so far

• Expressions have types
• Types have sets of values
• Expression evaluation
  • returns a value, of the correct type
  • or loops forever

well-typed expressions don’t cause stupid type errors
runtime errors

Even a well-typed expression may cause a runtime error

- fact 100
- 42 div 0
- hd (allqueens 3)

Overflow
Div
Empty
runtime errors

Even a well-typed expression may cause a runtime error

- fact 100
- 42 div 0
- hd (allqueens 3)

hd [42 div (length(allqueens 3) * fact 100)]
runtime errors

Even a *well-typed* expression may cause a runtime error

- fact 100  
  - Overflow
- 42 div 0  
  - Div
- hd (allqueens 3)  
  - Empty

hd [42 div (length(allqueens 3) * fact 100)]
  =>* uncaught exception Overflow
it’s ok

• Type analysis is based on syntax
  • does not evaluate expressions

• Can't expect to prevent all runtime errors just by looking at syntax

• And type-checking still prevents the most common mistakes…

• …for everything else, there’s exceptions
exceptions

• ML has built-in and user-definable exceptions
  • built-ins include $Div$, $Overflow$, $Match$
• Exceptions can be declared, raised, handled
  • Very flexible
  • Simple scope rules
• Exceptions co-exist nicely with type discipline
however...

• The potential for exceptions means we must revise our foundational definitions!

• Evaluation

• Equality

• Referential transparency
• An expression evaluation will either return a value, of the correct type or loop forever or raise an exception
equality

- Expressions of type `int` are equal iff both evaluate, to the same value or both fail to terminate or both raise the same exception.
Expressions of type \( t \rightarrow t' \) are equal iff both evaluate, to equal values or both fail to terminate or both raise the same exception.

Values \( f \) and \( g \) are equal iff for all \( x, y : t \), \( x = y \) implies \( f(x) = g(y) \).
transparency

• Safe substitution for equal sub-expressions

\[ \text{If } e_1 = e_2 \text{ then } E[e_1] = E[e_2] \]
• Safe substitution for equal sub-expressions

\[ \text{If } e_1 = e_2 \text{ then } E[e_1] = E[e_2] \]

\[ 21 + 21 =_{\text{int}} 42, \]

so

\[ (\text{fn } x: \text{int} \Rightarrow 21 + 21) =_{\text{int} \rightarrow \text{int}} \text{fn } x: \text{int} \Rightarrow 42) \]
transparency

• Safe substitution for equal sub-expressions

\[
\text{If } e_1 = e_2 \quad \text{then } E[e_1] = E[e_2]
\]
transparency

• Safe substitution for equal sub-expressions

If \( e_1 = e_2 \) then \( E[e_1] = E[e_2] \)

2 + raise Foo =_{\text{int}} raise Foo,
so
\((\text{fn} \ x \Rightarrow 2 + \text{raise Foo}) =_{\text{int -> int}} (\text{fn} \ x \Rightarrow \text{raise Foo})\)
• Laws of equivalence need to be re-assessed
• So far we assumed expressions were \textit{pure} (evaluate to a value, or loop)
• All equivalences discussed so far hold for \textit{pure} expressions, but may fail for impure

\[
e_1 + e_2 = e_2 + e_1
\]

is valid when \( e_1, e_2 : \text{int} \) \text{ pure}

(\text{raise Foo}) + (\text{raise Bar}) \neq (\text{raise Bar}) + (\text{raise Foo})
declaring an exception

```python
exception Negative
exception Ring-ding-ding-ding-dingeringedinging
```

usually,
choose a short and appropriate name
OK to raise and handle Foo here
raising an exception

• In scope of an exception named Foo, the expression `raise Foo` causes a runtime error
  - `raise Foo;`
    uncaught exception Foo

• The expression `raise Foo` has type `'a`
  and doesn’t evaluate to a proper value
raising an exception

• In scope of an exception named Foo, the expression `raise Foo` causes a runtime error
  - `raise Foo;
    uncaught exception Foo`

• The expression `raise Foo` has type `'a`
  and doesn’t evaluate to a proper value

`42 + raise Foo`
raising an exception

• In scope of an exception named Foo, the expression `raise Foo` causes a runtime error
  
  - `raise Foo;`  
  uncaught exception Foo

• The expression `raise Foo` has type `'a` and doesn’t evaluate to a proper value

```
42 + raise Foo  
(fn x:int => 0) (raise Foo)
```
gcd : int * int -> int

(* REQUIRES x>0 & y>0 *)
(* ENSURES gcd(x, y) = the g.c.d. of x and y. *)
fun gcd (x, y) =
  case Int.compare(x, y) of
   LESS => gcd(x, y-x)
   | EQUAL => x
   | GREATER => gcd(x-y, y)
gcd : int * int -> int

(* REQUIRES x>0 & y>0 *)
(* ENSURES gcd(x, y) = the g.c.d. of x and y. *)
fun gcd (x, y) =
    case Int.compare(x, y) of
        LESS    => gcd(x, y-x)
        | EQUAL  => x
        | GREATER => gcd(x-y, y)

gcd(10,24) = gcd(10,14)
        = gcd(10, 4)
        = gcd(6, 4)
        = gcd(2,4)
        = gcd(2,2)
        = 2
gcd : int * int -> int

(* REQUIRES x>0 & y>0 *)
(* ENSURES gcd(x, y) = the g.c.d. of x and y. *)

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

(* REQUIRES x>0 & y>0 *)
(* ENSURES gcd(x, y) = the g.c.d. of x and y. *)

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

gcd(1, 0) =>* gcd(1-0, 0) =>* gcd(1, 0) =>* ...
**gcd** : int * int -> int

(* REQUIRES x>0 & y>0 *)

(* ENSURES gcd(x, y) = the g.c.d. of x and y. *)

fun gcd (x, y) =

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

gcd(1, 0) =>* gcd(1-0, 0) =>* gcd(1, 0) =>* ...

*will loop forever*
gcd : int * int -> int

(* REQUIRES x>0 & y>0 *)
(* ENSURES gcd(x, y) = the g.c.d. of x and y. *)

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

  will loop forever

  gcd(1, 0) =>* gcd(1-0, 0) =>* gcd(1, 0) =>* ...

  gcd(1, ~1) =>* gcd(2, ~1) =>* gcd(3, ~1) =>*...
**gcd : int * int -> int**

(* REQUIRES x>0 & y>0 *)
(* ENSURES gcd(x, y) = the g.c.d. of x and y. *)

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

gcd(1, 0) =>* gcd(1-0, 0) =>* gcd(1, 0) =>* ...

gcd(1, ~1) =>* gcd(2, ~1) =>* gcd(3, ~1) =>* ...

*will loop forever*  
*will raise Overflow*
gcd : int * int -> int

(* REQUIRES x>0 & y>0 *)
(* ENSURES gcd(x, y) = the g.c.d. of x and y. *)

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

gcd(1, 0) =>* gcd(1-0, 0) =>* gcd(1, 0) =>*
will loop forever

gcd(1, ~1) =>* gcd(2, ~1) =>* gcd(3, ~1) =>*
- uncaught exception Overflow [overflow]
  raised at: <file stdIn>

will raise Overflow
GCD : int * int -> int

(* REQUIRES true *)
(* ENSURES GCD(x,y) = the g.c.d of x and y if x > 0 and y > 0, *)
(* GCD(x,y) = raise NotPositive if x ≤ 0 or y ≤ 0. *)

exception NotPositive;

fun GCD (x, y) =
  if (x <= 0 orelse y <= 0) then raise NotPositive else
  case Int.compare(x, y) of
    LESS => GCD(x, y-x)
    | EQUAL => x
    | GREATER => GCD(x-y, x)
GCD : int * int -> int

(* REQUIRES true *)
(* ENSURES GCD(x,y) = the g.c.d of x and y if x > 0 and y > 0, *)
(* GCD(x,y) = raise NotPositive if x ≤ 0 or y ≤ 0. *)

exception NotPositive;

fun GCD (x, y) =
    if (x <= 0 orelse y <= 0) then raise NotPositive else
    case Int.compare(x,y) of
        LESS    => GCD(x, y-x)
        | EQUAL   => x
        | GREATER => GCD(x-y, x)

    GCD(10, 24) =>* 2
    GCD(1, 0)   =>* raise NotPositive
    GCD(1, ~1)  =>* raise NotPositive
GCD : int * int -> int

(* REQUIRES true *)
(* ENSURES GCD(x,y) = the g.c.d of x and y if x > 0 and y > 0, *)
(* GCD(x,y) = raise NotPositive if x ≤ 0 or y ≤ 0. *)

exception NotPositive;

fun GCD (x, y) =
  if (x <= 0 orelse y <= 0) then raise NotPositive else
  case Int.compare(x,y) of
    LESS    => GCD(x, y-x)
  | EQUAL   => x
  | GREATER => GCD(x-y, x)

GCD(10, 24) =>* 2
GCD(1, 0)   =>* raise NotPositive
GCD(1, ~1)  =>* raise NotPositive
**GCD : int * int -> int**

(* REQUIRES true *)

(* ENSURES GCD(x,y) = the g.c.d of x and y if x > 0 and y > 0, *)

(* ENSURES GCD(x,y) = raise NotPositive if x ≤ 0 or y ≤ 0. *)

**exception NotPositive**

**fun gcd (x, y) =**

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

**fun GCD (x, y) =**

```haskell
if (x > 0 andalso y > 0) then gcd(x,y) else raise NotPositive
```
exception NotPositive

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

fun GCD (x, y) =
    if (x > 0 andalso y > 0) then gcd(x,y) else raise NotPositive

    GCD(10, 24) =>* 2
    GCD(1, 0)   =>* raise NotPositive
    GCD(1, ~1)  =>* raise NotPositive
GCD': int * int -> int

exception NotPositive

local
  fun gcd (m,n) =
    case Int.compare(m,n) of
    LESS    => gcd(m, n-m)
    | EQUAL   => m
    | GREATER => gcd(m-n, n)
  in
  fun GCD' (x, y) =
    if x>0 andalso y>0 then gcd(x, y)
    else raise NotPositive
  end;

even better: the dangerous gcd function is not available outside
GCD = GCD’

GCD : int * int -> int
GCD’ : int * int -> int

are extensionally equal, because:
for all integer values x and y,

EITHER x>0 & y>0, and
   GCD(x,y) and GCD’(x,y) both evaluate
to the g.c.d of x and y,

OR not(x>0 & y>0), and
   GCD(x,y) and GCD’(x,y) both raise NotPositive
handling

\[ e_1 \text{ handle } \langle \text{exn name} \rangle \Rightarrow e_2 \]

• If \( e_1 \) and \( e_2 \) have type \( t \),
so does \( e_1 \text{ handle } \text{Foo} \Rightarrow e_2 \)

• If \( e_1 \Rightarrow^* v \),
so does \( e_1 \text{ handle } \text{Foo} \Rightarrow e_2 \)

• If \( e_1 \) raises \( \text{Foo} \),
\( e_1 \text{ handle } \text{Foo} \Rightarrow e_2 \Rightarrow^* e_2 \)

• If \( e_1 \) raises \( \text{Bar} \),
so does \( e_1 \text{ handle } \text{Foo} \Rightarrow e_2 \)

• If \( e_1 \) loops, so does \( e_1 \text{ handle } \text{Foo} \Rightarrow e_2 \)
The scope of the handler for Foo in
\[ e \text{ handle } \text{Foo} \Rightarrow e' \]
is \( e \).

Can also combine handlers
\[ e \text{ handle } \text{Ringerding} \Rightarrow e_1' \]
\[ | \text{Hatee-hatee-ho} \Rightarrow e_2' \]
\[ | \text{Wa-pow-pow} \Rightarrow e_3' \]
\[ | _ \Rightarrow \text{raise NotFox} \]

\( (e, e_1', e_2', e_3' \text{ must have the same type}) \)
making change

- Given a list \( L \) of positive integers (coin sizes) and a non-negative integer \( a \) (an amount)
- Find a list of items from \( L \) that adds up to \( a \)
  - *can use coin sizes more than once*
idea

• If $a = 0$, make change with the empty list.
• If $a > 0$ and the coins list is $c::R$,
  – if $c > a$ you can’t use it;
  – otherwise use $c$ and make change for $a - c$.

“greedy”
change

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

fun change (_, 0) = [ ]
| change (c::R, a) =
  if a<c then change(R, a)
  else c :: (change(c::R, a-c))
results

stdIn:60.5-63.41 Warning: match nonexhaustive
   (_,0) => ...
   (c :: R,a) => ...

val change = fn : int list * int -> int list
results

stdIn:60.5-63.41 Warning: match nonexhaustive

    (_,0) => ...
    (c :: R,a) => ...

val change = fn : int list * int -> int list

- change ([2,3], 8);
  val it = [2,2,2,2] : int list

- change ([3,2], 8);
  val it = [3,3,2] : int list
results

stdin:60.5-63.41 Warning: match nonexhaustive

(_,0) => ...
(c :: R,a) => ...

val change = fn : int list * int -> int list

- change ([2,3], 8);
val it = [2,2,2,2] : int list

- change ([3,2], 8);
val it = [3,3,2] : int list

- change([ ], 42);
uncaught exception Match

Match usually means incomplete pattern-matching
results

stdin:60.5-63.41 Warning: match nonexhaustive
   (_,0) => ...
   (c :: R,a) => ...

val change = fn : int list * int -> int list

- change ([2,3], 8);
  val it = [2,2,2,2] : int list

- change ([3,2], 8);
  val it = [3,3,2] : int list

- change([ ], 42);
  uncaught exception Match

- change([2,3], 9);
  uncaught exception Match

Match usually means incomplete pattern-matching

should be [3,3,3]
style issues

- It’s bad to use a built-in exception like `Match` to signal a problem-specific error
  - `Match` is used for a specific purpose
- Instead declare and raise your own exception
- Give it a suggestive name like `Impossible`
style issues

• It’s bad to use a built-in exception like `Match` to signal a problem-specific error
  
  • `Match` is used for a specific purpose

• Instead declare and raise your own exception

• Give it a suggestive name like `Impossible`
style issues

• It’s bad to use a built-in exception like Match to signal a problem-specific error

  • Match is used for a specific purpose

• Instead declare and raise your own exception

• Give it a suggestive name like Impossible

NoChange
CannotMakeChange
style issues

• It’s bad to use a built-in exception like **Match** to signal a problem-specific error
  
  • **Match** is used for a specific purpose

• Instead declare and raise your own exception

• Give it a *suggestive* name like **Impossible**

  NoChange
  CannotMakeChange
  InsufficientFunds
idea 2

• If \( a = 0 \), make change with the empty list.

• If \( a > 0 \) & coins list is \( \text{c::R} \)
  – if \( c > a \) you can’t use it;
  – otherwise use \( c \) & make change for \( a - c \).

• If \( a > 0 \) & coins list is [ ], raise Impossible.
change 2

exception Impossible;

(* change : int list * int -> int list *)
fun change (_, 0) = []
  | change ([], _) = raise Impossible
  | change (c::R, a) =
    if a<c then change (R, a)
    else c :: (change (c::R, a-c))
val change = fn : int list * int -> int list

- change ([2,3], 8);
val it = [2,2,2,2] : int list
results

val change = fn : int list * int -> int list

- change ([2,3], 8);  
val it = [2,2,2,2] : int list

- change ([ ], 42);  
uncaught exception Impossible
results

val change = fn : int list * int -> int list

- change ([2,3], 8);
val it = [2,2,2,2] : int list

- change ([ ], 42);
uncaught exception Impossible

- change ([2,3], 9);
uncaught exception Impossible

should be [2,2,2,3]
idea 3

• If $a = 0$, make change with the empty list.

• If $a > 0$ & coins list is $c::R$
  – if $c > a$ you can’t use it;
  – otherwise *try to* use $c$ and make change for $a-c$; if this fails, *handle the exception* by making change for $a$ without $c$.

• If $a > 0$ & coins list is [ ], raise Impossible.
exception Impossible;

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

fun change (_, 0) = []

|    change ([ ], _) = raise Impossible
|    change (c::R, a) =
|      if a<c then change (R, a)
|      else c :: (change (c::R, a-c))
|        handle Impossible => change(R, a)
results

- change ([2,3], 8);
val it = [2,2,2,2] : int list
results

- change ([2,3], 8);
  val it = [2,2,2,2] : int list

- change([ ], 42);
uncaught exception Impossible
results

- change ([2,3], 8);
  val it = [2,2,2,2] : int list

- change([], 42);
  uncaught exception Impossible

- change ([2,3], 9);
  val it = [2,2,2,3] : int list
value equations

- The `change` definition yields these equations, for all values \( L, c, R, a \)

\(1\) change \((L, 0) = [\ ]\)

\(2\) change \(([\ ], a) = \text{raise Impossible}\)

\[\text{change } (c::R, a)+\]

\[\quad \text{if } a < c \text{ then change } (R, a)\]

\[\quad \text{else } c :: \text{(change } (c::R, a-c))\]

\[\quad \text{handle Impossible } \Rightarrow \text{change}(R, a)\]
value equations

• The change definition yields these equations, for all values $L, c, R, a$

(1) change $(L, 0) = [ ]$
(2) change $( [ ], a) = \text{raise Impossible}$
value equations

- The change definition yields these equations, for all values $L, c, R, a$

1. change $(L, 0) = []$
2. change $([], a) = \text{raise Impossible}$
3. change $(c::R, a) = \text{change } (R, a)$ if $a < c$
4. change $(c::R, a) = c :: (\text{change } (c::R, a-c))$
   - handle Impossible $\Rightarrow$ change$(R, a)$ if $a \geq c$
change([5,2], 6)

   = (5 :: change([5,2], 1))
      handle Impossible => change([2], 6)
change([5,2], 6)

= (5 :: change([5,2], 1))

    handle Impossible => change([2],6)

= (5 :: change([2], 1))

    handle Impossible => change([2],6)
change([5,2], 6)

= (5 :: change([5,2], 1))
    handle Impossible => change([2],6)
= (5 :: change([2], 1))
    handle Impossible => change([2],6)
= (5 :: change([ ], 1))
    handle Impossible => change([2],6)
change([5,2], 6)

= (5 :: change([5,2], 1))

    handle Impossible => change([2], 6)

= (5 :: change([2], 1))

    handle Impossible => change([2], 6)

= (5 :: change([ ], 1))

    handle Impossible => change([2], 6)

= (5 :: raise Impossible)

    handle Impossible => change([2], 6)
change([5,2], 6)

= (5 :: change([5,2], 1))
  handle Impossible => change([2],6)

= (5 :: change([2], 1))
  handle Impossible => change([2],6)

= (5 :: change([ ], 1))
  handle Impossible => change([2],6)

= (5 :: raise Impossible)
  handle Impossible => change([2],6)

= (raise Impossible)
  handle Impossible => change([2],6)
change([5,2], 6)

= (5 :: change([5,2], 1))
  handle Impossible => change([2], 6)
= (5 :: change([2], 1))
  handle Impossible => change([2], 6)
= (5 :: change([ ], 1))
  handle Impossible => change([2], 6)
= (5 :: raise Impossible)
  handle Impossible => change([2], 6)
= (raise Impossible)
  handle Impossible => change([2], 6)
= change([2], 6) = [2,2,2]
change spec

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

(* REQUIRES
   L is a list of positive integers & a ≥ 0 *)

(* ENSURES
   change(L, a) = a list of items from L
   with sum equal to a, if there is one;
   change(L, a) = raise Impossible, otherwise. *)
mkchange

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

fun mkchange (coins, a) =
   SOME (change (coins, a))
   handle Impossible => NONE
scope

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

fun change (_, 0) = [ ]
| change ([ ], _) = raise Impossible
| change (c::R, a) =
  if a<c then change (R, a)
  else c :: change (c::R, a-c)
  handle Impossible => change(R, a)

This function satisfies the given specification
This function satisfies the given specification
This function satisfies the given specification
(* change : int list * int -> int list *)

fun change (_, 0) = [ ]

|    change ([ ], _) = raise Impossible
|    change (c::R, a) = 

    if a<c then change (R, a)

    else c :: change (c::R, a-c)

    handle Impossible => change(R, a)

This function satisfies the given specification
scope

(* change : int list * int -> int list *)
fun change (_, 0) = [ ]
|    change ([ ], _) = raise Impossible
|    change (c::R, a) =
    if a<c then change (R, a)
    else c :: change (c::R, a-c)
    handle Impossible => change(R, a)

This function satisfies the given specification
different scope

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

fun change (_, 0) = [ ]
|    change ([ ], _) = raise Impossible
|    change (c::R, a) =

  (if a<c then change (R, a)
     else c :: change (c::R, a-c) )

  handle Impossible => change(R, a)
different scope

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

fun change (_, 0) = []
| change ([ ], _) = raise Impossible
| change (c::R, a) =

(if a<c then change (R, a)
  else c :: change (c::R, a-c) )

handle Impossible => change(R, a)
different scope

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

fun change (_, 0) = [ ]
| change ([ ], _) = raise Impossible
| change (c::R, a) =

(if a<c then change (R, a)
   else c :: change (c::R, a-c) )

handle Impossible => change(R, a)

This function also satisfies the specification,
but is less efficient!
different scope

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

fun change (_, 0) = []
| change ([ ], _) = raise Impossible
| change (c::R, a) =
  (if a<c then change (R, a)
   else c :: change (c::R, a-c))
  handle Impossible => change(R, a)

This function also satisfies the specification, but is less efficient!
This function also satisfies the specification, but is less efficient!
different scope

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

fun change (_, 0) = []

|    change ([ ], _) = raise Impossible
|    change (c::R, a) =

(if a < c then change (R, a)

else c :: change (c::R, a-c) )

handle Impossible => change(R, a)

This function also satisfies the specification, but is less efficient!