15-150
Fall 2024

Dilsun Kaynar

LECTURE 1

Introduction,Philosophy, Some Basics

About 15-150

Instructors: Stephanie Balzer, Dilsun Kaynar

19 **TAs**

http://www.cs.cmu.edu/~15150/

We are on Canvas!

Today

- Organization of the course
- Philosophy of the course
- Basics of types, values, expressions in SML

Course tasks

- **Assignments** 40%
- **Labs** 10%
- **Midterm 1** 15% (Sep 26)
- **Midterm 2** 15% (Nov 7)
- **Final** 20%

Collaboration policy

- Make sure to read and understand the policy for this semester

Extra help

- Office Hours by TAs
- Instructors available by appointment
- Student Academic Success Center
  - Drop-in Tutoring
    - Wednesdays POS 280
  - 1-on-1 tutoring by appointment

Course philosophy

- **Computation** is functional.
- **Programming** is an explanatory linguistic process.

Functional programming

LISP • APL • FP • Scheme • KRC • Hope
Miranda™ • Erlang • Curry • Gofer • Mercury
Charity • Cayenne • Mondrian • Epigram
SML • Clean • Caml • Haskell

Everything else is just *dys*functional programming!

Computation is functional

- **values** classified with respect to **types**
- **expressions**
- **functions** map values to values

Imperative vs. Functional

command	expression
executed	evaluated
has an effect	no effect
`x := 5`	`3 + 5`
(new state)	(new value)

Programming as explanation

- Problem statement
  - Invariants
  - Specifications
  - Proofs of correctness
- Analyze, decompose and fit, prove

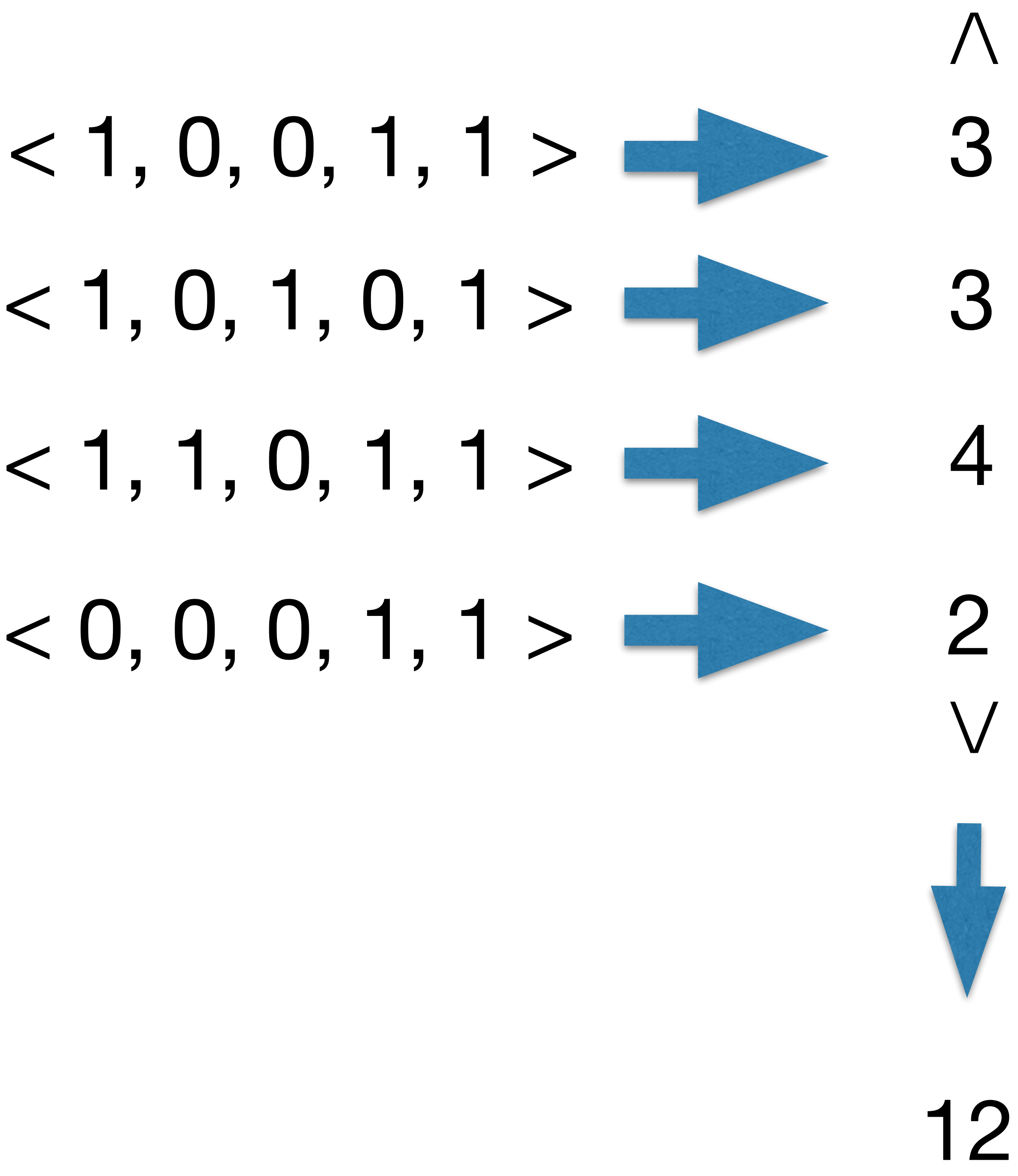
High expectation to explain precisely and concisely

Parallelism

How many people have taken 15-122?

Let’s count it using parallelism.

Parallelism



< 1, 0, 0, 1, 1 > → 3

< 1, 0, 1, 0, 1 > → 3

< 1, 1, 0, 1, 1 > → 4

< 0, 0, 0, 1, 1 > → 2

12

```
sum: int sequence → int

type row = int sequence

type room = row sequence
```

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

Analysis

- How could you improve the running time of `count`?

Divide and conquer

Parallelism

- Expression evaluation has ***no side-effects***
  - can evaluate *independent* code *in parallel*
  - evaluation order has *no effect* on *value*

- Parallel evaluation may be *faster* than sequential

Learn to *exploit* parallelism!

Cost Analysis

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

Introducing ML

- Types *t*
- Expressions *e*
- Values *v* (subset of expressions)

Examples

$$\texttt{(3 + 4) * 2}$$

$$\overset{1}{\texttt{==>}}\ \texttt{7 * 2}$$

$$\overset{1}{\texttt{==>}}\ \texttt{14}$$

$$\texttt{(3 + 4) * (2 + 1)}$$

$$\overset{3}{\texttt{==>}}\ \texttt{21}$$

How many steps would the second take if we used parallelism?

```
    "the " + "walrus"
==> "the walrus"
```

`"the walrus" + 1`   ill-typed

SML **never** evaluates an ill-typed expression!

Types, Expressions, Values

- A type is a “prediction” about the kind of value that an expression will have if it winds up having a value
- An expression is **well-typed** if it has at least one type, and **ill-typed** otherwise.
- A well-typed expression has a type, may have a value, and may have an effect (not for our effect-free fragment)

Every well-formed ML expression *e*

- has type *t*, written *as e : t*
- may have a value, written as *e* $\hookrightarrow$ *v*

  (or e ==> v)
- may have an effect (not our effect-free fragment)

Example:

(3 + 4) * 2 : int
(3 + 4) * 2 $\hookrightarrow$ *14*

Types in ML

- Basic types
  - `int, real, bool, char, string`
- Constructed types
  - Product types
  - Function types
  - User-defined types

Integers, Expressions

- Type `int`
- Values `…, ~1, 0, 1, …,`
- Expressions e_1 + e_2, e_1 - e_2, e_1 `*` e_2, e_1 `div` e_2, e_1 `mod` e_2, …
- Example `~4 * 3`

Integers, Typing

- Typing rules
  - n : `int`
  - $e_1 + e_2$: `int` if e_1 : `int` and e_2 : `int`
  - similar for other operations

`(3 + 4) * 2 : int` because

`(3 + 4): int`   `2: int`

`(3 + 4): int` because `3: int` and `4: int`

Integers, Evaluation

- $e_1 + e_2 \overset{1}{\Longrightarrow} e_1' + e_2$ if $e_1 \overset{1}{\Longrightarrow} e_1'$
- $n_1 + e_2 \overset{1}{\Longrightarrow} n_1 + e_2'$ if $e_2 \overset{1}{\Longrightarrow} e_2'$
- $n_1 + n_2 \overset{1}{\Longrightarrow} n$

  where n is the sum of n_1 and n_2

Example

Well-typed expression with no value

```
5 div 0 : int
```

Notation Recap

$e: t$	e has type t
$e \Longrightarrow e'$	e reduces to e'
$e \hookrightarrow v$	e evaluates to v

Extensional equivalence

$$\cong$$

An equivalence relation on expressions of the same type

Extensional Equivalence

- Expressions of type **int** are *extensionally equivalent* whenever one of the following is true
  - if they evaluate to the same integer
  - if they both loop forever
  - if they both raise the same exception

Equivalence is a form of *semantic equality*

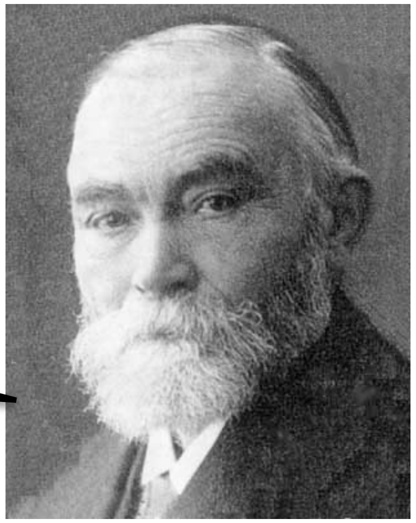
Equivalence

- Functions of type **`int -> int`** are *extensionally equivalent* if they map *extensionally equivalent arguments* to *extensionally equivalent results*

Referential transparency

for types and values

- The *type* of an expression depends only on the *types* of its sub-expressions
- The *value* of an expression depends only on the *values* of its sub-expressions



safe substitution,
compositional reasoning

Extensional Equivalence

- Expressions of type **int** are *extensionally equivalent* whenever one of the following is true
  - if they evaluate to the same integer
  - if they both loop forever
  - if they both raise the same exception

For now, we will mostly focus on the first condition by making appropriate assumptions.

Equivalence

`21 + 21 ≅ 42 ≅ 7 * 6`

`[2,4,6] ≅ [1+1, 2+2, 3+3]`

(**fn** x => x + x) ≅ (**fn** x => 2 * x)

Types in ML

- Basic types
  - `int, real, bool, char, string`
- Constructed types
  - Product types
  - Function types
  - User-defined types

Products, Expressions

- Types t_1 * t_2 for any type t_1 and t_2
- Values (v_1, v_2) for values v_1 and v_2
- Expressions (e_1, e_2), $\# e_1, \# e_2$
- Example `(~4 * 3, true)`

  `(3,5,"another example")`

usually bad style

Products, Typing

- $(e_1, e_2) : t_1 * t_2$ if $e_1 : t_1$ and $e_2 : t_2$
- Example

```
(~4 * 3, true): int * bool
(3,5,"another example"):
                     int * int * string
```

Products, Evaluation

- $(e_1\ ,\ e_2) \overset{1}{\Longrightarrow} (e_1'\ ,\ e_2)$ if $e_1 \overset{1}{\Longrightarrow} e_1'$
- $(v_1\ ,\ e_2) \overset{1}{\Longrightarrow} (v_1\ ,\ e_2')$ if $e_2 \overset{1}{\Longrightarrow} e_2'$
- $(v_1\ ,\ v_2) \overset{1}{\Longrightarrow} (v_1\ ,\ v_2)$

Evaluation:

```
       (3*4, 1.1+7.2, true)
==>    (12, 1.1+7.2, true)
==>    (12, 8.3, true)
```

We could also write:

```
(3*4, 1.1+7.2,true) ↪ (12, 8.3,true)
```

Exercises

What are the type and values of the following expressions?

	Type	**Value**
`(3*4, 1.1+7.2,true)`	`int * real * bool`	`(12,8.3,true)`
`(5 div 0, 2+1)`	`int * int`	*No value*
`(5 + "8 miles", false)`	`ill-typed`	*No value*
`(2, (true,"a"), 3.1)`	`int * (bool * string) * real`	`(2, (true,"a"), 3.1)`

Functions

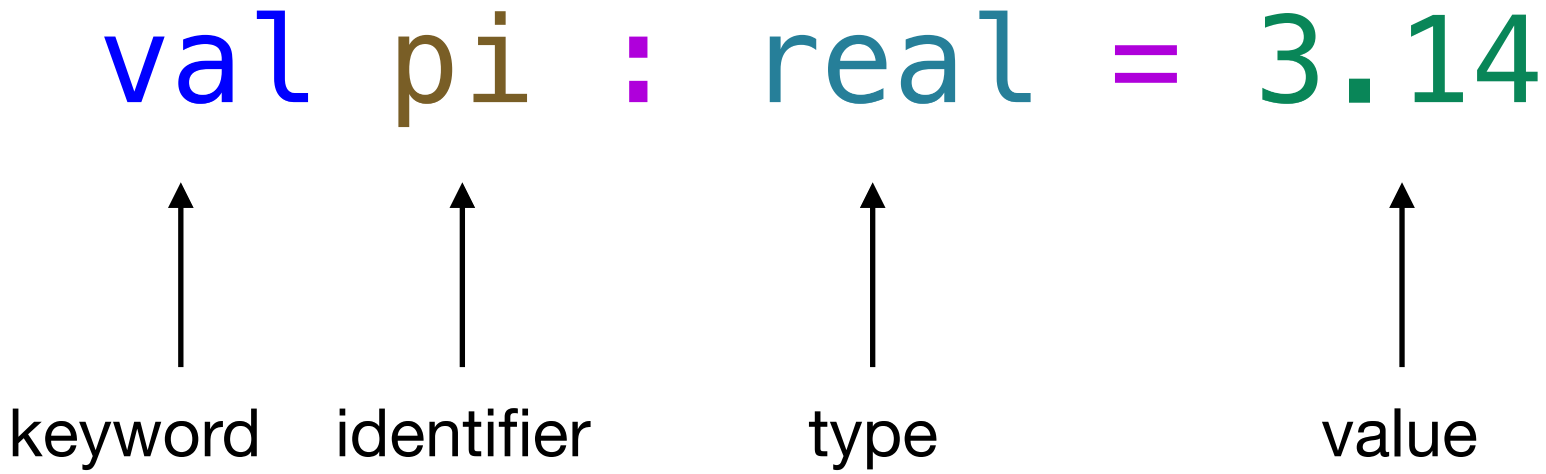
In math, one talks about a function f being a mapping between spaces X and Y.

$$f : X \to Y$$

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

Declarations, Environments, Scope

Declaration



```
val pi : real = 3.14
```

keyword identifier type value

Introduces binding of `3.14` to `pi`, sometimes written as [`3.14/x`]

Lexically statically scoped

Environment

```
val x : int = 8 - 5      [3/x]
val y : int = x + 1      [4/y]
```

Environment

```
val x : int = 8 - 5     [3/x]
val y : int = x + 1     [4/y]
val x : int = 10        [10/x]
val z : int = x + 1     [11/z]
```

Second binding of x ***shadows*** first binding. First binding has been *shadowed*.

Local declarations

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

This is an expression with type `int` and value `12`.

Local declarations

```
val k : int = 4

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

Type?
Value?

Local declarations

```
val k : int = 4

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

Type?
Value?

9.0 : real

```
k
```

Type?
Value?

4 : int

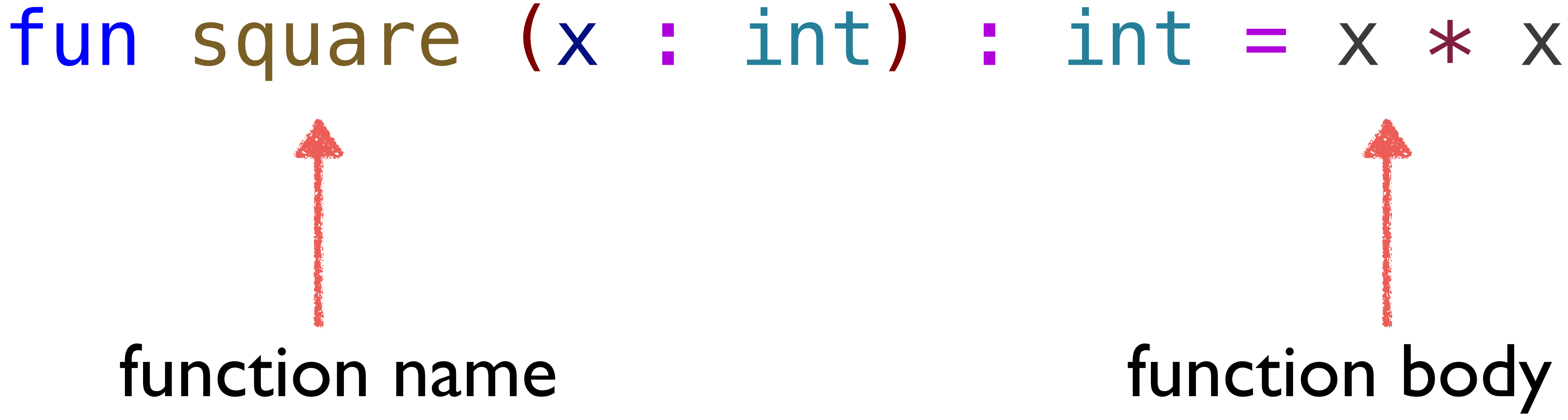
Concrete Type Definitions

```
type float = real
type point = float * float
val p : point = (1.0, 2.6)
```

Functions

Function declaration

```
(* square : int -> int
   REQUIRES: true
   ENSURES: square(x) evaluates to x * x
*)
```



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

function name

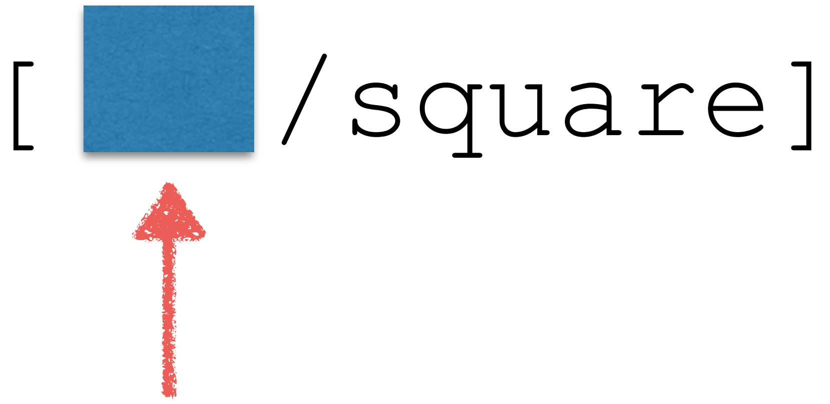
function body

Closures

Function declarations also create bindings:

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

binds the identifier square to a closure:



Lambda expression `fn x:int => x * x`
Environment (all prior bindings when square was declared)

5-step methodology

- Function name and type
- REQUIRES,
- ENSURES
- Function body
- Tests

Step 6: Proof

How does ML evaluate a function application e_2

- Evaluate e_2 to a function value f
- Reduce e_1 to a value v
- Locally extend the environment that existed at the time of the definition of f with a binding of value v to the variable x
- Evaluate the body in the resulting environment

To Do Tonight

- Canvas
  - Assignments
    - Set up lab