Principles of Programming Languages

Jiwon Seo

2022 Spring

Instructor: Jiwon Seo

- Faculty at Hanyang University (2017.9 ~)
- Faculty at UNIST (2016~2017)
- Pinterest, LinkedIn (2015, 2014)
- MS, PhD in Stanford
- **Research interest**
  - **Deep Learning Systems**
  - **Neural Network Security**

Lab Homepage: http://bigdata.hanyang.ac.kr

Communication w/ Instructors

- Announcements on course homepage
  - Your responsibility to check frequently
- Course homepage Q/A board:
  - Programming questions
- Administrative questions: pl.hanyang@gmail.com
  - TA or instructors will answer
  - seojiwon@hanyang only if you really need to communicate directly.

How To Write Emails

- Subject: [PL/Mon-Tue] summary of your request
- Body: include your name, student ID.

  be concise and to the point!

<u>I will only reply the emails following the above rule!</u>

- If you need to meet with me, please use my office hour.

Welcome!

We have 15 weeks to learn *the fundamental principles* of programming languages

With hard work, patience, and an open mind, this course makes you a much better programmer

- Even in languages we won't use
- Learn the core ideas around which *every* language is built, despite their surface-level differences and variations
- *Poor* course summary: "Learn ML, Scheme (Racket), Cuda"

Concise to-do list

In the next 24-48 hours:

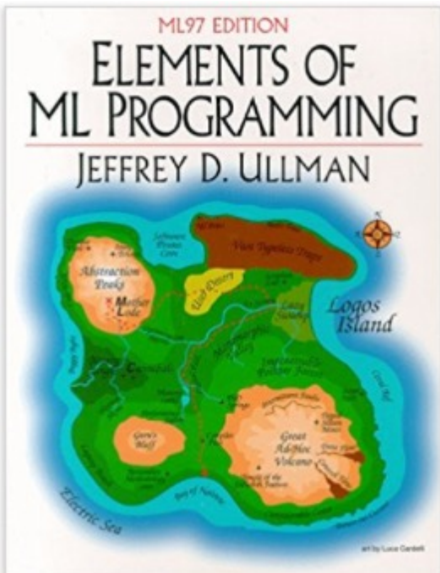
1. Read course web page
2. Read the course policy on the web

3. **Install and setup SML** (Standard ML)
   – Installation instructions on web page
     (under Resources 강의자료실)

4. **Read homework assignment 1 (due in 2 weeks)**

Textbooks – Optional

- Elements of ML Programming



- Textbook is optional
  - A few copies in Library
  - Look up details you want/need to know

Textbooks – Optional (Easier One)

- Programming in Standard ML '97: A Tutorial Introduction
- Available Online: http://homepages.inf.ed.ac.uk/stg/NOTES/notes.pdf

Office hours

- Office: IT/BT 405-1
- Office Hour: TBD
- TA Office hour: appointment only
- Try to take advantage of the lecture and ask questions during the lecture

Homework

- 5~6 in total
- To be done individually
- Doing the homework involves:
  1. Understanding the concepts being addressed
  2. Writing code demonstrating understanding of the concepts
  3. Testing your code to ensure you understand and have correct programs
  4. “Playing around” with variations, incorrect answers, etc.

  Only (2) is graded, but focusing on (2) makes homework harder

Academic Honesty

- Read the course policy carefully
  - Clearly explains how you can and cannot get/provide help on homework and projects

- Always explain any unconventional action

- We have scripts that automatically identify code copies
  - Among the students
  - Internet resources

Exams

- Midterm: To be decided (probably in class)

- Final: To be decided

- Same concepts, but different format from homework
  - More conceptual (but write code too)
  - Closed book/notes, but you bring one paper with whatever you want on it

Attendance

- If you are absent more than 9 times, you will get F grade (as per university academic rule)

- Otherwise your attendance will affect little on your grade

Questions?

Anything I forgot about course mechanics ?

What this course is about

- Many essential concepts relevant in any programming language
  - And how these pieces fit together
- Use ML, Racket, Cuda (and occasionally other languages):
  - They represent different language families
  - They let many of the concepts “shine”
  - Using multiple languages shows how the same concept can “look different” or actually be very similar
- Big focus on *functional programming*
  - Not using *mutation* (assignment statements) (!)
  - Using *first-class functions* (can’t explain that yet)
  - But many other topics too

Why learn this?

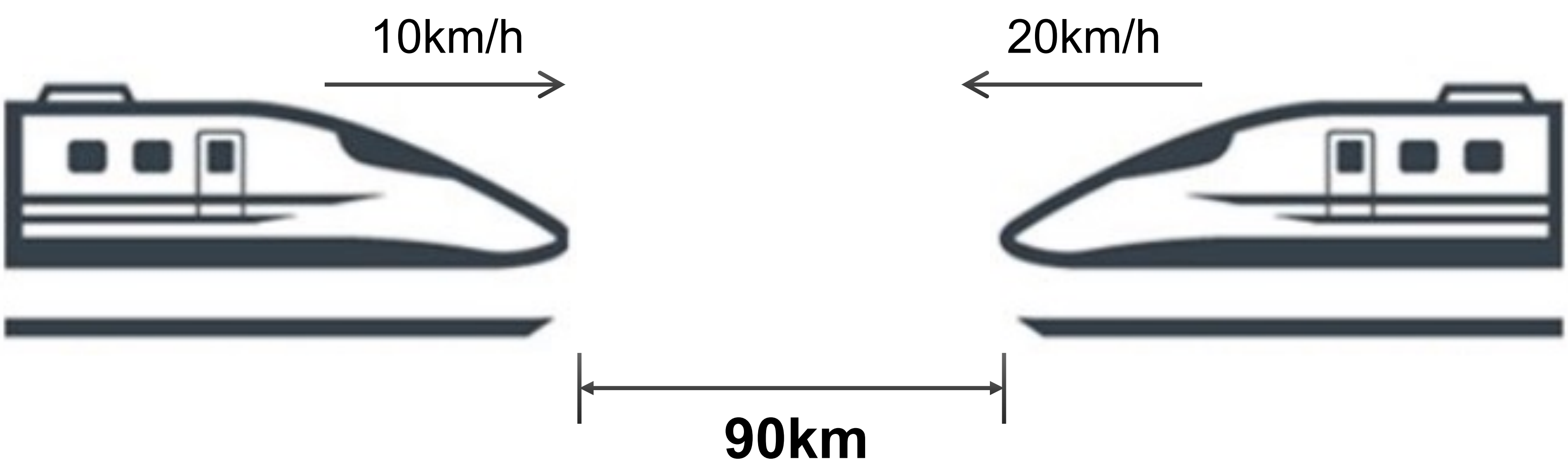
Learning to think about software in this "PL" way will make you a better programmer even if/when you go back to old ways

It will also give you the mental tools and experience you need for a lifetime of confidently picking up new languages and ideas

You will learn to think in different ways

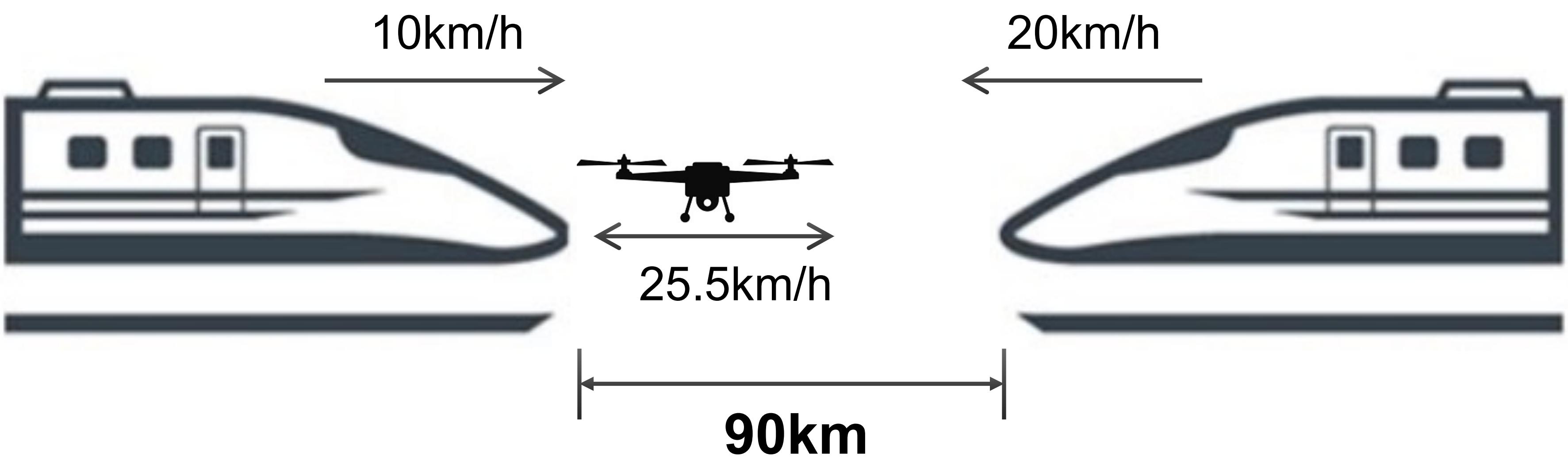
Why learn this? (more)

Quiz!



Why learn this? (more)

Quiz!



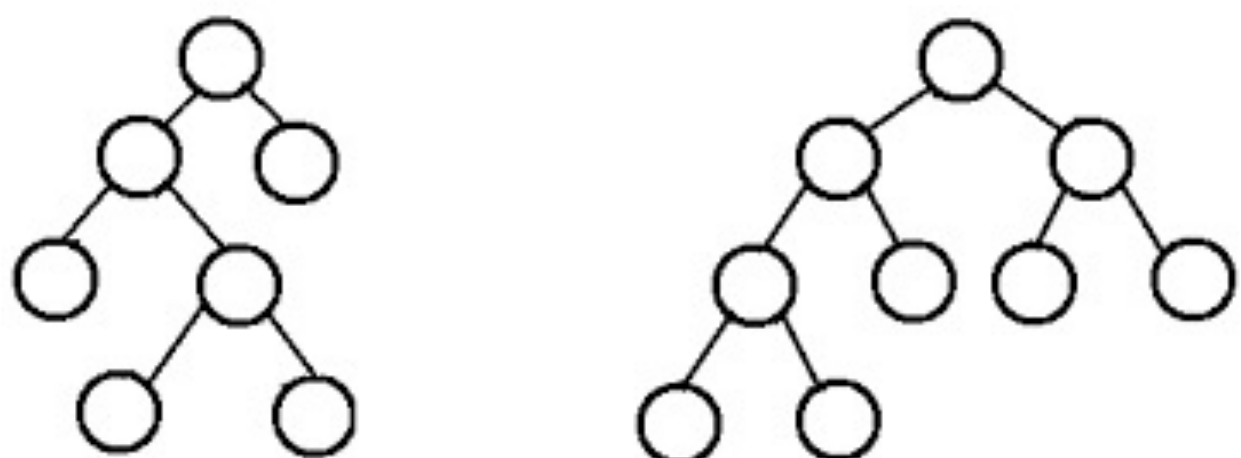
Calculate the total distance the drone flew.

Why learn this? (longer version)

Another quiz!

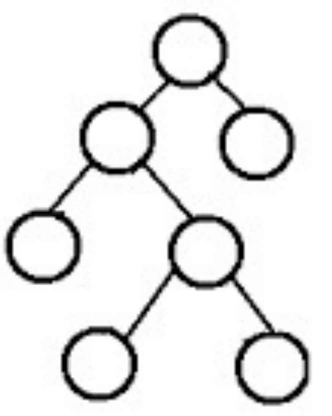
You have a tree data structure like following:

```
struct tree_node {
    int val;
    struct tree_node* left;
    struct tree_node* right;
}
```



Why learn this? (longer version)

```
struct tree_node {
    int val;
    struct tree_node* left;
    struct tree_node* right;
}
```

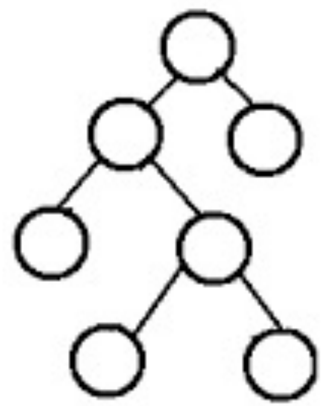


You need to implement the code that

1) adds up all the values in a tree
2) tests if a number is in a tree
3) tests if all the numbers in a tree are even numbers

Why learn this? (longer version)

```
struct tree_node {
    int val;
    struct tree_node* left;
    struct tree_node* right;
}
```



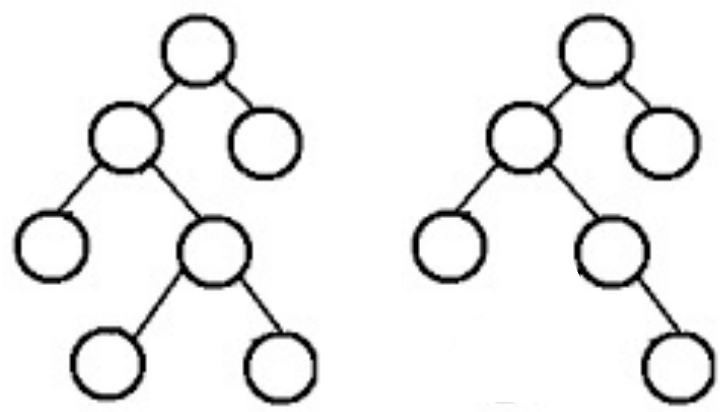
Typically you implement an iterator to do these.

```
int sum = 0;
Iterator i = Tree.iterator();
while (i.hasNext()) {
   sum += i.next();
}
```

However, what if you need to implement …

Why learn this? (longer version)

```
struct tree_node {
    int val;
    struct tree_node* left;
    struct tree_node* right;
}
```



For two trees, you need to implement the code that

1) tests if the two trees are exactly in same structure
2) tests if each sub-tree of a tree is taller than that of another

…

How would you implement this?

Why learn this? (longer version)

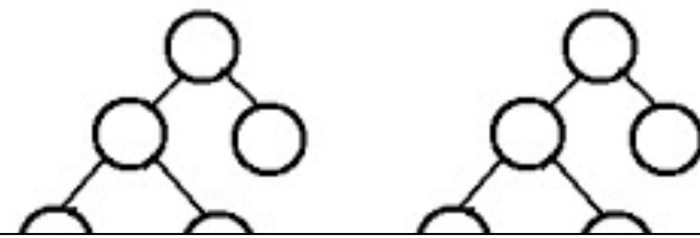
```
struct tree_node {
   int val;
```

My take on this:

```
int dual_tree_visit(tree_node* t1, tree_node* t2, function f) {
   int res = f(t1, t2);
   if (res == ITER_CONTINUE) {
      res = dual_tree_visit(t1->left, t2->left, f);
      if (res == ITER_CONTINUE) {
         res = dual_tree_visit(t1->right, t2->right, f);
      }
   }
   return res;
}
```

Why learn this? (longer version)

```
struct tree_node {
   int val;
   struct tree_node* left;
```



My take on this:

```
int tree_struct_equals(tree_node* t1, tree_node* t2) {
  if (t1->val == t2->val) { // null check omitted
     // (t1-> left==null) ==  (t2->left==null)
     // (t1-> right==null) ==  (t2->right==null)
     return ITER_CONTINUE;
  } else {
     return false;
  }
}
```

A strange environment

- Next 4-5 weeks will use
  - ML language
  - Read-eval-print-loop (REPL) for evaluating programs
- *You* need to get things installed and configured
  - We’ve written instructions (questions welcome)
- Only then can you focus on the content of Homework 1
- Working in strange environments is a CSE life skill

Mindset

- "Let go" of all programming languages you already know

- For now, treat ML as a "totally new thing"
  - Time later to compare/contrast to what you know
  - For now, "oh that seems kind of like this thing in [Java]" will confuse you, slow you down, and you will learn less

- Start from a blank file…

A very simple ML program

A very simple ML program

[The same program we just wrote in Vim; here for convenience if reviewing the slides]

```
(* This is a comment. My first ML program *)

val x = 34;
(* static environ: x:int *)
(* dynamic environ: x->34 *)
val y = 17;

val z = (x + y) + (y + 2);

val q = z + 1;

val abs_of_z = if z < 0 then 0 - z else z;

val abs_of_z_simpler = abs z
```

A variable binding

```
val z = (x + y) + (y + 2); (* comment *)
```

More generally:

```
val x = e;
```

- *Syntax*:
  - *Keyword* **`val`** and *punctuation* **=** and **`;`**
  - *Variable* ***`x`***
  - *Expression* ***`e`***
    - Many forms of these, most containing *subexpressions*

The semantics

- Syntax is just how you write something

- Semantics is what that something means
  - Type-checking (before program runs)
  - Evaluation (as program runs)

- For variable bindings:
  - Type-check expression and extend static environment
  - Evaluate expression and extend dynamic environment

So what is the precise syntax, type-checking rules, and evaluation rules for various expressions?

Expressions

- We have seen many kinds of expressions:

  `34`   **`true`**   **`false`**   **`x`**   ***`e1+e2`***   ***`e1<e2`***

  `if` ***`e1`*** **`then`** ***`e2`*** **`else`** ***`e3`***

- Can get arbitrarily large since any subexpression can contain subsubexpressions, etc.

- Every kind of expression has
  1. Syntax
  2. Type-checking rules
     - Produces a type or fails (with a bad error message ☹)
     - Types so far: **`int`** **`bool`** **`unit`**
  3. Evaluation rules (used only on things that type-check)
     - Produces a value (or exception or infinite-loop)

Variables

- Syntax:

- Type-checking:

- Evaluation:

Variables

- Syntax:
  sequence of letters, digits, _, not starting with digit
- Type-checking:
  Look up type in current static environment
  - If not there fail
- Evaluation:
  Look up value in current dynamic environment

Addition

- Syntax:

  `e1 + e2` where ***`e1`*** and ***`e2`*** are expressions

- Type-checking:

  If ***`e1`*** and ***`e2`*** have type **`int`**,

  then ***`e1 + e2`*** has type **`int`**

- Evaluation:

  If ***`e1`*** evaluates to **`v1`** and ***`e2`*** evaluates to **`v2`**,

  then ***`e1 + e2`*** evaluates to sum of **`v1`** and **`v2`**

Values

- All values are expressions
- Not all expressions are values
- A value "evaluates to itself" in "zero steps"
- Examples:
  - **`34`**, **`17`**, **`42`** have type **`int`**
  - **`true`**, **`false`** have type **`bool`**
  - **`()`** has type **`unit`**

Slightly tougher ones

What are the syntax, typing rules, and evaluation rules for conditional expressions?

What are the syntax, typing rules, and evaluation rules for less-than expressions?

Conditional Expression

Let’s go over it ourselves

Conditional Expression

- Syntax: **`if`** ***`e1`*** **`then`** ***`e2`*** **else** ***`e3`***

  (where **`if, then, else`** `are keywords`

  `e1, e2,` `and` ***`e3`*** are subexpressions)
- Type-checking:

  `e1` must have type bool

  `e2` and ***`e3`*** can have any type (let's call it t),

  but they must have the same type t

  the type of the entire expression is also t
- Evaluation:

  first evaluate ***`e1`*** to a value (v1).

  if it's true, evaluate ***`e2`*** → result of the whole expression

  else evalute ***`e3`*** → result of the whole expression

Less-than Expression

- Syntax: ***`e1 < e2`*** *try it yourself!*

- Type-checking:

- Evaluation:

Function definitions

Functions: the most important building block in the whole course

- Like Java methods, have arguments and result
- But no classes, **`this`**, **`return`**, etc.

Example *function binding*:

```
(* Note: correct only if y>=0 *)

fun pow (x : int, y : int) =
  if y=0
  then 1
  else x * pow(x,y-1)
```

Note: The *body* includes a (recursive) *function call*: **`pow(x,y-1)`**

Example, extended

```
fun pow (x : int, y : int) =
  if y=0
  then 1
  else x * pow(x,y-1)

fun cube (x : int) =
  pow (x,3)

val sixtyfour = cube 4

val fortytwo = pow(2,2+2) + pow(4,2) + cube(2) + 2
```

Some gotchas

Three common “gotchas”

- Bad error messages if you mess up function-argument syntax
- The use of `*` in type syntax is not multiplication
  - Example: `int * int -> int`
  - In expressions, `*` is multiplication: `x * pow(x,y-1)`
- Cannot refer to later function bindings
  - That’s simply ML’s rule
  - Helper functions must come before their uses
  - Need special construct for *mutual recursion* (later)

Recursion

- If you're not yet comfortable with recursion, you will be soon ☺
  - Will use for most functions taking or returning lists

- "Makes sense" because calls to same function solve "simpler" problems

- Recursion more powerful than loops
  - We won't use a single loop in ML
  - Loops often (not always) obscure simple, elegant solutions

Function bindings: 3 questions

- Syntax: `fun x0 (x1 : t1, … , xn : tn) = e`
  - (Will generalize in later lecture)

- Evaluation:

- Type-checking:

Function bindings: 3 questions

- Syntax: `fun x0 (x1 : t1, … , xn : tn) = e`
  - (Will generalize in later lecture)

- Evaluation: ***A function is a value!*** (No evaluation yet)
  - Adds **`x0`** to environment so *later* expressions can *call* it
  - (Function-call semantics will also allow recursion)

- Type-checking:
  - Adds binding **`x0 : (t1 * … * tn) -> t`** if:
  - Can type-check body **`e`** to have type **`t`** in the static environment containing:
    - "Enclosing" static environment (earlier bindings)
    - **`x1 : t1, …, xn : tn`** (arguments with their types)
    - **`x0 : (t1 * … * tn) -> t`** (for recursion)

More on type-checking

```
fun x0 (x1 : t1, … , xn : tn) = e
```

- New kind of type: `(t1 * … * tn) -> t`
  - Result type on right
  - The overall type-checking result is to give `x0` this type in rest of program (unlike Java, not for earlier bindings)
  - Arguments can be used only in `e` (unsurprising)

- Because evaluation of a call to `x0` will return result of evaluating `e`, the return type of `x0` is the type of `e`

- The type-checker "magically" figures out `t` if such a `t` exists
  - Later lecture: Requires some cleverness due to recursion
  - More magic after hw1: Later can omit argument types too

Function Calls

A new kind of expression: 3 questions

Syntax: ***`e0 (e1,…,en)`***

- (Will generalize later)
- Parentheses optional if there is exactly one argument

Type-checking:

If:

- **`e0`** has some type **`(t1 * … * tn) -> t`**
- **`e1`** has type **`t1`**, …, **`en`** has type **`tn`**

Then:

- **`e0(e1,…,en)`** has type **`t`**

Example: **`pow(x,y-1)`** in previous example has type **`int`**

Function-calls continued

`e0(e1,…,en)`

Evaluation:

1. (Under current dynamic environment,) evaluate **`e0`** to a function **`fun x0 (x1 : t1, … , xn : tn) = e`**
   – Since call type-checked, result *will be* a function
2. (Under current dynamic environment,) evaluate arguments to values **`v1, …, vn`**
3. Result is evaluation of **`e`** in an environment extended to map **`x1`** to **`v1`**, …, **`xn`** to **`vn`**
   – ("An environment" is actually the environment where the function was defined, and includes **`x0`** for recursion)

Comparisons

For comparing `int` values:

`=  <>  >  <  >=  <=`

You might see weird error messages because comparators can be used with some other types too:

- `> < >= <=` can be used with `real`, but not 1 `int` and 1 `real`
- `=  <>` can be used with any “equality type” but not with `real`
  - Let’s not discuss equality types yet

Debugging Errors

Your mistake could be:

- Syntax: What you wrote means nothing or not the construct you intended

- Type-checking: What you wrote does not type-check

- Evaluation: It runs but produces wrong answer, or an exception, or an infinite loop

Keep these straight when debugging even if sometimes one kind of mistake appears to be another

Related Sections in Elements of ML Programming

Section 2.1 (Expressions), 3.1 (Functions)