Course Philosophy

Computation is Functional.

Programming is an explanatory linguistic process.
Computation is Functional

values : types

expressions

Functions map values to values
Imperative vs. Functional

**Command**
- executed
- has an effect

\[ x := 5 \] (state)

**Expression**
- evaluated
- no effect

\[ 3 + 4 \] (value)
Programming as Explanation

Problem statement

- high expectation to explain precisely & concisely
- invariants
- specifications
- proofs of correctness
- code

Analyze, Decompose & Fit, Prove
Parallelism

\[
\begin{align*}
< 1, 0, 0, 1, 1, 1 > & \Rightarrow 3, \\
< 1, 0, 1, 1, 0, 0 > & \Rightarrow 2, \\
\end{align*}
\]
Parallelism

sum : int sequence → int
type row = int sequence
type room = row sequence

fun count (class : room) : int =
    sum (map sum sum class)
Parallelism

• **Work:**
  - Sequential Computation
  - Total sequential time; number of operations

• **Span:**
  - Parallel Computation
  - How long would it take if one could have as many processors as one wants; length of longest critical path
Defining ML (Effect-Free Fragment)

- Types $t$
- Expressions $e$
- Values $v$ (subset of expressions)
Examples:

\[(3 + 4) \times 2\]

\[7 \times 2\]

\[14\]

\[(3 + 4) \times (2 + 1)\]

\[21\]
"the " ^ "walrus"

1

⇒ "the walrus"

The expression "the " ^ "walrus" reduces to the value "the walrus".

It has type string.
"the walrus" + 1

⇒ ??

The expression
"the walrus" + 1
does not have a type
and it does not reduce
to a value.
Types

A *type* is a *prediction* about the kind of value an expression will have if it winds up reducing to a value.

An expression is *well-typed* if it has at least one type, and *ill-typed* otherwise.

(We may also say that an expression *type-checks*, meaning that it is well-typed.)
Every well-formed ML expression e

- has a type \( t \), written as \( e : t \)
- may have a value \( v \), written as \( e \rightarrow v \).
- may have an effect (not for our effect-free fragment)

Example: \((3+4) \times 2 : \text{int}\)

\((3 + 4) \times 2 \rightarrow 14\)
Integers, Expressions

Type \texttt{int}

Values \ldots, \ldots, 1, 0, 1, \ldots,

that is, every integer \( n \).

Expressions \( e_1 + e_2, \ e_1 - e_2, \ e_1 * e_2, \ e_1 \div e_2, \ e_1 \mod e_2 \), \( \ldots \), etc.

Example: \( \sim 4 \times 3 \)
Integers, Typing

Typing Rules

- \( n : \text{int} \)
- \( e_1 + e_2 : \text{int} \)

if \( e_1 : \text{int} \) and \( e_2 : \text{int} \)

similar for other operations.

Example:

\[
(3 + 4) * 2 : \text{int}
\]

Why?

\[
3 + 4 : \text{int} \quad \text{and} \quad 2 : \text{int}
\]

Why?

\[
3 : \text{int} \quad \text{and} \quad 4 : \text{int}
\]
Integers, Evaluation

Evaluation Rules

• \( e_1 + e_2 \xrightarrow{1} e'_1 + e_2 \) if \( e_1 \xrightarrow{1} e'_1 \)

• \( n_1 + e_2 \xrightarrow{1} n_1 + e'_2 \) if \( e_2 \xrightarrow{1} e'_2 \)

• \( n_1 + n_2 \xrightarrow{1} n \),

with \( n \) the sum of the integer values \( n_1 \) and \( n_2 \).
Example of a well-typed expression with no value

5 div 0 : int
5 \text{ div } 0 : \text{ int}

because \ 5 : \text{ int} \quad \& \quad 0 : \text{ int}

and because \text{ div} \text{ expects two \text{ int}s and returns an \text{ int}}.

However, \ 5 \text{ div } 0 \text{ does not reduce to a value.}
Notation Recap

\( e : t \)  "\( e \) has type \( t \)"

\( e \Rightarrow e' \)  "\( e \) reduces to \( e' \)"

\( e \rightarrow v \)  "\( e \) evaluates to \( v \)"
Extensional Equivalence

\[ \sim \]


\[ \equiv \]

An equivalence relation on expressions (of the same type).
Extensinal Equivalence

• Functional programs are *referentially transparent*
  – The *value* of an expression depends only on the *values* of its sub-expressions
  – The *type* of an expression depends only on the *types* of its sub-expressions

• Expressions are *extensionally equivalent* if they both reduce to the same value, or both raise the same exception, or both loop forever.

• Functions are *extensionally equivalent* if they map equivalent arguments to equivalent results.
  – *safe substitution* for equivalent code

• Examples:
  – 21 + 21 is equivalent to 7 * 6 is equivalent to 42
  – [2,7,6] is equivalent to [1+1, 2+5, 3+3]
  – (fn x => x+x) is equivalent to (fn y => 2*y)

  – In proofs, use ≈ for “equivalent”, e.g., [2, 7] ≈ [1+1, 2+5].
Types in ML

Basic types:
- int, real, bool, char, string

Constructed types:
- product types
- function types
- user-defined types
You will learn how to extract components using pattern matching
Typing Rules

\[ (e_1, e_2) : t_1 \times t_2 \]

if \( e_1 : t_1 \) and \( e_2 : t_2 \)

Example: \((3+4, \text{true}) : \text{int} \times \text{bool}\)
Evaluation Rules

- \((e_1, e_2) \xrightarrow{1} (e'_1, e_2)\) if \(e_1 \xrightarrow{1} e'_1\)

- \((v_1, e_2) \xrightarrow{1} (v_1, e'_2)\) if \(e_2 \xrightarrow{1} e'_2\)
Functions

In math, one talks about a function $f$ mapping between spaces $X$ and $Y,$

$$f : X \to Y$$

In SML, we will do the same, with $X$ and $Y$ being types.

**Issue:** Computationally, a function may not always return a value. That complicates checking equivalence.

**Definition:** A function $f$ is **total** if $f(x)$ returns a value for all values $x$ in $X.$
Sample Function Code

(* square : int -> int

  REQUIRES: true
  ENSURES: square(x) evaluates to x * x

  *)

fun square (x:int) : int = x * x

(keyword name argument result body of function
  name & type type)

(* Testcases: *)

val 0 = square 0
val 49 = square 7
val 81 = square (~9)
Declarations

Environments

Scope
Declaration

val pi : real = 3.14

Introduces binding of 3.14 to pi (sometimes written \[3.14/\pi\])

Lexically statically scoped.
val x : int = 8 - 5
val y : int = x + 1
val x : int = 10
val z : int = x + 1

[3/x]
[4/y]
[10/x]
[11/z]

Second binding of x
Shadows first binding.
First binding has been shadowed.
Local Declarations

let ... in ... end

let
  val m : int = 3
  val n : int = m * m
in
end

m + n

This is an expression.
What type does it have? int
What value? 12
Local Declarations

val k : int = 4

let val k : real = 3.0 in k * k end

\[ \rightarrow 9.0 : \text{real} \]

k \leftarrow \text{Type? Value?}

\[ \rightarrow 4 : \text{int} \]
Concrete Type Def

```plaintext
type float = real

type point = float*float

val p : point = (1.0, 2.6)
```
Course Tasks

- Assignments 35%
- Labs 10%
- Midterm 1 15%
- Midterm 2 15%
- Final 25%

Roughly one assignment per week, one lab per week.
Collaboration

Be sure to read the course and university webpages regarding academic integrity.