Functional Programming
History: Procedural Programming

- Do you think of programming as describing step-by-step instructions to the computer?
- A procedure is a subprogram that takes input but does not return any value
  - Instead, it has side effects...
  - ...by manipulating program data/memory directly
- Can we know what such a program does simply by reading its source code?
  - May be difficult
Procedural Programming: The Transform Pattern

```java
list = {1, 2, 3}
for(int i = 0; i < list.length(); i++)
    list[i] = 2 * list[i]
```
Procedural Programming: The Fold Pattern

```python
sum = 0
for (int i = 0; i < 5; i++)
    sum += i

list = {}
for (int i = 0; i < 5; i++)
    list.append(i)
```
Function!

- Is a function simply a procedure that returns a value?
  - No. A function is a (mathematically speaking) a relation between two sets
  - A function does not specify how to compute this relation
  - A function returns the same value for same inputs
  - A function does not have side effects

- Due to confusion, we’ll call the concept described above pure function
Types: Defining a Function

- A function is specified mathematically as
  \[ f : \mathbb{R} \rightarrow \mathbb{C} \]
  \[ x \mapsto \sqrt{x} \]
- \( f \) is a function that takes a real number and returns a complex number
- \( x \) has the type: \( \mathbb{R} \)
- \( \sqrt{x} \) has the type: \( \mathbb{C} \)
- \( f \) has the type: \( \mathbb{R} \rightarrow \mathbb{C} \)
Tuple Type

- A function that takes a tuple
  \[ g : (\mathbb{R}, \mathbb{R}) \rightarrow \mathbb{R} \]
  \[ (x, y) \mapsto x \times y \]
Function Type: As Parameter

- A function that takes another function

\[ h : (\mathbb{R} \rightarrow \mathbb{C}) \rightarrow \mathbb{C} \]

\[ f \quad \mapsto \quad f(3) \]
Function Type: As Return Value

- A function that returns a function

\[ h : \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \]
\[ x \mapsto \]
\[ f : \mathbb{R} \rightarrow \mathbb{R} \]
\[ y \mapsto x \times y \]

- The returned function \( f \) is a closure\(^1\), since it closes over (captured) \( x \)
- \( h \times y \) multiplies \( x \) by \( y \)
- \( h \) has type \( \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \)
- \( h\ 3 \) is a function that multiplies its input by \( 3 \)
  (this is known as **currying**)
  \[ (h\ 3)\ 4 = 12 \]

\(^1\)An object is state plus function. A closure, is function plus state.
Tuples are not curryable

- A function that takes a tuple
  \[ g : ( \mathbb{R}, \mathbb{R} ) \rightarrow \mathbb{R} \]
  \[ (x, y) \mapsto x \times y \]
- There is no such thing as \( g \ x \ y \) or \( g \ x \)
- There is \( g \ (x, y) \)
List Type

- A function that takes a list of real numbers and add them

\[
\begin{align*}
  & f : \mathbb{R}^2 \rightarrow \mathbb{R} \\
  & X \mapsto \sum_{i \in X} i
\end{align*}
\]

\[
\mathbb{R}^2 = \mathbb{R}^* = \bigcup_{n=0}^{\infty} \mathbb{R}^n.
\]
Another Example

- A function that takes two lists of real numbers and adds them

\[
f : ([\mathbb{R}], [\mathbb{R}]) \rightarrow [\mathbb{R}]
\]

\[
(X, Y) \mapsto (X_i + Y_i)_{i \in \text{supp}(X)}
\]
Composition

- Function composition\(^3\)
  \[ c : (\gamma \to \beta, \alpha \to \gamma) \to \alpha \to \beta \]
  \[ (f, g) \mapsto k : x \mapsto f \circ g \circ x \]

- \(k\) is a closure

- Exercise: make the example from the previous slide curryable

\(^3\) (greek letters are type variables, they match any type)
Conditionals

- A function that is different depending on the value

\[
\text{abs} : \mathbb{R} \rightarrow \mathbb{R}
\]

\[
x \mapsto \begin{cases} 
  x, & \text{if } x \geq 0 \\
  -x, & \text{otherwise}
\end{cases}
\]
Union Types

- Let type $\mathbb{B} = \{\text{true}, \text{false}\}$

- A function that returns either a Real or a Boolean:

  $c : \mathbb{Z} \rightarrow \mathbb{R} \cup \mathbb{B}$

  $x \mapsto \begin{cases} \sqrt{x}, & \text{if } x \geq 0 \\ \text{false}, & \text{otherwise} \end{cases}$

- To indicate undefined cases use instead the special value $\text{bottom} : \bot$
  - $\bot$ belongs to every type
  - (somehow like $\text{NULL}$)
Loops

- Loops do not exist. Use recursively-defined functions instead.
- Very loosely speaking: partial recursive functions are Turing complete\(^4\).

Example:

\[
\text{factorial} : \mathbb{Z} \rightarrow \mathbb{Z}
\]

\[
\begin{align*}
x & \mapsto \\
x \times \text{factorial}(x - 1), & \quad \text{if } x \geq 1 \\
1, & \quad \text{if } x = 0 \\
\bot, & \quad \text{otherwise}
\end{align*}
\]

\(^4\)Look up the Church Turing Thesis.
Functional Programming: The Transform Pattern

```java
list = {1, 2, 3}
for (int i = 0; i < list.length(); i++)
    list[i] = 2 * list[i]
```

- `map : (α → β, [α]) → [β]`
  
  \[(f, X) \mapsto (f(X_i))_{i \in \text{supp}(X)}\]

- `mult2 : [\mathbb{R}] \rightarrow [\mathbb{R}]`
  
  \[X \mapsto \text{map } ((\times) \ 2) \ X\]

- \(x \times y\) can also be written as: \((\times) \ x \ y\)

- Therefore, can be curried as: \((\times) \ x\)

- `mult2` can be succinctly written\(^5\) as: `map (\times) 2`
  
  - No need to specify the parameter name

\(^5\)If `map` was defined in a currying way.
Procedural Programming: The Fold Pattern

```java
sum = 0
for(int i = 0; i < 5; i++)
    sum += i
```

- `foldr a [x_1, x_2, \ldots, x_n]`
  - \( = f \, x_1 \, (f \, x_2 \, (\cdots \, f \, x_n \, a \cdots)) \)
  - `append(x_1,append(x_2,\cdots \, append(x_n,[])\cdots))`

- \( \text{foldr} : \quad (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \)

- \((f, a, X)^6 \mapsto \begin{cases} f \cdot x \cdot a, & \text{if } X = [x] \\ f \cdot x \cdot (f \cdot a \cdot Y), & \text{if } X = [x, Y] \end{cases}\)

- `sum = foldr (+) 0`
- `product = foldr (+) 1`
- `and = foldr (\wedge) \text{true}`

\(^{6}\)Should have been defined as three nested functions, but we wrote it this way instead for clarity.
Exercises

- Define a function `filter` which takes a predicate and a list and produces a list of the items which satisfy the predicate
- Define APL's \( \Lambda \)
Thank you!