15-150 Fall 2023

Stephen Brookes

LECTURE 2

Basic concepts

Types, expressions, declarations

Make a plan

- Attend **Class, Labs**
  - **Study** lecture notes and slides
- Start **Homework** promptly
  - Hand in **on time**
  - **Don’t cheat**
  - **All work for credit must be your own**
- **Office hours** *Email me if you need to talk*

Today

Introduction to ML

- Types, expressions and values
- Declarations, binding and scope
- Some example programs
  - *Calculating square roots*

ML syntax

A brief introduction …

- **Types** t
- **Expressions** e
  - Numerals n (e.g. 42, ~42)
  - Identifiers (variables, names) x, y, f, F, sum
- **Declarations** d
- **Patterns** p

Types

t ::= **int**	*integers*
| **real**	*real numbers*
| **bool**	*truth values*
| $t_1 * t_2 * \ldots * t_k$	*tuples*
| t_1 -> t_2	*functions*
| t_1 **list**	*lists*

There are typing rules
for *well-typed* expressions

Only *well-typed* expressions
can be evaluated

Expressions

e ::= x	*identifiers*
| n	*numerals*
| e_1 + e_2 | ~e	*arithmetic ops*
| **true** | **false**	*truth values*
| e_1 **andalso** e_2	*logical ops*
| **if** e_0 **then** e_1 **else** e_2	*conditional*
| $(e_1, \ldots, e_k)$	*tuples*
| **fn** $(x{:}t_1){:}\ t_2$ => e_2	*functions*
| e_1 e_2	*application*

\+ lists, reals, …

\+ declarations

list expressions

e ::= nil	*empty list*
| e_1 :: e_2	*cons*
| e_1 @ e_2	*append*
| [e_1, …, e_k]	*enumeration*

+ built-in functions, e.g.

```
length : 'a list -> int
    hd : 'a list -> 'a
    tl : 'a list -> 'a list
```

declarations

d ::= **val** x = e
| **fun** f(x:t_1):t_2 = e
| d_1; d_2

value binding
recursive function
sequential composition

e ::= **let** d **in** e_1 **end**

d ::= **local** d_1 **in** d_2 **end**

scoped use of a declaration

Values

- For each type `t` there is a set of (syntactic) *values*

- An expression of type `t`
  evaluates to a value of type `t`
  (or fails to terminate)

TYPE	(syntactic) VALUES	
• **int**	*integer numerals*	*42, ~42*
• **real**	*real numbers*	*42.0, 4.2, ~4.2*
• **bool**	*truth values*	**true**, **false**
• **t_1 -> t_2**	*functions* from… **t_1** to… **t_2**	**fn** (x:t_1):t_2 => e_2
• **t_1 * … * t_k**	*tuples of values of types* **t_1** *to* **t_k**	(v_1, …, v_k)
• **t list**	*lists of values of type* **t**	nil, v_1::v_2, [v_1,…,v_k]

Functions are **values**

A function value of type $\mathbf{t_1 \rightarrow t_2}$
has the form
fn (x : $\mathbf{t_1}$):$\mathbf{t_2}$ => e
where, if x has type $\mathbf{t_1}$, e has type $\mathbf{t_2}$

A function value of type $\mathbf{t_1 \rightarrow t_2}$
denotes
a *partial function* from values of type $\mathbf{t_1}$
to values of type $\mathbf{t_2}$

Examples

expression	value : type
(3 + 4) * 6	42 : int
(3.0 + 4.0) * 6.0	42.0 : real
(21+21, 2+3)	(42, 5) : int * int
fn x => x+42	**fn** x => x+42 : int -> int
fn x => 2+2	**fn** x => 2+2 : int -> int

Examples

- A function value of type **int -> int**
  denotes a *partial function from* $\mathbb{Z}$ *to* $\mathbb{Z}$

fun even(x:int):int = **if** x=0 **then** 0 **else** even(x-2)

even denotes $\{(v, 0) \mid v \geq 0 \;\&\; v \;\textbf{mod}\; 2 = 0\}$

even 42 evaluates to 0
even 41 loops forever

functions *are* functions

- An ML function expression F of type t_1 -> t_2 may *evaluate* to…
- … an ML function value **fn** $(x{:}t_1){:}t_2$ => e which *denotes*…
- … a *partial* function f from (values of type) t_1 to (values of type) t_2

f v_1 is *undefined* when F v_1 *doesn't terminate*

f v_1 = v_2 when F v_1 $\Longrightarrow^*$ v_2

ML system

```
Standard ML of New Jersey v110.98.1
- e;
 val it = v : t
```

You enter an expression e

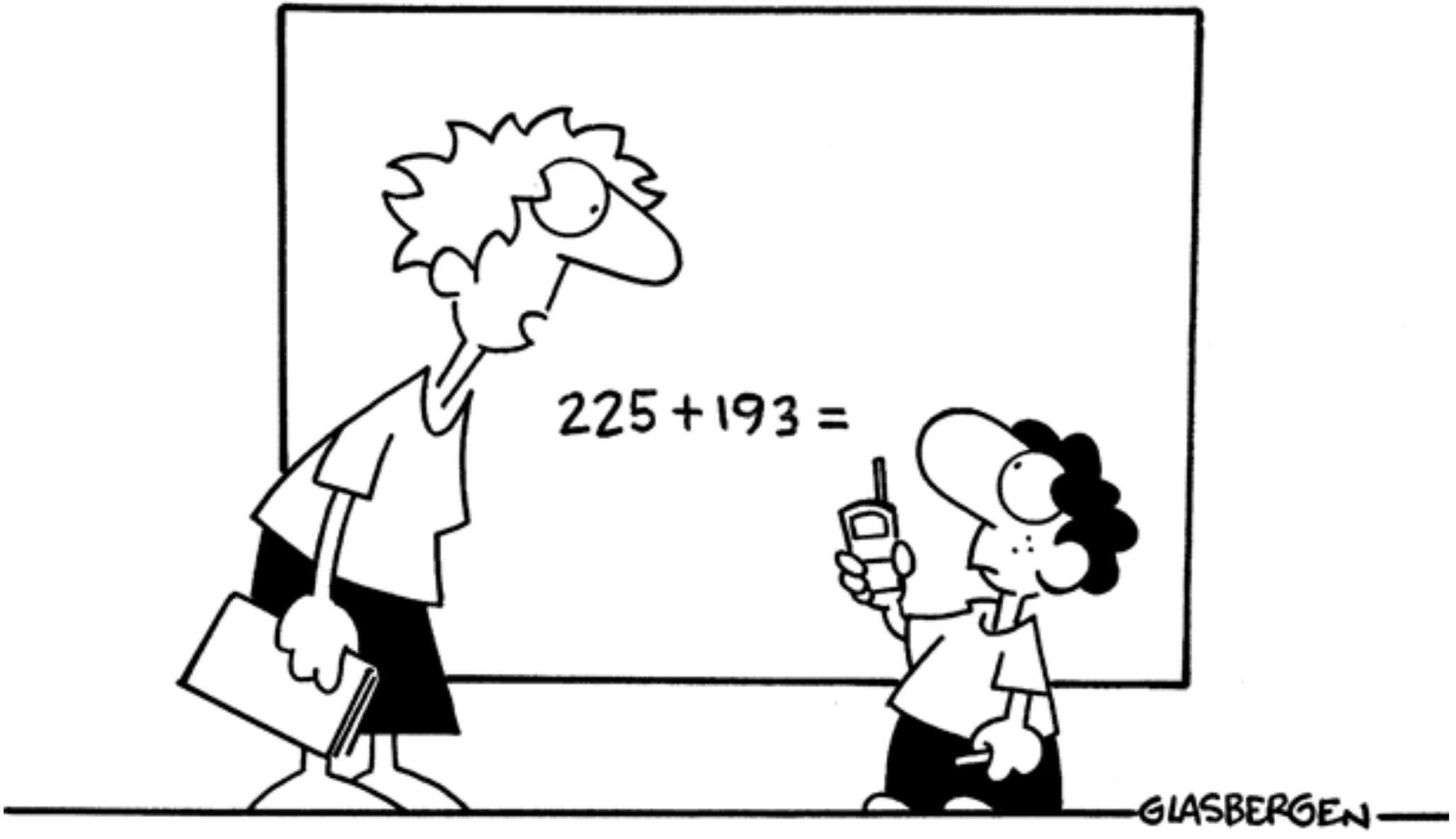
- The system checks the type…
- … and evaluates it, to a value v.

ML system

```
Standard ML of New Jersey v110.98.1
- d;
 val x1 = v1 : t1
 val x2 = v2 : t2
 val x3 = v3 : t3
```

You enter a declaration d

- The system checks the types…
- … and produces bindings,
  of names to values (of the indicated types)



“You have to solve this problem by yourself. You can’t call tech support.”

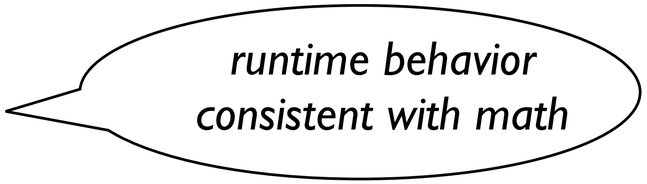
```
Standard ML of New Jersey [...]
- 225 + 193 ;
val it = 418 : int
```

Don't forget the semi-colon.

ML reports the type and value.

$$225 + 193 = 418$$

$$225 + 193 \Longrightarrow^{*} 418$$



Standard ML of New Jersey [...]

```
- fn (x:int):int => 2+2;

val it = fn - : int -> int
```

ML says "it's a function value of type int -> int"

The actual value is **fn** (x:int):int => 2+2
Note: 2+2 doesn't get evaluated (yet)

```
- it 99;
val it = 4 : int
```

Examples

expression	ML says value : type
fn (x:int):int => x + 1	fn - : int -> int
fn (x:real):real => x + 1.0	fn - : real -> real

+, -, * are *overloaded*

+, -, * can be used for
pairs of ints
and for
pairs of reals

Declarations

```
fun double(x:int) : int = x + x
```

```
- val double = fn - : int -> int
```

binds double
to the value

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

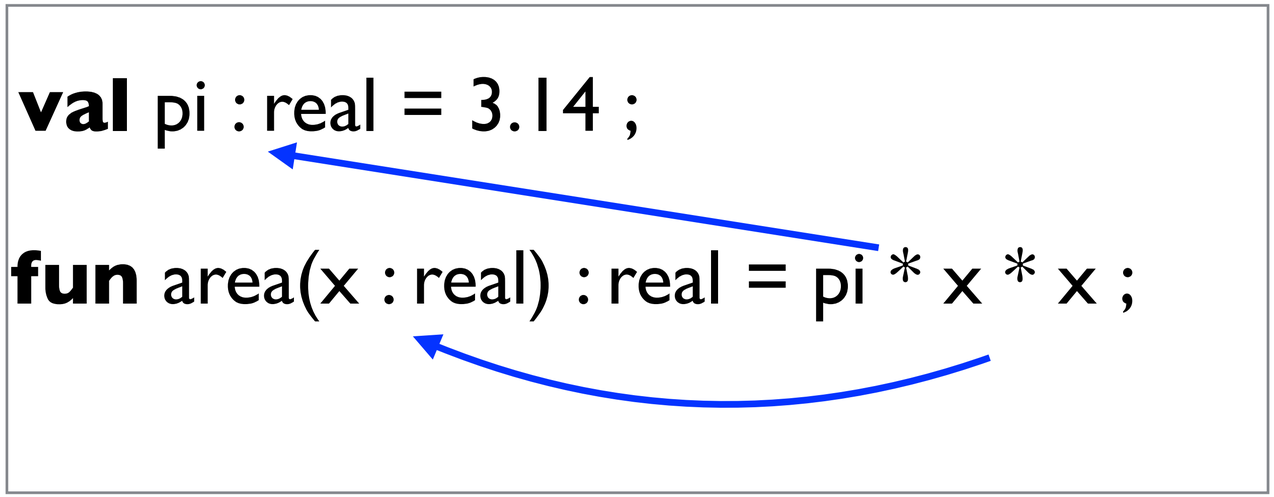
In the *scope* of this declaration,

```
double(double 3)
```

evaluates to 12

Scope

- Bindings have ***static*** (syntax-based) ***scope***

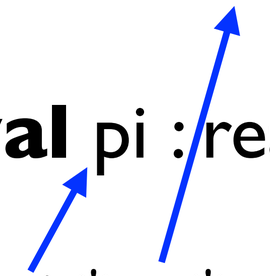


declares pi **and** area

Scope

- Bindings have ***static*** (syntax-based) ***scope***

```
fun area(x : real) : real =
let
   val pi : real = 3.14
in
     pi * x * x
end
```



declares area

Scope

- Bindings have ***static*** (syntax-based) ***scope***

```
local
   val pi : real = 3.14
in
   fun area(x : real) : real =  pi * x * x
end
```

declares area

Summary

- An expression of type `t` can be *evaluated*
- If it terminates, we get a *value* of type `t`
- ML reports the type and value

- Declarations produce *bindings*
- Bindings are *statically scoped*

List expressions

$e ::= nil \mid e_1 :: e_2 \mid [e_1, \ldots, e_k] \mid e_1 @ e_2$

All items in a list must have the *same* type

- nil has type t list
- $e_1 :: e_2$ has type t list
  if $e_1 : t$ and $e_2 : t$ list
- $[e_1, \ldots, e_k]$ has type t list
  if each e_i has type t
- $e_1 @ e_2$ has type t list
  if e_1 and e_2 have type t list

Examples

- [1, 3, 2, 1, 21+21] : int list
- [true, false, true] : bool list
- [[1],[2, 3]] : (int list) list
- [ ] : int list, [ ] : bool list, ...
- 1::[2, 3], 1::(2::[3]), 1::2::[3], 1::2::3::nil
- [1, 2]@[3, 4]
- nil = [ ]

Using ML

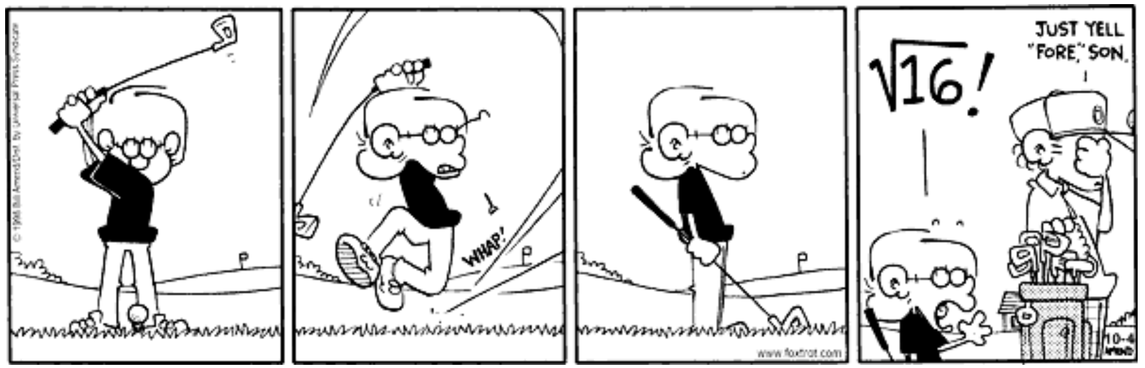
- Next, some ML functions
  designed to solve a simple problem.
- We introduce ML syntax (it's **fun**!)
- Don't worry if not familiar with ML.
- The examples are easy to follow (we hope).

Math background

- Every *non-negative* integer **n** has an integer square root, the unique non-negative integer **m** such that
  $m^2 \leq n < (m+1)^2$
- The integer square root of 6 is 2

How could we write an ML function to compute square roots?

- should have type int -> int
- needs to work for *non-negative* arguments



Finding square roots (1st way)

isqrt_0 : int -> int

```
fun isqrt_0 (n : int) : int =
    let
      fun loop (i : int) : int =
          if n < i*i then i-1 else loop (i+1)
    in
      loop 1
    end
```

- isqrt_0 n uses a *localized* recursive function loop : int -> int

    - loop 1 finds smallest positive integer i such that $n < i^2$ and returns the value of i-1

Finding square roots (2nd way)

isqrt_1 : int -> int

```
fun isqrt_1 (n : int) : int =
  if n=0 then 0 else
     let
        val r = isqrt_1 (n-1) + 1
     in
        if n < r * r then r - 1 else r
     end
```

- isqrt_1 is a recursive function
  - For n > 0, isqrt_1 n calls isqrt_1(n-1)
  - Uses a **let**-binding to avoid recalculation (r is used multiple times)
- Relies on arithmetic facts

Justification for isqrt_1

LEMMA

Let $n>0$ and let k be the square root of $n-1$.

So $k^2 \le n-1 < (k+1)^2$

Either $n < (k+1)^2$, and k is the square root of n,

or $(k+1)^2 \le n$, and $k+1$ is the square root of n.

Proof: easy

This is why we wrote the code!

Finding square roots (3rd way)

isqrt_2 : int -> int

```
fun isqrt_2 (n : int) : int =
  if n=0 then 0 else
    let
       val r = 2 * isqrt_2 (n div 4) + 1
    in
       if n < r * r then r - 1 else r
  end
```

- A recursive function definition
  - For n > 0, isqrt_2 n calls isqrt_2 (n **div** 4)
- Relies on (different) arithmetic facts

…which facts?

Justification for isqrt_2

LEMMA

If n>0 and k is the square root of n **div** 4,

then either 2k or 2k+1 is the integer square root of n.

Proof? Do the math! Show that

2k is the square root of n, if $n < (2k+1)^2$

and 2k+1 is the square root of n, if $n \geq (2k+1)^2$

Results

- All three functions compute integer square root *correctly*
- Try them out on larger and larger integer arguments….
- Can you see any differences?
- Why?

Let's try it

Start up the ML runtime system.
Enter the function definitions for
isqrt_0,
isqrt_1,
isqrt_2,
as given above.

1. Find the value of isqrt_0 123456789.
2. What happens with isqrt_1 123456789?
3. What happens with isqrt_2 123456789?

Questions

- Are the functions
  isqrt_0, isqrt_1 and isqrt_2
  equivalent?
- If so, how could you prove it?
- If not, how could you show it?

Design issues

Pay attention to scope!

```
fun isqrt_0 (n : int) : int =
    let
      fun loop (i : int) : int =
          if n < i*i then i-1 else loop (i+1)
    in
      loop 1
    end
```

loop uses n,
the value passed in as
argument of isqrt_0
so this is *in scope* and OK

Design issues

Pay attention to scope!

```
fun loop (i : int) : int =
    if n < i*i then i-1 else loop (i+1);

fun isqrt_0 (n : int) : int = loop 1
```

loop uses n
but there's no n in scope
so this is NOT OK

Design issues

Pay attention to scope!

```
fun loop (i : int, n:int) : int =
    if n < i*i then i-1 else loop (i+1, n);

fun isqrt_0 (n : int) : int = loop (1, n)
```

This is OK syntactically,
but it seems silly to pass
n around like this!

Lessons

- Expressions return values
- Declarations produce bindings (of names to values)
- Learn to write and use simple ML functions
    - integer square roots
    - try some other arithmetical basics
- Learn about declarations and scope
    - can only use bindings that are in scope