programming language with available implementations for many machine architectures. In lab you will see how to get started using the local implementation. There are also downloadable versions for PCs and Macs.

In addition to being functional, ML is a typed language: an expression can only be used if it has a “type”, and typability is determined statically, from the syntactic structure of the program, without attempting to evaluate. An advantage of this is that a well typed expression never goes wrong when evaluated, in the sense that you ever encounter “stupid type errors” such as an attempt to add 1 to a truth value. Forcing the programmer to pay attention to types prevents an enormous number of common errors. Another advantage is that ML actually uses a sophisticated type inference algorithm, so programmers often need say very little (or nothing explicit) about types and ML infers if the code is typable and if so, what types are plausible.

ML is a call-by-value functional language: functions always evaluate their arguments. In contrast, some functional languages are call-by-name, or lazy (e.g. Haskell). We will show that even though ML is call-by-value one can easily program lazily in ML, so this language design choice is not a limitation.

The ML syntax for function definition allows notation very similar to the style of functional description used in mathematics. In particular, one can define a function by giving a series of clauses (or "cases"), each clause defining the function’s behavior when applied to arguments whose value matches a simple pattern. For example, a function defined on integers may be defined by giving a clause for 0 and a clause for non-zero arguments. Patterns and pattern matching are very useful for structuring code to enhance readability.
5 Types

ML is a typed language. Types include:

- Primitive types such as:
  - int (integers)
  - real (real numbers)
  - bool (truth values)

- Product (or “tuple”) types, built with the infix type constructor *, e.g. int * int * real, (int * int) * real, int * (int * real).

- List types, built with the postfix type constructor list, e.g. int list, (int list) list, (int * int) list.

- Function (or “arrow”) types, built with the infix type constructor ->, e.g. int -> real, (int -> real) -> real, int -> (int -> int).

The type constructors can be nested, and we can use parentheses when needed to disambiguate structure. That being said, in order to encourage more streamlined notation and avoid excessive bracketing there are built-in conventions on priority and associativity, e.g. * binds more strongly than ->, and -> associates to the right. Thus int * int -> real means the same type as (int * int) -> real, and int -> int -> int means the same type as int -> (int -> int).

Note that we gave no association rule for the * operation on types. Indeed, (int * int) * int is not the same as int * (int * int), and neither is the same as int * int * int. These types represent, respectively, pairs of an integer pair and an integer; pairs of an integer and an integer pair; and triples of integers.

Values

Each type represents a set of “values”, and the type of an expression serves as a specification of the kind of value it denotes. For example, an expression of type int -> real denotes a function from integers to reals, and will (unless the application fails to terminate) produce a real result when applied to an integer-valued argument. Product (or tuple) types include
• int * int  
  (pairs of integers)

• int * int * int  
  (triples of integers)

• real * int  
  (pairs of a real and an integer)

• int * real  
  (pairs of an integer and a real); not the same as real * int

Function types include

• int -> int  
  (functions from integers to integers)

• real -> int  
  (functions from reals to integers)

• int * int -> int * int  
  (functions from pairs of integers to pairs of integers)

SML has primitive (built-in) arithmetical operators for combining integers and for combining reals, and the syntax echoes conventional math except that you may need to indicate which type of argument you intend to use. Infix operators include those for addition +, multiplication *, and subtraction -. The (unary) negation operator (minus) is written - to distinguish it from the infix subtraction operator. You can safely use + with two integer expressions, as in 21 + 21, or with two real expressions, as in 21.0 + 21.0, but you cannot mix them up: 21 + 21.0 will cause a type error.

You can turn these infix operators into functions (which can then be applied to pairs of arguments of an appropriate type) using the keyword op, as in op + which can be used as a function of type int * int -> int or as a function of type real * real -> real.

A major cause of type errors for novice ML programmers who may expect + (or * or -) to be “overloaded”: ML does not automatically “coerce” a real to an integer or vice versa. To convert between integers and reals there are built-in functions, such as the function real (of type int -> real) and floor (of type real -> int).
Real numerals include 3.0 and 333.999. Integer numerals include 3 and 42. You cannot use 3.0 instead of 3 (the type is different).

The truth values are written true and false, and they have type bool. You cannot use 1 and 0 (or even 1.0 and 0.0) in places where a truth value is expected.

6 Expressions, declarations and patterns

First some arithmetical examples. In these examples, the comments describe the value denoted by the preceding expression; they also resemble the results produced when we evaluate the expressions using the ML interpreter.

(3+4)*6;
(* = 42 : int *)

(3.6 + 3.4) * 6.0;
(* = 42.0 : real *)

42.0 / 7.0;
(* = 6.0 : real *)

(42 div 5, 42 mod 5);
(* = (8, 2) : int * int *)

5 * (42 div 5) + (42 mod 5) = 42;
(* = true : bool *)

Here is what the ML runtime read-eval-print loop said:

Standard ML of New Jersey v110.73 [. . .]
- (3+4)*6;
val it = 42 : int

- (3.6 + 3.4) * 6.0;
val it = 42.0 : real

- 42.0 / 7.0;
val it = 6.0 : real
val it = (8,2) : int * int

- 5 * (42 div 5) + (42 mod 5) = 42;
val it = true : bool

The last example is an instance of a Fundamental Theorem of arithmetic, that specifies the relationship between div and mod: For all integers \( m \) and all non-zero integers \( n \), \( n \times (m \text{ div } n) + (m \text{ mod } n) = m \).

ML has many built-in primitive operations, some used as “infix” operators (including addition, multiplication and subtraction). Integer division and remainder \( \text{div} \) and \( \text{mod} \) have type \( \text{int} * \text{int} -> \text{int} \) and are infix operators. Real division is the infix operator / of type \( \text{real} * \text{real} -> \text{real} \). There is also a “coercion” function \( \text{real} \) of type \( \text{int} -> \text{real} \).

The ML notation for tuples uses parentheses, e.g. \((1,42)\) and \((1,(2,3))\).

The syntax for functions includes \( \text{fn} \ \ p \Rightarrow \ e \) (a “function expression” or “abstraction”), and application, written as \( e \ e' \) (\( e \) applied to \( e' \)). You may insert parentheses around one or both of the expressions in an application, to emphasize grouping or disambiguate the notation. For example, \( e\,(e') \) or \( (e\ e') \). By convention, application associates to the left, so \( e\,e\,e' \) is the same as \( (e\,e)\,e' \).

Functions can use patterns \( p \) to match against values of their argument type. Patterns include variables, constants (like 0 and \( \text{true} \)), tuples, and lists. All variables used in a pattern must be different, so for example \((x,x)\) is not a legal pattern. The pair pattern \((x,y)\) matches pair values of form \((v1,v2)\), where \( v1 \) and \( v2 \) are values. In particular, this pattern matches the pair \((1,2)\) of type \( \text{int} * \text{int} \), and matches the pair \((\text{true}, \text{true})\) of type \( \text{bool} * \text{bool} \). When a pattern is used to match against a value, if the match succeeds it produces value bindings, of the variables occurring in the pattern.

Matching \((x,y)\) against the value \((1,2)\) succeeds, and binds \( x \) to 1, \( y \) to 2; these bindings are available for use throughout the scope of the pattern. As an example, the scope of the pattern \((x,y)\) in the expression

\[
((\text{fn} \ (x,y) \Rightarrow x+y+3) \ (1, \ 2)) \ + \ 4
\]

is the function body, i.e. the sub-expression \( x+y+3 \). The value of this whole expression is the same as the value of \( (1+2+3)+4 \).
You can, if desired (or required by us!), put type annotations in function expressions. This may help to guide the ML interpreter, or aid in debugging code. For example, \( \text{fn } x \Rightarrow x+1 \) is an abstraction of type \( \text{int } \rightarrow \text{int} \). We could have used any of the following alternatives:

\[
\begin{align*}
\text{fn } (x:\text{int}) & \Rightarrow (x+1):\text{int} \\
\text{fn } (x:\text{int}) & \Rightarrow x+1 \\
(\text{fn } x \Rightarrow x+1) : \text{int } \rightarrow \text{int} \\
\text{fn } x & \Rightarrow (x+1):\text{int}
\end{align*}
\]

In the first few weeks of class, we require you to annotate functions with argument and result types, so that you get used to using types. Later we will see that ML can automatically infer types using a syntax-directed algorithm, so that many of these annotations may safely be omitted.

Here is a simple function, which uses a tuple pattern to match against a pair of integers. When applied to an expression it evaluates that expression to obtain a pair of integers, binds \( x \) and \( y \) to the components, then returns a pair consisting of the quotient and remainder of these two values.

\[
\begin{align*}
\text{fn } (x:\text{int}, y:\text{int}):\text{int*int} & \Rightarrow (x \text{ div } y, x \text{ mod } y); \\
(* & : \text{int } \times \text{int } \rightarrow \text{int } \times \text{int} *) \\
\text{fn } (x:\text{int}, y:\text{int}):\text{int*int} & \Rightarrow (x \text{ div } y, x \text{ mod } y)) \ (42, 5); \\
(* & = (8, 2) : \text{int } \times \text{int} *)
\end{align*}
\]

Above, we used an “anonymous” function expression. You don’t always have to give a function a name. However, if you plan to use it many times, naming it is a good idea, since you can use the name every time you want to apply the function without having to write the entire abstraction. We use a declaration to bind an expression value to a name. For a simple (non-recursive) declaration the syntax is \( \text{let } \overset{\text{val}}{\text{p}} = e \text{ in } e' \text{ end} \). For a simple recursive function definition, the syntax is \( \text{fun } f \overset{\text{p}}{= e} \).

Here are two examples. We attach a comment giving each function’s name and type. After, we give another comment describing an example of the function’s use, and a specification of the function’s behavior. Every function should be accompanied by comments giving its name and type, and a specification that states clearly what assumptions you make about the arguments to which the function will be applied, and what properties the
value returned will have. Later we will introduce a more formal format for presenting specifications, which will help us to remember the key ingredients.

We can use a function name throughout the scope of its declaration. The scope of a declaration (at the top level of the ML interactive window, like this) begins at the declaration and continues unless another declaration for the same name is given later. The second declaration is thus allowed to use the first function. Note the use of \(=\) in the second function’s body, at type \(\text{int} \times \text{int} \to \text{bool}\). The second function’s body also uses a \texttt{let} expression that binds \(q\) and \(r\) to the components of (the value of) \(\text{divmod}(x, y)\) in the expression \(x = q*y + r\). The scope of these bindings is local, only as far as the matching \texttt{end}.

\[
\begin{align*}
\text{fun divmod(x:int, y:int):int*int} & = (x \div y, x \mod y); \\
\text{(\texttt{divmod}) : int \times int \to int \times int} & \\
\text{(\texttt{Specification}: if x:int, y:int, and y<>0,)} & \\
\text{(divmod(x,y) returns the pair (q, r),)} & \\
\text{(where q is the quotient and r is the remainder)} & \\
\text{(of x divided by y.)} & \\
\text{(Example: divmod(42, 5) = (8, 2) : int \times int)} &
\end{align*}
\]

\[
\begin{align*}
\text{fun check (x:int, y:int):bool} & = \\
\text{let} & \\
\text{val (q, r) = divmod(x, y)} & \\
\text{in} & \\
\text{x = q*y + r} & \\
\text{end;}
\end{align*}
\]

\[
\begin{align*}
\text{(\texttt{check}) : int \times int \to bool} & \\
\text{(\texttt{Specification}: For all x:int and all y>0: check(x,y) = true. \texttt{*)}} & \\
\end{align*}
\]

This specification is valid, by the Fundamental Theorem of arithmetic.

The spec for \texttt{divmod} carefully requires that the \(y\) argument is non-zero. There is a good reason for this! Evaluating \texttt{divmod(42, 0)} is “exceptional”, because you can’t divide an integer by zero. ML detects this at runtime and reports the error as \texttt{exception Div}. Later we will discuss in more detail the ML facilities for dealing with runtime errors.
Evaluation

As we said earlier, ML is a \textit{call-by-value} language: functions evaluate their arguments. For example, evaluation of the application

\[
\text{check}(2+2, 5)
\]

begins by evaluating 2+2 (result is 4, obviously!), then 5 (already a value); then evaluates the body of \texttt{check} with \texttt{x} bound to 4, \texttt{y} bound to 5; this will evaluate \texttt{divmod(4, 5)}, which returns the pair (0, 4), then bind \texttt{q} to 0 and \texttt{r} to 4, so the expression \texttt{x=q*y+r} gets evaluated with \texttt{x} bound to 4, \texttt{y} to 5, \texttt{q} to 0 and \texttt{r} to 4. Because \texttt{4 = 0 * 5 + 4}, the result is \texttt{true}.

Or, as ML says:

\[
- \text{check}(2+2, 5);
\quad \text{val it = true : bool}
\]

The above explanation is awkward and somewhat convoluted, because we tried to use English to summarize a computation and there was a lot of sequencing to describe. The value of the expression \texttt{check(2+2,4)} obviously depends on the values of its sub-expressions 2+2 and 5, but also on the value of \texttt{divmod(4,5)}. We will therefore introduce a convenient notation that allows us to be more succinct, and (if necessary) ignore some of the book-keeping. We write

\[
e \Rightarrow^* e'
\]

to mean that evaluation of \texttt{e} reaches \texttt{e'} in zero or more steps. Where relevant we indicate the name-value bindings that get produced (and used in substitutions) during evaluation.

We revisit the earlier evaluation example. Note that

\[
divmod \ (4,5) \Rightarrow^* \ (\text{fn} \ (x,y) \Rightarrow \ (x \ 	ext{div} \ y, \ x \ 	ext{mod} \ y)) \ (4, 5)
\Rightarrow^* \ [x:4, \ y:5] \ (x \ 	ext{div} \ y, \ x \ 	ext{mod} \ y) \quad (* \text{note the bindings} *)
\Rightarrow^* \ (4 \ 	ext{div} \ 5, \ 4 \ 	ext{mod} \ 5) \quad (* \text{after the substitution} *)
\Rightarrow^* \ (0, 4)
\]

Similarly we have

\[
\text{check}(2+2, 5) \Rightarrow^* \ [x:4, \ y:5] \ \text{let val} \ (q,r) = \text{divmod(4,5)} \ \text{in} \ x=q*y+r \ \text{end}
\Rightarrow^* \ [x:4, \ y:5] \ \text{let val} \ (q,r) = (0,4) \ \text{in} \ x=q*y+r \ \text{end}
\]
See why the first fact above (about \texttt{divmod(4,5)}) justifies the second line in this derivation.

Here we have deliberately skirted around the issue of how to give a precise definition of the one-step evaluation relation \texttt{=>} for ML expressions. Even without being precise, by using \texttt{=>*} we are able to abstract away from the details and the number of steps. All of the statements that we make above in the example discussion are valid, and you should be able to understand what they say about expression evaluation at an intuitive level.

\section*{More examples}

Some more examples using declarations, to explain more about bindings and scope. First we bind the name \texttt{pi} to the real number \texttt{3.14}, a not very accurate approximation to the value of \pi. Then we define functions \texttt{circ} and \texttt{area} for calculating the corresponding approximations to the circumference and the area of a circle with a given radius. These function definitions for \texttt{circ} and \texttt{area} are in the scope of this declaration of \texttt{pi}, so the occurrences of \texttt{pi} in their declarations get the value \texttt{3.14}. The attached comments give some examples to illustrate what happens.

\begin{verbatim}
val pi : real = 3.14;
(* pi = 3.14 : real *)

fun circ (r : real) : real = 2.0 * pi * r;
(* circ : real -> real *)

(* Example: circ 1.0 = 6.28 : real *)

(* area : real -> real *)
fun area (r : real) : real = pi * r * r;

(* Example: area 1.0 = 3.14 *)
\end{verbatim}
In the scope of these definitions, \( \pi \) evaluates to 3.14, \( \text{area} \) behaves like the function \( \text{fn } r \to 3.14 \times r \times r \) and \( \text{circumference} \) behaves like the function \( \text{fn } r \to 2.0 \times 3.14 \times r \).

Now let's re-define \( \pi \), binding it to a slightly better approximation.

\[
\text{val } \pi : \text{real} = 3.14159;
\]

(* \( \pi = 3.14159 : \text{real} \) *)

Although this binding “shadows” the earlier one – the current value of \( \pi \) here is 3.14159 – it doesn’t affect the behavior of the functions defined above, since the definitions of \( \text{area} \) and \( \text{circumference} \) given above are still in scope: \( \text{area } 1.0 = 3.14 \), still.

If we now redefine \( \text{area} \), by typing:

\[
\text{fun } \text{area } (r : \text{real}) : \text{real} = \pi \times r \times r;
\]

this introduces a new binding for \( \text{area} \), shadowing the earlier one. Now we get \( \text{area } 1.0 = 3.14159 \).

To maintain consistency we would probably want to redefine \( \text{circ} \) similarly.

\[
\text{fun } \text{circ}(r: \text{real}): \text{real} = 2.0 \times \pi \times r;
\]

(* Example: \( \text{circ } 1.0 = 6.141318 : \text{real} \) *)

We could have used a \text{local} declaration, as follows, to emphasize that the sub-expression 2.0 \( \times \pi \) is needed every time the function gets used:

\[
\text{local}
\]

\[
\text{val } \text{pi2:real} = 2.0 \times \pi
\]

\[
\text{in}
\]

\[
\text{fun } \text{circ}' (r : \text{real}) : \text{real} = \text{pi2} \times r
\]

\[
\text{end};
\]

(* Local binding for \( \text{pi2} \) not in scope here *)

(* \( \text{circ}' 1.0 = 6.141318 : \text{real} \) *)

The functions \( \text{circ} \) and \( \text{circ}' \) are “equivalent” in the sense that when applied to equal arguments they produce equal results. For this reason we say that these functions are \text{extensionally equivalent}, or just equivalent.
7 Lists

ML has a type constructor \texttt{list} (used as a postfix operator) and constructs for building and manipulating lists.

For example, \texttt{int list} is the type of integer lists, \texttt{real list} is the type of lists of real numbers, and \texttt{(real * real) list} is the type of lists of pairs of real numbers. You can also have types such as \texttt{(int list) list} (lists of lists of integers), \texttt{(int -> int) list} (lists of functions from \texttt{int} to \texttt{int}), and so on.

The syntax for list expressions includes enumeration, such as \texttt{[ ]}, \texttt{[1]}, \texttt{[true, false]}, \texttt{[3,1,4,1,5]}; \texttt{nil}, \texttt{x::L}, \texttt{L@R}. Note that :: is called “cons”, and @ is “append”. There is some redundancy in this notation. For instance, \texttt{nil} = \texttt{[ ]}, and \texttt{[1,2] = 1::(2::nil)}. The cons operation :: builds a list from an item and a list; the item must have the same type as all the items in the list. The append operator @ combines two lists (with items of exactly the same type) into a single list by concatenation.

You can use \texttt{nil}, :: and enumerations to build patterns for matching against list values, but \texttt{not} append! For example, \texttt{[ ]} is a list pattern matching only an empty list; \texttt{x::L} is a list pattern only matching non-empty lists (and it binds \texttt{x} to the list’s head value, \texttt{L} to the list’s tail); the pattern \texttt{[x,y,z]} matches lists of length 3, and binds \texttt{x} to the first item, \texttt{y} to the second, \texttt{z} to the third. The syntax \texttt{L@R} is not a legal pattern; to allow append patterns would make matching a much less well-behaved concept (can you see why?).

A value of type \texttt{t list} is a list of values of type \texttt{t}. For example a value of type \texttt{int list} is a list of integers. When writing list values we will either use :: or \texttt{[...]}, whichever is more convenient; in fact :: is the more primitive constructor and \texttt{[1,2,3,...,n]} is a really just a handy abbreviation for \texttt{1::(2::(...::(n::nil)...)}}. By convention :: associates to the right, so this is the same as \texttt{1::2::...::n::nil}.

The append operator evaluates from left to right, then conses the items of the first list on the front of the second. In general, if \texttt{e_1} evaluates to the list value \texttt{L_1 = [v_1,\ldots,v_n]} and \texttt{e_2} evaluates to the list \texttt{L_2}, \texttt{e_1@e_2} evaluates to \texttt{v_1::v_2::\ldots::v_n::L_2}. Because of the order in which evaluation occurs, the number of steps to evaluate \texttt{e_1@e_2} is the sum of the number of steps to evaluate \texttt{e_1}, the number of steps to evaluate \texttt{e_2}, and the length of the list that is the value of \texttt{e_1} (here, \texttt{n}).

14
8 Self-test 2

1. Which of the following, if any, are well-typed, and what are their values?

   (a) 1 + (2 + 3)
   (b) 1.0 + (2.0 + 3.0)
   (c) 1 + (2.0 + 3.0)
   (d) 1 + real(2.0 + 3.0)
   (e) floor(1) + 2.0
   (f) floor(1.4) + 2

2. Which of the following pattern matches succeeds, and what bindings get produced?

   (a) Matching x::L to [1,2,3]
   (b) Matching _::_ to [1,2,3]
   (c) Matching x::(y::_) to [1,2,3,4]
   (d) Matching (x::L, y::R) to ([1,2], [true, false])

3. Give an ML pattern that matches only the following sets of values.

   (a) Pairs containing two non-empty lists.
   (b) Non-empty lists of pairs.

4. Suppose e is an ML expression of type int list and e evaluates to the value [1,2,...,n] in n^2 steps. For each of the following expressions, how many steps are needed to evaluate it to a value (to within an additive constant)? What is the value obtained?

   (a) e@e
   (b) let val L = e in L@L end

5. Write an ML function last of type int list -> int such that for all non-empty integer lists L, last L evaluates to the final item in L. For example, last [1,2,3,4] = 4. What does your function do when you apply it to the empty list?
9 Answers to Self-test 1

1. What values belong to the following types? In each case, describe the set of values and give an ML expression having the given type.

   (a) int * real
       Pairs of an integer (first) and a real number (second)

   (b) real * int
       Pairs of a real number (first) and an integer (second)

   (c) real -> real
       Functions from real numbers to real numbers

   (d) int list -> int
       Functions from integer lists to integers

   (e) int -> int list
       Functions from integers to integer lists

2. Which of the following assertions about equivalence, if any, are true?

   (a) fn x:int => (21+21) is equivalent to fn x:int => 42

   (b) fn x:int => 42 is equivalent to fn y:int => 42

   (c) (fn x:int => (21+21))(3+3) = 42

   (d) (fn x:int => (21+21))(6) = (fn y:int => 42)(3+3)

Each of these is true. Remember that functions are equivalent if they send equivalent arguments to equivalent results, and integer expressions are equivalent if they evaluate to the same integer.

3. Write a recursive ML function product of type int list -> int such that for all integer lists L, product L evaluates to the product of the items in L.

   The most obvious answer, similar to sum from class, is:

   ```ml
   fun product [ ] = 1
   | product (x::L) = x * product L
   ```

   By convention the product of an empty list is taken to be 1, and this makes the above function work as intended. Other solutions are also possible.
4. Which, if any, of the following are true?

(a) \( \text{product } [1,2,3] = 6 \)
(b) \( \text{product } [1,2,3] = \text{product } [3,2,1] \)
(c) \( \text{product } [0,42,42,42,42] = \text{product } [42,42,42,42,0] \)

These are all true. We rely here on the fact that multiplication is associative and commutative, and that \( 0 \times n = 0 = n \times 0 \) for all integers \( n \).

5. Which, if any, of the following are true?

(a) \( \text{product } [1,2,3] \Rightarrow 6 \)
(b) \( \text{product } [1,2,3] \Rightarrow \text{product } [3,2,1] \)
(c) \( \text{product } [0,42,42,42,42] \Rightarrow 0 \)
(d) \( \text{product } [42,42,42,42,0] \Rightarrow 0 \)

(a), (c) and (d) are true. (b) is false, because the first expression does not evaluate to the second, even though they both evaluate down to the value 6.

10 Coming soon

• Testing may be helpful to convince you that a function seems to meet its spec, but testing cannot always cover all cases.

• How to prove that a function meets its specification.

• The most effective proof techniques, especially for recursive functions, are based on induction.

• You will learn to choose an appropriate form of induction, based on the way the function is defined syntactically.