15-150 Fall 2013
Lecture 16

Stephen Brookes

Midterm grades

A	87+
B	77+
C	67+
D	55+

WEIGHTS

exam	50%
hw	40%
lab	10%

This week

- **Labs**: pick up your exams
- **Homework**: *exceptions*, *abstract data types*
- **Lectures**: *modular programming*
  - Designing large programs
  - Information hiding
  - Representation independence

problem of the day

Write an ML function

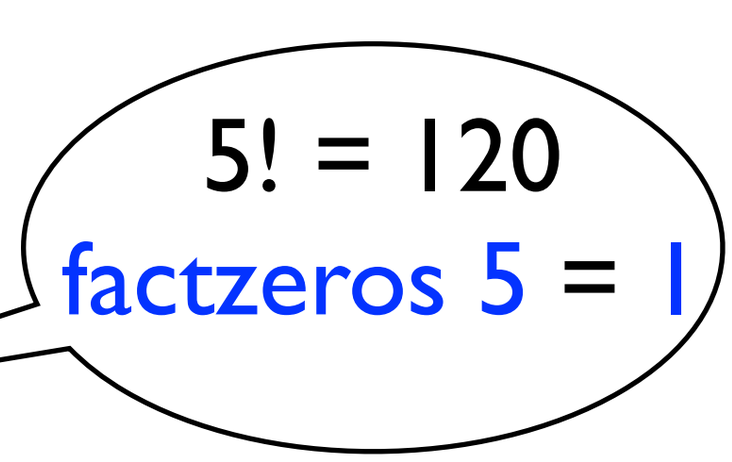
factzeros : int -> int

such that for all n>0,

factzeros n =

the number of zeros

on the end of n! (in decimal)



modular programs

- Design program as collection of small *units*
  - manageable
  - easy to maintain
- Give an *interface* for each unit
  - other units rely *only* on interface
- Separate development
- Information hiding...

language support

- **Signatures**
  - interfaces
- **Structures**
  - implementations (or “units”)
- **Functors**
  - ways to *combine* structures...

signatures

```
signature ARITH =
  sig
    type integer
    val rep : int -> integer
    val display : integer -> string
    val add : integer * integer -> integer
    val mult : integer * integer -> integer
  end
```

A type named **integer**

A function **rep : int -> integer**

A function **display : integer -> string**

A function **add : integer * integer -> integer**

A function **mult : integer * integer -> integer**

just an interface

- *Declaring* this signature doesn't *create* any types or values
- An *implementation* of this signature is called a *structure*
- *Defining* a structure with this signature creates the types and values
- A signature can have many implementations

implementing ARITH

Build a structure with definitions for

```
type integer = ...
fun rep (n:int):integer = ...
fun display (n:integer):string = ...
fun add(m:integer, n:integer):integer = ...
fun mult(m:integer, n:integer):integer = ...
```

Ints

```
structure Ints =
struct
    type integer = int
    fun rep (n:int):integer = n
    fun display (n:integer):string = Int.toString n
    val add : integer * integer -> integer = (op +)
    val mult : integer * integer -> integer = (op * )
end
```

An implementation of Arith
in which integer is the type int

result

ML says

```
structure Ints :
  sig
    type integer = int
    val rep : int -> integer
    val display : integer -> string
    val add : integer * integer -> integer
    val mult : integer * integer -> integer
end
```

implements

Ints *implements* ARITH

by defining

- types
- values of appropriate types
- consistent with the signature

```
structure Ints :
  sig
    type integer = int
    val rep : int -> integer
    val display : integer -> string
    val add : integer * integer -> integer
    val mult : integer * integer -> integer
end
```

```
signature ARITH =
 sig
    type integer
    val rep : int -> integer
    val display : integer -> string
    val add : integer * integer -> integer
    val mult : integer * integer -> integer
end
```

what can go wrong

- **structure** Wheel : CIRCLE = Square;
> Error: unmatched value specification: radius

ascription

Ascribing a signature to a structure

```
structure Ints : ARITH =
struct
    type integer = int
    fun rep (n:int):integer = n
    fun display (n:integer):string = Int.toString n
    val add:integer * integer -> integer = (op +)
    val mult:integer * integer -> integer = (op * )
end
```

ML says

```
structure Ints : ARITH
```

open

```
- open Ints;

opening Ints
  type integer = int
  val rep : int -> integer
  val add : integer * integer -> integer
  val mult : integer * integer -> integer
  val display : integer -> string
```

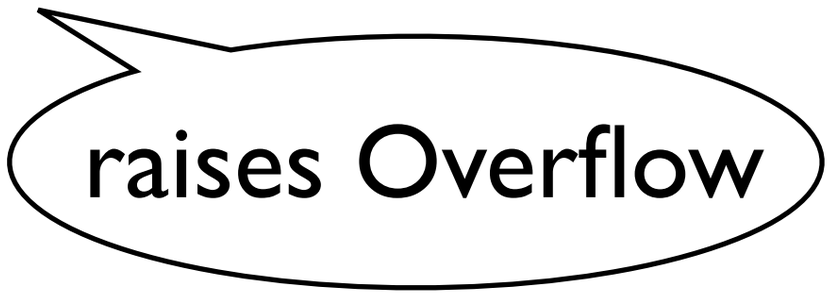
Opening a structure
reveals the types and values
declared in its signature

using Ints

```
open Ints

fun fact(n:int):integer =
    if n=0 then rep 1 else mult(rep n, fact(n-1))
                : integer          : integer
```

fact 100



qualified names

- String.compare : string * string -> order
- Ints.display : Ints.integer -> string

Used to *disambiguate* when there
are several structures
with the same signature

using Ints

```
fun fact(n:int): Ints.integer =
    if n=0 then Ints.rep 1
           else Ints.mult(Ints.rep n, fact(n-1));
```

```
fact 100
```

Without opening a structure
you'll need to use *qualified names*
to access its types and values

transparency

We ascribed the signature ***transparently*** using **:**

```
structure Ints : ARITH = ...
open Ints

fun fact(n:int):integer =
    if n=0 then 1 else mult(rep n, fact(n-1))
                  : int          : integer
```

this is *allowed*,
but likely a mistake

ML allows users to *exploit* the
fact that type integer is int

opacity

We can ascribe the signature ***opaquely*** using **:>**

```
structure Ints :> ARITH = ...
open Ints

fun fact(n:int):integer =
    if n=0 then 1 else mult(rep n, fact(n-1))
                  : int          : integer

Error: types of if branches do not agree
```

ML *prevents* users from *exploiting*
the fact that integer is int

implementing ARITH

- Build a structure with definitions for

  type integer =
  fun rep (n:int):integer =
  fun display (n:integer):string =
  fun add(m:integer, n:integer):integer =
  fun mult(m:integer, n:integer):integer =

  Ints isn't the only way...

decimal digits

```
structure Dec : ARITH =
 struct
   type digit = int
   type integer = digit list

   fun rep 0 = [ ]
     |   rep n = (n mod 10) :: rep(n div 10)

   fun display [ ] = "0"
     |   display L = foldl (fn (d, s) => Int.toString d ^ s) "" L
```

.....

decimal digits

.... continued ...

```
(* carry : digit * integer -> integer *)
  fun carry (0, ps) = ps
    |   carry(c, [ ]) = [c]
    |   carry (c, p::ps) =
          ((p+c) mod 10) :: carry ((p+c) div 10, ps)

  fun add ([ ], qs) = qs
    |   add (ps, [ ]) = ps
    |   add (p::ps, q::qs) =
          ((p+q) mod 10) :: carry ((p+q) div 10, add(ps, qs))
```

decimal digits

.... continued ....

```
(* times : digit * integer -> integer *)
fun times (0, qs) = [ ]
 |    times (p, [ ]) = [ ]
 |    times (p, q::qs) =
      ((p * q) mod 10) :: carry ((p * q) div 10, times(p, qs))


fun mult ([ ], _) = [ ]
 |    mult(_, [ ]) = [ ]
 |    mult (p::ps, qs) = add (times(p, qs), 0 :: mult (ps, qs))
 end
```

all together

```
structure Dec : ARITH =
 struct
   type digit = int
   type integer = digit list

   fun rep 0 = [ ] | rep n = (n mod 10) :: rep (n div 10);

   fun carry (0, ps) = ps | ...

   fun add ([ ], qs) = qs | ....

   fun times(0, qs) = [ ] | ....

   fun mult([ ], _) = [ ] | ....
   fun display L = foldl (fn (d, s) => Int.toString d ^ s) "" L
end;
```

implements

- Dec does *implement* the signature ARITH
- It's OK to define extra data, such as

  carry : digit * integer -> integer

  times : digit * integer -> integer
- These are not in the ascribed signature, so not visible outside the structure
- The type **integer** is **int list**
- The only *relevant* lists contain *decimal digits*

abstract data type

- The signature for ARITH specifies an ***abstract data type***
- A type **integer**
- Equipped with basic operations
  - **rep** : int -> integer — *initialize*
  - **add**, **mult** : integer * integer -> integer — *combine*
  - **display** : integer -> string — *examine*

correctness

- Dec implements non-negative integers
  in a way that is ***faithful*** to *standard arithmetic*
    - Every non-negative value is ***rep****resentable*
    - **add** implements +
    - **mult** implements *

invariant

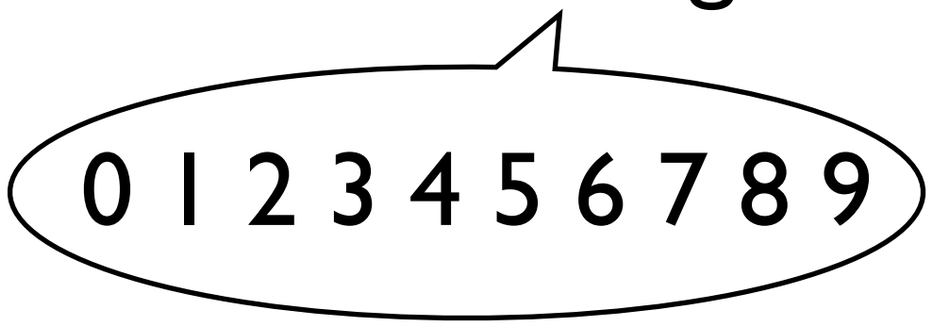
- To prove correctness we need to introduce a ***representation invariant***

inv_{10} : int list -> bool

$inv_{10}(L)$ = true

iff

every item in L is a decimal digit



abstraction

- And we need an ***abstraction function***

$eval_{10}$: integer -> int

For all L : int list,
if inv_{10}(L) = true, then
$eval_{10}$ L = the value represented by L

non-negative

inv_{10}

(* inv_{10} : int list -> bool *)
fun inv_{10} [ ] = **true**
 | inv_{10} (d::L) = (0 <= d **andalso** d <= 9) **andalso** inv_{10} L

$eval_{10}$

(* $eval_{10}$: int list -> int *)
fun $eval_{10}$ [ ] = 0
 | $eval_{10}$ (d::L) = d + 10 * $eval_{10}$(L)

termination

- For all L:int list, inv_{10}(L) terminates
- For all L:int list, $eval_{10}$(L) terminates
- For all n≥0, rep(n) terminates

If ps and qs satisfy **inv_{10}**, then
add(ps, qs) terminates
& mult(ps, qs) terminates

correctness

- Every **integer** value built from **rep**, **add**, **mult** satisfies $\mathbf{inv_{10}}$
- For all $n \geq 0$, **rep(n)** satisfies $\mathbf{inv_{10}}$
- For all **ps**, **qs** : int list,

  if **ps** and **qs** satisfy $\mathbf{inv_{10}}$

  so do **add(ps, qs)** and **mult(ps, qs)**

correctness

- For all **ps**, **qs** : int list,

  if **ps** and **qs** satisfy $\mathbf{inv_{10}}$ then

  eval(**add(ps, qs)**) = (eval **ps**) + (eval **qs**)

  eval(**mult(ps, qs)**) = (eval **ps**) * (eval **qs**)

using Dec

```
open Dec;

fun fact(n:int):integer =
   if n=0 then rep 1 else mult(rep n, fact(n-1))

(* fact : int -> integer *)

(* REQUIRES n >= 0  *)

(* ENSURES fact(n) returns an int list representing n!  *)
(* ENSURES  eval(fact(n)) = n!  *)
```

results

display(fact 100) =
"933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000"

24 trailing zeros

problem of the day

- Any answers?

factzeros 100 = 24

```
fun loop i =
    if i<1 then 0 else i + loop (i div 5)

fun factzeros n = loop (n div 5)
```