15-150 Fall 2021
Functional Programming
Lectures 1 and 2

©Stephen Brookes

Abstract

We introduce the main themes, objectives and concepts of this class. We illustrate with simple example programs, with emphasis on the main ideas behind the code design but without going into detail on the programming language syntax. We will explore the programming language syntax, concepts, definitions and techniques more thoroughly in later lectures. For now, our primary aim is to get you motivated and interested, as well as to contrast the functional programming style with imperative programming.

1 Introduction

The plan is for lectures to be delivered in person, from 11:50am to 1:20pm in Rashid Auditorium. Please plan to show up in Rashid at the scheduled time, **wearing a face-covering mask** as required by university regulations. Masks must be worn inside the classroom and also in labs or any other meeting room. Anyone who refuses to follow the rules will be asked to leave. Do not attend class if you feel ill or you are running a temperature – instead, contact the student health facility and seek medical treatmemt, and you may need to quarantine yourself. Please contact me by email in case of problems.

I will provide a set of lecture notes to accompany the material on lecture slides. You should plan to study the notes after class and absorb the material — this should help prepare you for homeworks and exams, and allow you to fully appreciate the next lecture! The best way to learn and make progress is to keep up by showing up for class, and paying attention for the entire period of class (yes, I know it's hard!), reading the notes promptly and carefully, and trying some exercises embedded in the notes and slides. Ask me if you get confused, but please make an honest effort to understand the material by yourself first.

In general the lecture notes (and this is the first set to be released) should serve to complement and expand on the material from class. This one covers material from the first two lectures of the semester. Usually some sections of the notes may echo lecture slide content more or less directly, but the notes are not always exactly synchronized with lectures. It's not easy to predict how much material we will get through in a class session, and anyway it sometimes makes sense for me to include extra stuff in the notes.

2 Themes

The main themes of the class are summarized in the following list:

- Functional programming
- Types, specifications, and proofs
- Evaluation and equivalence
- Referential transparency and compositional reasoning
- Correctness, termination, and efficiency
- Exploiting parallelism

Of course we will define all these terms as we go! In the first couple of classes we discuss the main ideas, concepts and techniques informally. As the semester passes we will obviously go into much more detail and you will (we hope) come to appreciate the benefits of the functional style of programming.

3 Objectives

We expect you to learn how to:

- Write well-designed functional programs
- Write specifications for your programs
- Use rigorous techniques to prove program correctness, including use of various forms of induction
- Analyze the sequential and parallel running time of functional programs, including asymptotic analysis
- Choose data structures, design functions, and exploit parallelism to improve efficiency
- Structure your code using abstract types and modules, with clear and well designed interfaces

Again, we also expect you to learn what all these terms mean! Homeworks and exams are intended to help you demonstrate mastery of concepts and techniques.

4 Functional Programming and ML

We use the programming language *Standard ML*, just *ML* for short.

- ML is a *functional* programming language
  - computation = evaluation, to produce a value
- ML is a typed language
  - expressions and declarations must well typed
  - types are determined by a syntax-directed algorithm, based on simple and intuitive rules
  - well-typed expressions and declarations can be evaluated
- Evaluation is type-safe
  - if evaluation terminates you get a value of the expected type
- ML is *polymorphically typed*
  - every well-typed expression has a *most general* type, and can be used safely at any instance of that type
- ML is a *call-by-value* language
  - function calls evaluate their arguments

There are some *advantages*:

- Functional programs are *referentially transparent*
  - expressions of the same type are "equivalent" (equal) when their evaluation produces equal results
  - compositional reasoning: substitution of "equals for equals"
- Functional programs are *mathematical* objects
  - can exploit mathematical techniques to prove correctness
- ML allows *recursive* definition of functions and datatypes
  - can use mathematical *induction* to analyze recursive code
- Evaluation order is largely irrelevant
  - expression evaluation does not cause "side effects", so the order in which independent sub-expressions get evaluated does not affect the result
- We may be able to exploit parallel evaluation
  - to improve runtime, if parallel processors are available

5 Principles

In lecture we listed some general principles, accompanied by pithy catchphrases that you may or may not find easier to remember.

- Expressions must be well-typed.
  Well-typed expressions don't go wrong.
- Every function needs a specification.
  Well-specified programs are easier to understand.
- Every specification needs a proof.
  Well-proven programs do the right thing.
- Large programs should be designed as modules.
  Well-interfaced programs are easier to maintain and develop.
- Data structures algorithms.
  Sensible choice of data structure leads to better code.
- Think parallel, when feasible.
  Parallel programs may go faster.
  Use trees instead of lists if you can.
- Strive for simplicity.
  Programs should be as simple as possible, but no simpler.
  Simple code is usually easier to debug and easier to prove correct!

Some of these statements may sound obvious, but they truly embody important fundamental principles that can improve your programming skills (not just in this class!).

The first principle is a key feature in the SML implementation, which will only *evaluate* an expression if it is well typed. The need to write ML code carefully and obey the "type discipline" can be hard to master for programmers who may be familiar with some other language in which certain "type errors" (such as using the integer 0 instead of a truth value) might get swept under the rug, perhaps by being converted at runtime. However, once you get used to it, you wll find that the benefits outweigh the burden! No stupid runtime type errors. And we will see that ML's implementation actually helps us to reduce the amount of type annotation in our code, since there is a sophisticated built-in algorithm for *inferring* types based on syntax.

6 Introducing functional programming

A brief introduction to functional programming, with some forward references to concepts, techniques and themes that will be developed later in the semester. Don't worry about the details. For now, just try to appreciate the elegance and simplicity!

Types

Expressions (and declarations) in ML must be well-typed, and the rules for types are syntax-directed; only well-typed expressions can be evaluated. *This prevents many common programming errors!*
Today we refer mainly to the types `int` (integers), `int list` (lists of integers), `(int list) list` (lists of lists of integers) and function types like `int list -> int` (functions from integer lists to integers). We also use tuple types like `int list * int` (pairs of an integer list and an integer).

Functions

First, a declaration for a (recursive) function for adding the integers in a list. It is defined using cases: empty list, and non-empty list. The lines of form `(* ... *)` are comments and serve solely to document the intended type and specification of the function.

```
(* sum : int list -> int                                    *)
 fun sum [ ] = 0
   |  sum (x::L) = x + (sum L)

(* Specification:                                           *)
(* For all integer lists L,                                 *)
(*    sum(L) = the sum of the integers in L.                *)
```

Note how similar this function declaration is to a mathematical definition of a function. In fact (as we'll illustrate next and justify later) we can reason about this function's behavior, when applied to an argument (an integer list), using equations derived directly from the ML function definition.

List notation: `[ ]` is the empty list, and `[2,3]` is the integer list with `2` as head and `[3]` as tail. The infix "cons" operator `::` combines a value and a list of values. For example, `1::[2,3]=[1,2,3]`. We use

mathematical notation and math-style equational reasoning to explain how this function works, as follows:

```
sum [1,2,3]
  = 1 + sum [2,3]              by def of sum
  = 1 + (2 + sum [3])          by def of sum
  = 1 + (2 + (3 + sum [ ]))    by def of sum
  = 1 + (2 + (3 + 0))          by def of sum
  = 6                          by basic properties of +
```

Exercise: Show that if `L` is a list of `0`'s, then `sum L = 0`.
(Use induction on the length of `L`)

Next, a function for adding the integers in a list of integer lists. Again recursive, again by cases. We use the `sum` function from above.

```
(* count : (int list) list -> int                          *)
fun count [ ] = 0
 |  count (r::R) = (sum r) + (count R)

(* Specification:                                           *)
(* For all lists of integer lists R,                       *)
(*    count R = the sum of the integers in the lists of R.  *)
```

Show, using equational reasoning as above, and the definitions of `sum` and `count`, that `count [[1], [2,3,4], [5]] = 15`.

Evaluation

Computation is evaluation. Expression evaluation stops when we reach a "value", such as an integer or a list of integers. Addition expressions evaluate from left to right. (Again we're only introducing the ideas, not giving the details on how we define evaluation properly.)

We will write `=>*` for "evaluates in a finite number of steps to". (On the lecture slides we are able to use a nicer symbol $\Longrightarrow^*$ for this purpose, but here we are using a notation that can be entered as text into ML comments.)

```
(* sum [1,2,3] =>* 1 + sum [2,3]                           *)
(*             =>* 1 + (2 + sum [3])                       *)
(*             =>* 1 + (2 + (3 + sum [ ]))                 *)
```

```
(*             =>* 1 + (2 + (3 +0))                          *)
(*             =>* 1 + (2 + 3)                               *)
(*             =>* 1 + 5                                     *)
(*             =>* 6                                         *)
```

Similarly,

```
(* count [[1,2,3], [1,2,3]]  =>* sum [1,2,3] + count [[1,2,3]]  *)
(*                            =>* 6 + count [[1,2,3]]           *)
(*                            =>* 6 + (sum [1,2,3] + count [ ])  *)
(*                            =>* 6 + (6 + count [ ])           *)
(*                            =>* 6 + (6 + 0)                   *)
(*                            =>* 6 + 6 =>* 12                  *)
```

These evaluational derivations look similar to the equational derivations shown above. Evaluation and equational reasoning are, in fact, closely related concepts. We'll explore their relationship later, when we'll see that sometimes equational reasoning has benefits and in some contexts it's better to deal with evaluation directly. As an example to show you why there's a need for both, consider:

```
fun silly(x:int):int = silly(x)
```

We can say using equations based on the function definition that `silly 42 = silly 42`. But that doesn't really tell us anything useful. With evaluational reasoning we'll be able to show that `silly 42 =>* silly 42` and that it takes at least one step to get there, so there is an infinite evaluation sequence and the expression doesn't terminate.

Tail recursion

A recursive function definition is called *tail recursive* if in each clause of the function's definition any recursive call is the last thing that gets done. The definitions of `sum` and `count` are not tail recursive, because for `sum` or `count` on a non-empty list there is an addition operation to do after the recursive call. You can see this in the layout of our evaluation display above.

Here is an addition function `sum'` for integer lists that uses an extra argument as an *accumulator* to hold an integer representing the result of the additions so far, and does an addition onto the accumulator value before making the recursive call. The definition of `sum'` is tail recursive.

```
(* sum' : int list * int -> int                                *)
fun sum' ([ ], a) = a
 |  sum' (x::L, a) = sum' (L, x+a)

(* Specification:
     For all L:int list, a:int,
        sum'(L,a) = sum(L)+a.                                   *)
```

Ue equational reasoning to show that `sum'([1,2,3], 4) = 10`.

Later we will discuss tail recursion in more detail, in connection with efficient implementation; typically tail recursive functions need less stack space, so may be more "space-efficient". Compare the shape of the evaluation diagram for `sum'` with that of `sum`, when applied to `[1,2,3]`. For the tail recursive version the shape doesn't grow wide then shrink back.

```
(* sum' ([1,2,3], 0) =>* sum' ([2,3], 1)                       *)
(*                   =>* sum' ([3], 3)                         *)
(*                   =>* sum' ([ ], 6)                         *)
(*                   =>* 6                                     *)
```

Using `sum'` we can define a function `Sum` that is "equivalent" to `sum`:

```
(* Sum : int list -> int                                       *)
fun Sum L = sum' (L, 0)
```

By *equivalence* we mean that the functions `Sum` and `sum` have the same type, and produce equal results when applied to equal arguments: for all integer lists `L`, `Sum L = sum L`. (This is easy to show by induction on the length of `L`.)

Equivalence and Referential Transparency

As we just said, it is easy to show by induction on the length of `L` that for all integer lists `L`, `Sum L = sum L`. Hence we say the functions `Sum` and `sum` are "equivalent", (or "extensionally equivalent"), written as `Sum = sum`. This notation echoes the usage in math, where functions are regarded as equal if they map equal arguments to equal results.

Induction is a key technique for proving correctness of recursive functions, and for proving termination. We will make extensive use of induction, in various and general forms, throughout the course.

A very important (and useful) property of functional programming languages is that a sub-expression of a program can be replaced by an

equivalent expression, without affecting the evaluational properties of the program. Technically, this is a form of *referential transparency*, often summarized as "it is safe to replace equals by equals", or "the value of an expression depends only on the values of its sub-expressions". And to be completely precise we would need to give a more careful formulation of this property that clarifies what we mean by "without affecting evaluation": basically we mean that the value obtained by evaluation will be equivalent. Here we state a (slightly simplified, but more precise) version of this property.

First, for each type we define a notion of "equivalence":

```
(* Expressions of type int are "equivalent"        *)
(* if they evaluate to the same integer.           *)

(* Expressions of type int list are "equivalent"   *)
(* if they evaluate to the same integer list.      *)

(* Function expressions are "equivalent"           *)
(* if, when applied to "equivalent" arguments      *)
(* they produce "equivalent" results.              *)
```

Referential transparency is the property that:

> Replacing a sub-expression by an "equivalent" sub-expression produces an "equivalent" expression.

It's also possible to break up the above definition of equivalence into one that only applies to (syntactic) values of a given type, and then say that expressions (of the same type) are equivalent iff they evaluate to equivalent values (or both fail to terminate). See the lecture slides, for example.

WARNING:
The discussion above of referential transparency is designed for "pure" functional programs. Contrary to what Wikipedia says, it's not true that referential transparency is a feature uniquely characteristic of pure functional languages, and impure or imperative languages "are not referentially transparent". In fact Standard ML isn't really "pure", as we will see later. (Neither is another well known language, Haskell, also commonly referred to as "functional"). We prefer not to get into dogmatic disputes about purity, the lack thereof, or even the relative merits of SML and Haskell! For "impure" languages there is a more

sophisticated version of referential transparency based on a notion of equivalence that takes account of "effects" such as runtime errors and side-effects. In conclusion, don't believe everything you see on the internet, especially on Wikipedia.

In the meantime we will strive to advance the claim that the "pure" subset of SML (most of the language!) is a powerful programming language in which to explore the advantages of functional style, and we will restrict our attention to this subset for now.

Examples

Understand why the following statements are accurate.

- `21+21` and `42` are equivalent expressions of type `int`.
- The expressions `(21+21)*3` and `42*3` have the same value.
- The functions `Sum` and `sum`, of type `int list -> int`, are equivalent.
- Now, assuming that we have just declared `Sum` as above, consider the following function definition:

  ```
  (* Count : int list list -> int *)
  fun Count [ ] = 0
   |  Count (r::R) = Sum r + Count R
  ```

  By referential transparency, it follows that `Count` is equivalent to `count`.

This discussion is an example of *compositional reasoning* based on referential transparency. For the "pure" functional subset of ML, it is safe to replace an expression by an equivalent expression (of the same type). The result will be equivalent to the original program. We do this kind of compositional or substitutive reasoning all the time in math. You will get used to doing this with functional programs, too!

7 Parallelism

When evaluating a functional program expression, the order in which we evaluate independent sub-expressions does not affect the result. For example, the value of `(2+4)*(3+4)` is `42`, and it doesn't matter if we calculate using left-right evaluation, or right-left evaluation, or parallel evaluation. We would surely all agree that:

```
(2+4)*(3+4) = 6*(3+4) = 6*7 = 42    (left-to-right)
(2+4)*(3+4) = (2+4)*7 = 6*7 = 42    (right-to-left)
(2+4)*(3+4) = 6*7 = 42              (parallel)
```

Addition is an associative operation, i.e. for all integer values x, y, z, $x + (y + z) = (x + y) + z$. Hence it should be fairly obvious that in summing the integers in a collection of integer lists, we should be able to add up the entries in each list *independently* of all the other lists in the collection. Later we will explore using data structures (such as trees and sequences, as opposed to lists) with operations that support parallel evaluation. We will talk about ways to estimate the asymptotic runtime and space usage of a functional program, and we will see that parallelsm can often yield more efficient code.

If we have n integers in a list and we add them sequentially (like in `sum`), it obviously takes $O(n)$ time, provided we make the natural assumption that a single addition takes constant time. There is no way to exploit parallelism if we just have a single list of integers like this and we can only access its items from the front of the list. So sequential summation of a list of length n has to do $O(n)$ "work"; further, even the smartest parallel implementation is hampered by the list structure and must also do $O(n)$ additions in a row, so we say that `sum` has "span" $O(n)$.

Similarly, if we have N integers held in a list of rows, each row being an integer list, using the `count` function to add all these integers (sequentially) is going to take time proportional to N, regardless of the lengths of the rows.

If we have N integers in a list of k rows, each row of length N/k, and an unlimited supply of processors, we could in principle[1] add the rows in parallel (using `sum` for each row, taking time proportional to

[1] Assuming we have primitive functions `reduce` and `map` that use parallel evaluation, we might express this algorithm as `fun parcount R = reduce (op +) (map sum R)`. Such "higher-order" functions permit very concise and elegant program designs. We will return to this topic later!

N/k) and then add the row sums (again using `sum`, in time proportional to k). The total runtime for this algorithm would be $O(k+N/k)$. The work here is still $O(N)$, because you have to do n additions altogether.The "span" is the length of the critical path, which in this case is $O(k + N/k)$: No matter how smart you are at dividing the work among processors, you have to wait until the first phase is over (time proportional to N/k) before starting the second phase (another k units of time).

Finally, if we have n integers at the leaves of a balanced binary tree, and an unlimited supply of processors, we could add using a parallel divide-and-conquer strategy, starting at the leaves and working up towards the root in $O(log\ n)$ phases, each phase taking $O(1)$ time. The work here is $O(n)$ again, but the span is $O(log\ n)$.

Here we have been rather informal about the precise meanings of the terms "work" and "span". Later we will develop these ideas in more detail. Throughout the course we will pay attention to the work and span of the code that we develop. This should help you to become familiar with the potential benefits of parallelism, and you should develop an appreciation for whether and where you can safely exploit parallel evaluation and you should be able to figure out what asymptotic benefits this can bring.

8 Functional Programming

Let's discuss some of the key features and advantages of functional programming again, for emphasis. Some of these points have already been introduced above, but there's no harm in a little repetition.

Computation as evaluation

In a functional programming language, computation is evaluation, to produce a value. In contrast, in an imperative programming language computation causes state change, through destructive operations like assignment that cause side-effects. Expressions in a functional programming language evaluate to values, and declarations produce bindings (of names to values). Evaluation causes no side effects or state change, so repeated evaluation of the same expression always produces the same result. This means it can be easier to reason about functional programs than it tends to be for imperative programs –

there's no need to worry about side effects or the order they occur in. This feature is often cited as motivation for the development of functional languages and to encourage "functional style". 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. (Our language of choice, Standard ML, also contains imperative constructs, but we will pretend for now that ML is purely functional. and we will restrict our attention to the functional subset of ML syntax. Leter we will explore the 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

Expressions in a functional language obey a fundamental principle known as *Referential Transparency*[2]: 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

[2]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 thisidea is the Compositionality Principle.

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) * 2` and `42 * 2` are also equal, as predicted by this principle!

Referential transparency supports "equational reasoning". Roughly speaking, this is "substitution of equals for equals", a familiar notion from mathematics and so common that we do it all the time without making a fuss. While this may sound obvious, this principle is very 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: imperative languages also obey a form of referential transparency, but we need to take account of both values and 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.

9 Standard ML

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.

ML is a *typed* language: an expression can only be evaluated if it has a "type", and types are determined statically, based on syntactic structure, without the need to evaluate. An advantage of imposing this typing constraint 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 (but novice functional programmers may get frustrated at first by the need to obey typing rules). A major 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 an expression has. This greatly reduces the need for programmers to annotate their code with type indications. (However, we will start by insisting that you do include types explicitly, at first, to help you become familir with the type discipline.)

ML is a *call-by-value* language: functions always evaluate their arguments. In contrast, some functional languages are call-by-name, or lazy (e.g. Haskell). 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 (or declaration) is simple and flexible, allowing programs to look 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.

10 Types

ML is a typed language. Only well-typed expressions can be evaluated, and only well-typed declaratiions can be executed. You will need to learn how to write properly typed expressions and declarations! Later we will say (a lot) more about typing rules. For now we will be able to get by with a fairly informal account.

Types built into the ML language 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 * real`, `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 types, built with the infix type constructor `->`,
  e.g. `int -> real`, `(int -> real) -> real`,
  and `int -> (int -> int)`.

The type constructors can be nested, and we use parentheses when needed to disambiguate structure. 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 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 (which is `-`). 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 the these infix operators into functions (which can then be applied to pairs of arguments of an appropriate type) using the keyword `op`. For example `op +` can be used as a function of type `int * int -> int` or as a function of type `real * real -> real`. *This overloading of notation is a major cause of type errors for novice programmers: ML does not automatically "coerce" a real to an integer or vice versa. To convert from integers and reals and back again there are built-in functions,* `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 `1.0` and `0.0`) in places where a truth value is expected.

If you think this section has so far been a bit fussy, that's because we've been trying to explain the technicalities without getting formal, and it's all too easy to get longwinded when writing in English. The next section has some examples to help get familiar with the basics.

11 Expressions and declarations

First some arithmetical examples. Each comment describes the value of 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

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

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

ML uses the name `it` to stand for the value of the last expression you evaluated.

```
- 2+3;
val it = 5 : int

- it * it;
val it = 25
```

The example above involving `div` and `mod` is actually an instance of a Fundamental Theorem of arithmetic, that specifies the relationship

between $\widehat{\text{div}}$ and `mod`: For all integers m and all non-zero integers n, $n * (m \texttt{ div } n) + (m \texttt{ mod } n) = m$.

The integer division operators `div` and `mod` are both infix and have type `int * int -> int`. Real division is the infix operator `/` of type `real * real -> real`. There's no operator analogous to `mod` for reals! There is a "coercion" function `real` of type `int -> real`.

The ML notation for tuples uses parentheses, e.g. `(1,42)` and $(1,(2,3))$ and $(1,2,3,4)$.

The syntax for functions includes simple function expressions uch as `fn p => e` (a "function expression" or "lambda" or "abstraction"), and function 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 e1 e2` is the same as `(e e1) e2`.

Functions can specify patterns `p` to match against argument values. Patterns include variables, constants (like `0` and `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 `int * int`, and matches the pair `(true, true)` of type `bool * 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

```
((fn (x,y) => 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, `fn x => x+1` is an abstraction of type `int -> int`. We could have used any of these alternatives:

```
fn (x:int) => (x+1):int
fn (x:int) => x+1
(fn x => x+1) : int -> int
fn x => (x+1):int
```

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.

An example

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.

```
fn (x:int, y:int):int*int  => (x div y, x mod y);
(*        : int * int -> int * int                    *)

(fn (x:int, y:int):int*int => (x div y, x mod y)) (42, 5);
(*      = (8, 2) : int * int                          *)
```

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 `let val p = e in e' end`. For a simple recursive function definition, the syntax is `fun f 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. This scope 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 `int * int -> bool`. The second function's body also uses a `let` expression that binds `q` and `r` to the components of (the value of) `divmod(x, y)` in the expression `x = q*y + r`. The scope of these bindings is local, only as far as the matching `end`.

```
fun divmod(x:int, y:int):int*int = (x div y, x mod y);

(*   divmod : int * int -> int * int                          *)
(*   Specification: if x:int, y:int, and y<>0,                 *)
(*     divmod(x,y) returns the pair (x div y, x mod y).        *)
(*   Example: divmod(42, 5) = (8, 2) : int * int               *)

fun check (x:int, y:int):bool =
let
   val (q, r) = divmod(x, y)
in
   x = q*y + r
end;
(* check : int * int -> bool *)
(* Specification: For all x:int and all y>0: check(x,y) = true.  *)
```

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

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

Here are three (equivalent) definitions for a factorial function, to compute the product of the integers from 1 to `n`, written in math as `n`!, when `n` ≥ 0. We take this product to be 1 when `n` is 0.

```
   fun fact (n:int) : int =
      if n=0 then 1 else
         n * fact(n-1)

   fun fact (n:int) : int =
      if n=0 then 1 else
         (fn y:int => n*y) (fact(n-1))

   fun fact(n:int) : int =
      if n=0 then 1 else
         let val y = fact(n-1) in n*y end
```

Stylistically, each of these definitions for `fact` is acceptable, and each has its own virtues. The first one is perhaps closest in form to the usual math way of defining factorial. The second one shows that when `n` is non-zero we make a recursive call and then multiply by `n`, making this explicit by writing the function expression `fn y => n*y`, which represents the "multiply by `n`" function. The third one introduces a name (`y`) to refer to the value returned by the recursive call, and says what to do to it (evaluate `n*y`). Moreover the third form makes it easy to read off from the syntax the order in which things happen: when `n` is not 0, evaluate `fact(n-1)`, name it `y`, then evaluate `n*y`. (The *same* order of evaluation as happens with the other function definitions!) Of course the scope of the binding of `y` is limited — inside the `let`-expression.

12 Evaluation

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

```
check(2+2, 5)
```

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

Or, as ML says:

```
- check(2+2, 5);
  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 `check(2+2,4)` obviously depends on the values of its sub-expressions `2+2` and `5`, but also on the value of `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 =>* e'
```

to mean that evaluation of `e` reaches `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) =>* (fn (x,y) => (x div y, x mod y)) (4, 5)
             =>* (4 div 5, 4 mod 5)
             =>* (0, 4)
```

Similarly we have

```
check(2+2, 5) =>* [x:4, y:5] let val (q,r) = divmod(4,5) in x=q*y+r end
              =>* [x:4, y:5] let val (q,r) = (0,4) in x=q*y+r end
              =>* [x:4,y:5,q:0,r:4] (x=q*y+r)
              =>* (4=0*5+4)
              =>* (4=4)
              =>* true
```

See why the first fact above (about `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 `=>` for ML expressions. Even without being precise, by using `=>*` 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.

13 Declarations and scope

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

```
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                             *)
```

In the scope of these definitions, `pi` evaluates to `3.14`, `area` behaves like the function `fn r => 3.14 * r * r` and `circumference` behaves like the function `fn r => 2.0 * 3.14 * r`.

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

```
val pi:real = 3.14159;
(*     pi = 3.14159 : 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 `area` and `circumference` given above are still in scope: `area 1.0 = 3.14`, still.

If we now redefine `area`, by typing:

```
fun area (r : real) : real = pi * r * r;
```

this introduces a new binding for `area`, shadowing the earlier one. Now we get `area 1.0 = 3.14159` .

To maintain consistency we would probably want to redefine `circ` similarly.

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

(*   Example: circ 1.0 = 6.141318 : real          *)
```

We could have used a `local` declaration, as follows, to emphasize that the sub-expression `2.0 * pi` is needed every time the function gets used:

```
(*   circ' : real -> real               *)
local
    val pi2:real = 2.0 * pi
in
    fun circ' (r : real) : real = pi2 * r
end;

(* Local binding for pi2 not in scope here *)

(*   circ' 1.0 = 6.141318 : real        *)
```

The functions `circ` and `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 *extensionally equivalent*, or just equivalent.

14 Lists

ML has a type constructor `list` (used as a postfix operator) and constructs for building and manipulating lists.

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

The syntax for list expressions includes enumeration, such as `[ ], [1], [true, false], [3,1,4,1,5]`; `nil`, `x::L`, `L@R`. Note that `::` is called "cons", and `@` is "append". There is some redundancy in this notation. For instance, `nil = [ ]`, and `[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 `nil`, `::` and enumerations to build patterns for matching against list values, but *not* append! For example, `[ ]` is a list pattern matching only an empty list; `x::L` is a list pattern only matching non-empty lists (and it binds `x` to the list's head value, `L` to the list's tail); the pattern `[x,y,z]` matches lists of length 3, and binds `x` to the first item, `y` to the second, `z` to the third. The syntax `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 `t list` is a list of values of type `t`. For example a value of type `int list` is a list of integers. When writing list values we will either use `::` or [...], whichever is more convenient; in fact `::` is the more primitive constructor and $[\mathtt{1}, \mathtt{2}, \mathtt{3}, \ldots, \mathtt{n}]$ is a really just a handy abbreviation for $\mathtt{1} :: (\mathtt{2} :: (\ldots :: (\mathtt{n} :: \mathtt{nil}) \ldots))$. By convention `::` associates to the right, so this is the same as $\mathtt{1} :: \mathtt{2} :: \ldots :: \mathtt{n} :: \mathtt{nil}$.

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 e_1 evaluates to the list value $L_1 = [v_1, \ldots, v_n]$ and e_2 evaluates to the list L_2, $e_1 @ e_2$ evaluates to $v_1 :: v_2 :: \ldots :: v_n :: L_2$. Because of the order in which evaluation occurs, the number of steps to evaluate $e_1 @ e_2$ is the sum of the number of steps to evaluate e_1, the number of steps to evaluate e_2, and the length of the list that is the value of e_1 (here, n).

15 Self-test

These questions test your understanding of the lecture slides and these notes. We haven't yet gone into the details behind the notation, so use your intuition and common sense to fill the gaps. Try to understand the reason behind each answer. If you're not yet familiar with the program notation, try to learn from the way we write it here!

1. For each of the following types, describe the set of values of that type and give an ML expression having the given type. For example, for the type `int * real` values are pairs of an integer and a real number, and the expression `(0.12,42)` has this type.

   (a) `real * int`

   (b) `real -> real`

   (c) `int list -> int`

   (d) `int -> int list`

   Notice that `int * real` and `real * int` are not the same type! **Test your answers using the ML implementation.**

2. Which of the following assertions, if any, are true? Say why.

   (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)`

   Note that `=` here means equivalence, not the `=` operator in ML.

3. Recall the function `sum:int list -> int` defined above. Which, if any, of the following are true? Say why.

   (a) `sum [1,2,3] = 6`

   (b) `sum [1,2,3] = sum [3,2,1]`

   (c) `sum [1,2,3] =>* 6`

   (d) `sum [1,2,3] =>* sum [3,2,1]`

4. A temperature is measured as a real number (of degrees), but there are two temperature scales, in which degrees mean different things. (The boiling point of water is `100.0 C` and `212.0 F`.)
   - To convert from Fahrenheit (F) to Celsius (C), subtract 32.0 then multiply by 5.0/9.0.
   - To convert from Celsius to Fahrenheit, multiply by 9.0/5.0 then add 32.0.

   Write ML functions

   ```
   c_to_f : real -> real   f_to_c : real -> real
   ```

   that implement these conversions, and check the values of:

   (i) `f_to_c 451.0`
   (ii) `c_to_f (~273.15)`
   (iii) `f_to_c (~40.0)`

5. Which 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`

6. Which pattern matches succeed, with what bindings?
   (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])`

7. Give an ML pattern that matches only the given sets of values.
   (a) Pairs containing two non-empty lists.
   (b) Non-empty lists of pairs.

8. 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?

16 Comments on style

It's a bit early to be preaching about *style*, as we haven't really covered enough of the ML language for you to be able to make an informed and tasteful choice of syntax. Program style is a highly contentious topic: one person's elegant one-liner may be regarded as horrendously obscure by others. In later weeks we may return to example programs introduced above, and see how we could have expressed the same algorithmic content in alternative syntax, arguably with better taste. For example, the `gcd` function can be defined with a *nested* if-then-else:

```
fun gcd(x:int, y:int) : int =
   if x>y then gcd(x-y, y) else
   if y>x then gcd(x, y-x) else x
```

The `else` branch is itself an if-then-else expression. Some people get all upset when you do that, and admittedly it is very easy to write weirdly nested code that's hard to unravel – especially if you use weird layout. (Actually the `gcd` function above is perfectly acceptable to me exactly as written, since the flow of control is so clear and simple.) But really the function needs to do one of three things, depending on the result of *comparing* the values of `x` and `y` (which will either be "less-than", or "greater-than", or "equal"), so it is a little inelegant to use a nested if-then-else. ML provides a built-in function

```
Int.compare : int * int -> order
```

where `order` is a built-in type. The values of type `order` are

```
LESS, EQUAL, and GREATER
```

and can be used to indicate the 3-way result of a comparison. Here is a version of `gcd` written with a `case` expression rather than nested if-then-else:

```
fun gcd(x:int, y:int) : int =
case Int.compare(x, y) of
    GREATER => gcd(x-y, y)
  | LESS    => gcd(x, y-x)
  | EQUAL   => x
```

Again layout on the page is worth paying attention to. We aligned the `=>` symbols in the three clauses of the `case` expression, to make it easier to understand the syntactic structure. Do NOT write something as badly laid out as

```
fun gcd(x:int, y:int) : int =
case Int.compare(x, y) of LESS => gcd(x, y-x)
  |  GREATER =>
  gcd(x-y, y)   |  EQUAL   => x
```

A common style mistake that may signal a failure to think things through before writing code is to use redundant syntactic structure: something long or verbose where there's a more succinct and clear way. For example:

```
fun both(x:bool, y:bool) :bool =
    if x then (if y then true else false) else false
```

This is better expressed (equivalently!) as

```
fun both(x:bool, y:bool) :bool =
    if x then y else false
```

and even more succinctly as

```
fun both(x:bool, y:bool) :bool = x andalso y
```

It's almost never good to write `if b then true else false`, as it's clearly more succinct (and equivalent) to just write `b`.
Similarly `if b then false else true` is just a long-winded way of saying `not b`.

Another common bad design idea is to introduce tons of helper functions. We will try to teach you to use sensible helpers and avoid irrelevant ones. Our slogan is: Helper functions should really help!

Style Advice

Study (and learn to use) the syntax and program style used *in class* and *in slides and notes*. This (usually!) is tasteful and clear. Don't use exotic ML syntax that you pick up on the internet, especially when NOT yet shown in class or labs [3]. (This also applies to math notation. Don't use $\simeq$ for "equality", as I always use `=` for this purpose!)

If you have questions about syntax or style, ask the professor or a TA! We're always (usually) willing to pontificate.

[3] If you do this, you may be asked to explain where you got your code from!

17 Coming soon

- Testing may be helpful to convince you that a function seems to meet its specification, but testing cannot always cover *all* cases.
- We'll show you how to write clear and useful specifications, and *prove* that a function meets its specification.
- The most effective and generally applicable proof techniques, especially for recursive functions, are based on *induction* of some kind.
- You will learn to choose an appropriate form of induction, based on the way the function is defined syntactically.