

15-150 Fall 2020

Stephen Brookes

LECTURE 2

Types, expressions and declarations

Make a plan

- **Class, Labs** (remote)
 - **Study** lecture material
- **Homework**
 - Start as early as possible, end on time
 - Don't cheat — ask us if you need advice
- **Office hours** (remote)

Today

- Types, expressions and values
- Declarations, binding and scope
- Introduction to ML syntax
- Some example programs

Types

$t ::= \mathbf{int} \mid \mathbf{real} \mid \mathbf{bool}$	<i>integers, reals, truth values</i>
$t_1 * t_2 * \dots * t_k$	<i>tuples</i>
$t_1 \rightarrow t_2$	<i>functions</i>
$t_1 \mathbf{list}$	<i>lists</i>

There are syntax rules
for *well-typed* expressions

Only *well-typed* expressions
can be evaluated

Expressions

$e ::= x$	<i>variables</i>
n	<i>numerals</i>
$e_1 + e_2$	<i>arithmetic ops</i>
true false	<i>truth values</i>
e_1 andalso e_2	<i>logical ops</i>
if e_0 then e_1 else e_2	<i>conditional</i>
(e_1, \dots, e_k)	<i>tuples</i>
fn $(x:t_1): t_2 \Rightarrow e_2$	<i>functions</i>
$e_1 e_2$	<i>application</i>

+ lists, reals, ...

+ declarations

list expressions

$e ::= \text{nil}$	<i>empty list</i>
$e_1 :: e_2$	<i>cons</i>
$e_1 @ e_2$	<i>append</i>
$[e_1, \dots, e_k]$	<i>enumeration</i>

declarations

$d ::=$ **val** $x = e$ *val*
| **fun** $f(x:t_1):t_2 = e$ *recursive function*
| $d_1; d_2$ *sequential*
| d_1 **and** d_2 *simultaneous*

$e ::=$ **let** d **in** e_1 **end** *scoped use*

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

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

- **int**
- **real**
- **bool**
- **$t_1 \rightarrow t_2$**
- **$t_1 * \dots * t_k$**
- **t_1 list**

VALUES

- integer numerals* 42, ~42
- real numbers* 4.2, ~4.2
- truth values* **true, false**
- functions from... t_1 to... t_2*
fn (x: t_1): $t_2 \Rightarrow e_2$
- tuples of values of type $t_1 \dots t_k$*
(v_1, \dots, v_k)
- lists of values of type t_1*
nil, $v_1::v_2$, [v_1, \dots, v_k]

Functions are values

A function value of type $t_1 \rightarrow t_2$

is a syntactic form

fn $(x : t_1) : t_2 \Rightarrow e$

where, if x has type t_1 , e has type t_2

A function value of type $t_1 \rightarrow t_2$

denotes

a *partial function* from values of type t_1

to values of type t_2

Examples

expression	value : type
$(3 + 4) * 6$	42 : int
$(3.0 + 4.0) * 6.0$	42.0 : real
$(21+21, 2+3)$	$(42, 5) : \text{int} * \text{int}$
fn $x \Rightarrow x+42$	fn $x \Rightarrow x+42 : \text{int} \rightarrow \text{int}$
fn $x \Rightarrow 2+2$	fn $x \Rightarrow 2+2 : \text{int} \rightarrow \text{int}$

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 $\{(v, 0) \mid v \geq 0 \ \& \ v \bmod 2 = 0\}$

even 42 evaluates to 0

even 41 loops forever

ML system

- You enter an expression
 - The system checks it's well-typed...
 - ... and evaluates, to a syntactic value.
-
- You enter a declaration
 - The system checks it's well-typed...
 - ... and produces bindings,
of names to syntactic values.

Standard ML of New Jersey [...]

- 225 + 193;

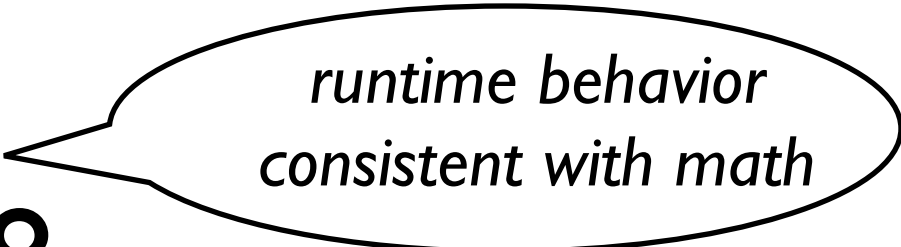
val it = 418 : int

Don't forget the semi-colon.

ML reports the type and value.

$$225 + 193 = 418$$

$$225 + 193 \Rightarrow^* 418$$



*runtime behavior
consistent with math*

Standard ML of New Jersey [...]

- **fn** (x: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 => 2+2

The 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

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,

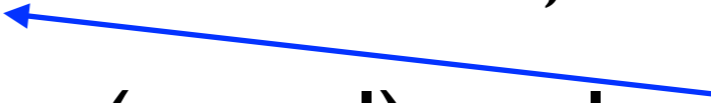
```
double(double 3)
```

evaluates to **12**


Scope

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

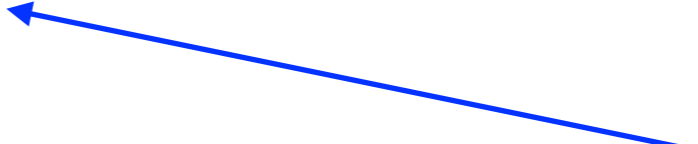
```
val pi : real = 3.14;  
fun area(x:real):real = pi*x*x
```



```
let  
  val pi : real = 3.14  
in  
  2.0 * pi  
end
```



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



Design issues

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

*every call to circ
evaluates 2.0*pi*

```
fun circ(r:real):real =  
let  
  val pi2:real = 2.0 * pi  
in  
  pi2 * r  
end
```

*every call to circ
evaluates 2.0*pi*

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

*2.0*pi only gets
evaluated once*

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
 - `val it = 3 : int`
 - `val it = fn - : int -> int`
- Declarations produce *bindings*
- Bindings are *statically scoped*

*Use well scoped declarations to
avoid re-evaluating code repeatedly*

List expressions

$e ::= \text{nil} \mid e_1::e_2 \mid [e_1,\dots,e_k] \mid e_1@e_2$

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
- $[e_1,\dots,e_k]$ 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, 2|+2|]` : 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 = []`

Examples

- To finish, some ML functions to solve a simple problem.
- Introduces ML syntax (it's **fun!**)
- Don't worry if you aren't 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 integer square roots?

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

Finding integer square root

`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 $n < i^2$
 - returns the value of $i-1$

Finding integer square root

`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

If $n > 0$ and k is the integer square root of $n-1$,
then either k or $k+1$ is the integer square root of n .

Proof? Do the math!

Can show that

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

This is why we wrote the code!

Finding integer square root

`isqrt_2 : int -> int`

```
fun isqrt_2 n =  
  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?

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 2020`
2. What happens when you evaluate `isqrt_1 123456789`?
3. What happens when you evaluate `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?

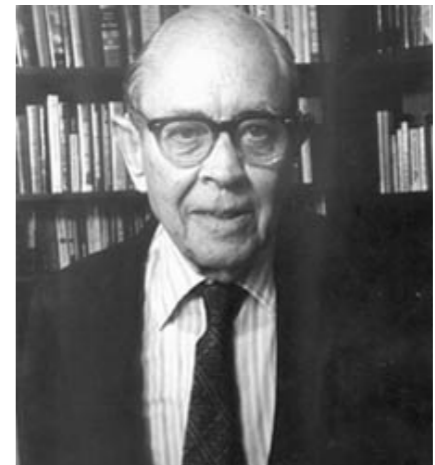
covid testing

- Population size N
- Tests assumed accurate
- Naive testing algorithm:
take a sample from each person and test it
 - needs a total of N tests



We can do better... with fewer tests!

The Detection of Defective Members of Large Populations
Robert Dorfman, *Annals of Math Stats*, 1947



covid testing (a smarter algorithm?)

- Let p be probability that a test is *positive*
- Split population of N into groups of size n
- Test the *grouped samples*
 - prob that a group test is *negative* is $(1-p)^n$
- For each *positive* group, test its members
- The total *expected* number of tests is
 $(N \text{ div } n) + P * (N \text{ div } n) * n,$
where P is $1-(1-p)^n$

if n divides N , simplifies to
 $(N \text{ div } n) + \text{ceil}(P) * N$

fewer tests

```
fun exp(r:real, n:int) : real =  
  if n=0 then 1.0 else r * exp(r, n-1)
```

```
fun cost (N:int, n:int, p:real) : int =  
let  
  val P : real = 1.0 - exp(1.0 - p, n)  
in  
  (N div n) + ceil((real N ) * P)  
end;
```

```
- cost(150,10,0.01);  
val it = 30 : int
```

150 people,
when $p = 1\%$,
can be assessed
with just 30 tests

TBD

- Given N and p , what's the optimal n ?
 - the cheapest method