Putting it All Together

- Multi-paradigm programming means picking and choosing from our various paradigms,
  - Imperative
  - Declarative with pattern matching
  - Functional
  - OOP
  - First-class patterns
- To create the most readable and maintainable programs.

Case Study: QuickSort

- We start with the imperative and the functional versions of the quicksort
  - Examining both the strengths and weaknesses of each approach
- We then pick and choose from each of these implementations and create a multi-paradigm version of the quicksort.
- Finally, we'll create some extensions such as a flexible sorting predicate based on higher-order programming.

Imperative Programming

```
-- 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
        end
        return qsort(less) + [pivot] + qsort(more).
    end
end

assert(qsort([3,7,1,6,9,5,2,10,8,4]) == [1,2,3,4,5,6,7,8,9,10]).
```

ln017/qimp.ast

Functional Programming

```
-- functional version of the quicksort
function qsort
   with [] do
      []
   with [a] do
      [a]
   with [pivot|rest] do
      function filter
         with ([],_) do
            ([],[])
         with ([e|rest],pivot) do
            let (a,b) = filter (rest,pivot).
            return ([e]+a,b) if e <= pivot else (a,[e]+b).
      end
      let (less,more) = filter (rest,pivot).
      qsort less + [pivot] + qsort more.
end

assert (qsort [3,7,1,6,9,5,2,10,8,4] == [1,2,3,4,5,6,7,8,9,10]).
```

ln017/qfun.ast

Multi-Paradigm Programming

```
-- 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

assert (qsort [3,7,1,6,9,5,2,10,8,4] == [1,2,3,4,5,6,7,8,9,10]).
```

ln017/qmulti.ast

Multi-Paradigm Programming - Python

```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)

unsorted_arr = [5, 3, 8, 4, 2, 7, 1, 10]
sorted_arr = [1, 2, 3, 4, 5, 7, 8, 10]
assert(quicksort(unsorted_arr) == sorted_ar
```

ln017/qimp.py

```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)

unsorted_arr = [5, 3, 8, 4, 2, 7, 1, 10]
sorted_arr = [1, 2, 3, 4, 5, 7, 8, 10]
assert(quicksort(unsorted_arr) == sorted_arr)
```

ln017/qmulti.py

Constraint Patterns

```
-- constraint patterns to define the qsort domain
load system type.
let f = lambda with (acc,x) do acc and type @isscalar x.
let Scalar_List = pattern %[(a:%list) if a @reduce (f,true)]%.

function qsort
    with []:*Scalar_List do
        []
    with [a]:*Scalar_List do
        [a]
    with [pivot|rest]:*Scalar_List 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

assert (qsort [3,7,1,6,9,5,2,10,8,4] == [1,2,3,4,5,6,7,8,9,10]).
```

ln017/qconstr.ast

Higher-Order Programming

ln017/qhigh.ast

```
-- higher-order programming version of the quicksort
function qsort
   with ([],%function) do
      []
   with ([a],%function) do
      [a]
   with ([pivot|rest],order:%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).
end

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

Higher-Order Programming - Python

ln017/qhigh.py

```python
# 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)
```

Higher-Order Programming

- The version quicksort that uses a passed in order predicate is interesting because it is now generic over the objects it can sort…

```
load system type.

structure Person with
    data name.
    data age.
    function __str__ with () do this@name+"("+this@age+")" end
end

let people = [
    Person("Liz",32),
    Person("Joe",20),
    Person("Jessica",22),
    Person("Peter",18)
].

function order_age with (a:%Person,b:%Person) do
    a@age <= b@age.
end

function qsort
    with ([],%function) do ⋯
    with ([a],%function) do ⋯
    with ([pivot|rest],order:%function) do ⋯
end

-- sort people by their age
let sorted = qsort (people,order_age).
assert (type @tostring sorted == "[Peter(18),Joe(20),Jessica(22),Liz(32)]")
```

Higher-Order Programming - Python

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__ (self):
        return self.name+"("+str(self.age)+")"

people = [
    Person("Liz",32),
    Person("Joe",20),
    Person("Jessica",22),
    Person("Peter",18)
]

def order_age (a,b):
    return a.age <= b.age

def quicksort(arr, order):...

# sort people by their age
sorted = quicksort(people, order_age)
for p in sorted:
    print(p)
```

Case Study: SpaceObjects

- This program is inspired by the programs from the Wikipedia page: https://en.wikipedia.org/wiki/Multiple_dispatch
- The idea is that we are given pairs of space objects and we have to write a function that determines what kind of collision we are looking at and print out messages accordingly.
- We'll start with an imperative solution to this

Imperative Solution

```
load system io.
load system type.

structure Asteroid with data size end
structure Spaceship with data size end

function collide with (a,b) do
    let typea = type @gettype a.
    let typeb = type @gettype b.
    if (typea in ["Asteroid","Spaceship"]) and
        (typeb in ["Asteroid","Spaceship"]) and
        (a@size > 100) and
        (b@size > 100) do
        return "Big boom! collision"
    elif typea == "Asteroid" and typeb == "Asteroid" do
        return "asteroid <-> asteroid collision ".
    elif typea == "Spaceship" and typeb == "Spaceship" do
        return "spaceship <-> spaceship collision".
    elif (typea in ["Asteroid","Spaceship"]) and
        (typeb in ["Asteroid","Spaceship"]) do
        return "spaceship <-> asteroid collision".
    else do
        throw Error("unkown collision")
    end
end

io @println (collide(Asteroid(101), Spaceship(300))).
io @println (collide(Asteroid(10), Spaceship(10))).
io @println (collide(Spaceship(101), Spaceship(10))).
```

- Everything is accomplished computationally.
- Developer's intentions are not immediately visible.

```
lutz$ asteroid spaceimp.ast
Big boom! collision
spaceship <-> asteroid collision
spaceship <-> spaceship collision
```

ln017/spaceimp.ast

Multi-Paradigm Solution

```
load system io.
load system type.

structure Asteroid with data size end
structure Spaceship with data size end

let SpaceObject = pattern %[x if (x is %Asteroid) or (x is %Spaceship)]%.
let BigObject = pattern %[(x:*SpaceObject) if x@size > 100]%.

function collide
   with (a:*BigObject, b:*BigObject) do
      return "Big boom! collision"
   with (a:%Asteroid, b:%Asteroid) do
      return "asteroid <-> asteroid collision".
   with (a:%Spaceship, b:%Spaceship) do
      return "spaceship <-> spaceship collision".
   with (*SpaceObject, *SpaceObject) do
      return "spaceship <-> asteroid collision".
end

io @println (collide(Asteroid(101), Spaceship(300))).
io @println (collide(Asteroid(10), Spaceship(10))).
io @println (collide(Spaceship(101), Spaceship(10))).
```

Employs:
- Multi-dispatch
- Pattern matching
- First-Class Patterns

A more declarative approach due to pattern matching ☞ makes developer intentions much more visible!

ln017/spacemulti.ast