MIT study says if you get more Z’s, you’ll get more A’s
Midterm Exam I

• In class, Thursday October 10
• Write with pen, NOT pencil
• You can refer to ONE page of written notes
• If you have permission for extra time, please schedule with Disability Services (and check with me)
and now...

- Generalizing the **subset-sum** problem
- Developing **specs** and **code** together
- Another lesson in program design
making change

• Given a non-negative integer n, a list of positive integers L, and a constraint \( p : \text{int list} \rightarrow \text{bool} \)

• Is there a sublist of L that satisfies \( p \) and adds up to n?
using folds

- Choose an appropriate base value, a sensible combining function, and pay attention to combination order

```haskell
fun sublists L = foldr (fn (x, S) => S @ (map (fn A => x::A) S)) [[]] L

val sublists = fn : 'a list -> 'a list list
- sublists [1,2,3];
val it = [[]],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3] : int list list
```

What happens with foldl instead of foldr?
badchange

badchange : int * int list -> (int list -> bool) -> bool

fun badchange (n, L) p =
    exists (fn A => sum A = n andalso p A) (sublists L)

A non-recursive function that returns a boolean

fun exists q = foldl (fn (x, t) => q x orelse t) false

fun sum A = foldl (op +) 0 A

sublists [1,2,3] = [[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]

badchange satisfies the spec
critique

badchange (300, [1,2,3,...,24]) (fn _ => true)

===========>* true

brute force search

• generates the list of all sublists \(2^{24}\)

• tests them sequentially

... only the final sublist will work!

\[1 + 2 + \ldots + 24 = 300\]
specification

“can we make change for \((n, L)\) that satisfies \(p\)?”

\[
\text{change} : \text{int} \times \text{int list} \rightarrow (\text{int list} \rightarrow \text{bool}) \rightarrow \text{bool}
\]

**REQUIRES** \(p\) is total, \(n \geq 0\), \(L\) a positive list

**ENSURES** \(\text{change } (n, L) p = \text{true}\)

if there is a sublist \(A\) of \(L\) with

\[
\text{sum } A = n \text{ and } p A = \text{true}
\]

\(\text{change } (n, L) p = \text{false}\), otherwise

+ must be faster!
a better strategy

Avoid building the list of sublists

• only call p on sublists with the correct sum

Deal with special cases first

• n = 0

• n > 0, L = []

For n > 0, L = x::R, use **recursion**...

*the spec suggests this might be feasible!*
A recursive function that returns a boolean

```scala
fun change (0, L) p = p [ ]
| change (n, [ ]) p = false
| change (n, x::R) p =
  if x <= n
  then change (n-x, R) (fn A => p(x::A))
  orelse
  change (n, R) p
else change (n, R) p
```

change (300, [1,2,3,...,24]) (fn _ => true) => true
equivalently

fun change (0, L) p =  p [ ]
| change (n, [ ]) p =  false
| change (n, x::R) p =
  if x <= n
  then
    case change (n-x, R) (fn A => p(x::A)) of
      true  => true
      false => change (n, R) p
  else change (n, R) p

(if you don’t like nested if-then-else)
fun change (0, L) p = p [ ]
| change (n, [ ]) p = false
| change (n, x::R) p =
  if x <= n
  then change (n-x, R) (fn A => p(x::A))
  orelse
  change (n, R) p
else change (n, R) p

REQUIRES p is total, n ≥ 0, L a list of positive integers

MUST CHECK
if requirements hold for (n, L, p) and change (n, L) p calls change (n′,L′) p′
then requirements hold for (n′, L′, p′)
correctness?

For all positive integer lists $L$, $n \geq 0$, and total functions $p : \text{int list} \to \text{bool}$,

\[
\text{change} \ (n, L) \ p = \text{true}
\quad \text{if there is a sublist } A \text{ of } L \text{ with}
\quad \text{sum } A = n \text{ and } p \ A = \text{true}
\]

\[
\text{change} \ (n, L) \ p = \text{false}, \text{ otherwise}
\]

- PROOF: induction on $L$
change

change : int * int list -> (int list -> bool) -> bool

fun change (0, L) p = p [ ]
| change (n, [ ]) p = false
| change (n, x::R) p =
  if x <= n
  then change (n-x, R) (fn A => p(x::A))
  orelse
    change (n, R) p
else change (n, R) p
examples

change (10, [5,2,5]) (fn _ => true) 
= true

change (210, [1,2,3,...,20]) (fn _ => true) 
=>* true (FAST!)

change (10, [10,5,2,5]) (fn A => length(A)>1) 
= true

change (10, [10,5,2]) (fn A => length(A)>1) 
= false
the right question?

change : int * int list -> (int list -> bool) -> bool

Is there a sublist of that ...?

What is it?

YES!
You didn’t ask
boolean blindness

- By returning a *truth value* we only get a small amount of information *(true or false)*

- From the context, we know this tells us “if it is *possible* to make change...”

- But what if we want *more* information?
  - “a *way* to make change, if there is one”

needed: a recursive function that computes a suitable sublist, if there is one

plan: use result type *(int list)* option
mkchange

A recursive function that returns an (int list) option

mkchange : int * int list -> (int list -> bool) -> int list option

REQUIRES p is total, n >= 0, L is a positive list

ENSURES mkchange (n, L) p = SOME A,
    where A is a sublist of L
    with sum A = n and p A = true,
    if there is one

        mkchange (n, L) p = NONE, otherwise
fun mkchange (0, L) p = 
  if p [ ] then SOME [ ] else NONE
| mkchange (n, [ ]) p = NONE
| mkchange (n, x::R) p = 
  if x <= n
  then case mkchange (n-x, R) (fn A => p(x::A)) of
    SOME A => SOME (x::A)
    | NONE => mkchange (n, R) p
  else mkchange (n, R) p
correctness?

For all positive integer lists L, n ≥ 0, and total functions p : int list -> bool,

\[
\text{mkchange (n, L) p = } \begin{cases} 
\text{SOME A} \\
\text{NONE}
\end{cases}
\]

where A is a sublist of L with sum A = n and p A = true, if there is one

\[
\text{mkchange (n, L) p = NONE, otherwise}
\]

• PROOF: induction on L

(similar to the proof of change)
fun mkchange (0, L) p = 
  if p [] then SOME [] else NONE 
| mkchange (n, [ ]) p = NONE 
| mkchange (n, x::R) p = 
  if x <= n 
  then 
    case mkchange (n-x, R) (fn A => p(x::A)) of 
    SOME A => SOME (x::A) 
    NONE => mkchange (n, R) p 
  else 
    mkchange (n, R) p
The definitions have similar control flow, with pattern-matching based on the result type.
What if the boss decides he wants yet another type of “answer”

How could we avoid having to keep re-doing the code design all over again?
Design a very flexible function

Instead of returning a **bool** or **int list option**, use a type variable ‘a for the **answer** type *(to be chosen later)* and use **function** parameters s and k to implement success and **failure**

\[
s : \text{int list } \rightarrow \ 'a \quad \text{call } s \text{ if you find a solution}
\]

\[
k : \text{unit } \rightarrow \ 'a \quad \text{call } k \text{ if you fail}
\]
more flexible

changer : int * int list -> (int list -> bool)
  -> (int list -> 'a) -> (unit -> 'a) -> 'a

changer (n, L) p s k : t
  : int list -> t
  : unit -> t

this recipe works
for any type t
of “answers”
success

\[ s : \text{int list} \to t \] is a \textit{success} continuation
failure

\[ k : \text{unit} \rightarrow t \] is a \textit{failure} continuation
more flexible spec

changer : int * int list -> (int list -> bool)
    -> (int list -> 'a) -> (unit -> 'a) -> 'a

REQUIRES  n>=0, L a list of positive integers, p total

ENSURES  changer (n, L) p s k = s A
    where A is a sublist of L such that
        sum A = n and p A = true,
        if there is one

changer (n, L) p s k = k ()
    otherwise

success,
call s

fail,
call k
fun changer (0, L) p s k = 
  if p [ ] then s [ ] else k( )
| changer (n, [ ]) p s k = k( )
| changer (n, x::R) p s k = 
  if x <= n 
  then 
    changer (n-x, R) 
    (fn A => p(x::A)) 
    (fn A => s(x::A)) 
    (fn ( ) => changer (n, R) p s k) 
  else 
    changer (n, R) p s k
examples

changer (12, [1,1,1,1,1,1,1,1,2,4,6])
  (fn _ => true) SOME (fn () => NONE);

val it = SOME [1,1,1,1,1,1,1,2,4] : int list option

changer (12, [1,1,1,1,1,1,1,1,2,4,6])
  (fn A => length A < 4) SOME (fn () => NONE);

val it = SOME [2,4,6] : int list option
For all lists $L$ of positive integers, $n \geq 0$, total functions $p : \text{int list} \to \text{bool}$, all types $t$ and all $s : \text{int list} \to t$, $k : \text{unit} \to t$

$$\text{changer} \ (n, L) \ p \ s \ k = s \ A$$

where $A$ is a sublist of $L$ with

$$\text{sum} \ A = n \text{ and } p \ A = \text{true},$$

if there is one

$$\text{changer} \ (n, L) \ p \ s \ k = k(\ ), \text{otherwise}$$

• PROOF Use induction on structure of $L$

(similar to the proofs for...)
We make argument assumptions explicit

We indicate flexibility of result type

We use accurate math notation

The requirements are clear and precise

“For all types t, all total functions p,

...yada yada yada”
utility

changer : int * int list -> (int list -> bool) -> (int list -> 'a) -> (unit -> 'a) -> 'a

change : int * int list -> (int list -> bool) -> bool

fun change (n, L) p =
  changer (n, L) p (fn _ => true) (fn () => false)

mkchange : int * int list -> (int list -> bool) -> int list option

fun mkchange (n, L) p =
  changer (n, L) p SOME (fn () => NONE)
robust

- There's no need to re-do the code if the manager decides to change the spec again, e.g. by *giving an extra penny*

```plaintext
fun generous_change (n, L) p =
changer (n, L) p (fn A => 1::A) (fn () => [1])

- generous_change (12, [5,5,2,42]) (fn _ => true)

val it = [1,5,5,2] : int list
```
lessons

• May be better to solve a more general problem
  • suitable for recursion

• Propagate enough information
  • a boolean may not be sufficient

• Make sure that requirements hold for recursive calls…
  so you can use inductive hypothesis that recursive calls work correctly
next

- Using *continuations* to implement *backtracking*