Semester Review

- Imperative Programming
  - Inspired by the explicit state manipulation of Von-Neuman hardware architecture
  - CPU↔Memory
Type Systems

- “A type is a set of values”
- Help identify programming errors
  - A type mismatch usually indicates a programming error
  - Type propagation
- Dynamic/static type systems
- Subtypes/Supertypes
  - Type hierarchies
  - Automatic type coercion (conversion, promotion)
  - Widening/narrowing conversions
Semester Review

- Pattern matching
  - Simple patterns are expressions that consist purely of constructors and variables
  - Canonical representations!
  - Destructuring
    - let (x,y) = (1,2)
  - Powerful declarative way of accessing substructures of objects
Semester Review

- OOP
  - “classic” vs “modern” OOP
  - Modern OOP
    - No classes, instead structures with behavior
    - No (class) inheritance – traits/interfaces instead or object composition
    - Limited if any member protection – facilitates pattern matching on objects.
  - Subtype polymorphism with dynamic dispatch for statically typed languages
  - *Duck typing* for dynamically typed languages
Semester Review

- Functional Programming
  - Based on the lambda calculus
  - “Everything is a value”
  - No explicit state
  - First-class functions
  - Declarative:
    - “The What rather than the How”

```
function len
  with [] do
    0
  with [_[remaining_list]] do
    1 + len remaining_list
end
```

Function application:

\[(\lambda x. x + 1) 1 \Rightarrow x + 1[x \leftarrow 1] \Rightarrow 1 + 1 \Rightarrow 2\]
Semester Review

- First-Class Patterns
  - Patterns themselves are considered values
    - Store in variables
    - Pass to/from functions
  - Promoting features to *first-class status* increases expressiveness of programming languages
    - Shorter programs that make intentions of programmer clearer.

```plaintext
let pos_int = pattern (x:%integer) if x>0.

function fact
    with 0 do
        1
    with n:*pos_int do
        n*fact(n-1)
end

assert (fact 3 == 6).
```
Semester Review

- Putting it all together

```bash
// imperative version of the quicksort
function qsort with a do
    if len(a) <= 1 do
        return a
    else do
        let pivot = a@0.
        let rest = a@range(1,len(a)).
        let less = [].
        let more = [].
        for e in rest do
            if e <= pivot do
                less @append(e).
            else
                more @append(e).
        end
        return qsort(less) + [pivot] + qsort(more).
    end
end

// multi-paradigm version of the quicksort
function qsort
    with [] do
        []
    with [a] do
        [a]
    with [pivot|rest] do
        let less = [].
        let more = [].
        for e in rest do
            if e <= pivot do
                less @append e.
            else do
                more @append e.
            end
        end
        qsort less + [pivot] + qsort more.
end
```
# Semester Review

- Putting it all together – multi-paradigm

```python
# imperative version of quicksort
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)
```

```python
# declarative version of quicksort
def quicksort(arr):
    match arr:
        case []:
            return []
        case [a]:
            return [a]
        case (pivot,*rest):
            less = [x for x in rest if x <= pivot]
            greater = [x for x in rest if x > pivot]
            return quicksort(less) + [pivot] + quicksort(greater)
```
Putting it all together – higher-order

```python
-- higher-order programming version of the
function qsort
with ([], %function) do
  []
with ([a], %function) do
  [a]
with ([pivot|rest], %function) do
  let less = [].
  let more = [].
  for e in rest do
    if order(e, pivot) do
      less @append e.
    else do
      more @append e.
  end
end
qsort(less, order) + [pivot] + qsort(more, order).

# higher-order version of quicksort
def quicksort(arr, order):
  match arr:
    case []:
      return []
    case [a]:
      return [a]
    case (pivot,*rest):
      less = [x for x in rest if order(x, pivot)]
      greater = [x for x in rest if not order(x, pivot)]
      return quicksort(less, order) + [pivot] + quicksort(greater, order)

unsorted_arr = [5, 3, 8, 4, 2, 7, 1, 10]
sorted_arr = [1, 2, 3, 4, 5, 7, 8, 10]
assert(quicksort(unsorted_arr, lambda a,b: a <= b) == sorted_arr)

assert (qsort ([2,5,1,3,4], lambda with (a,b) do a<=b) == [1,2,3,4,5]).
```
Semester Review

The End