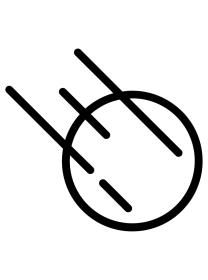
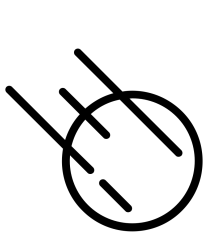


# Flow of Control

- Control structure implementation in Asteroid is along the lines of any of the modern programming languages such as Python, Swift, or Rust. For example,
  - The for loop allows you to iterate over lists without having to explicitly define a loop index counter.
  - The if-elif-else statement expresses familiar condition handling



# Flow of Control

ln005/loop1.ast

```
load system io.

for i in 0 to 10 step 2 do
    io @println i.
end
```

ln005/if1.ast

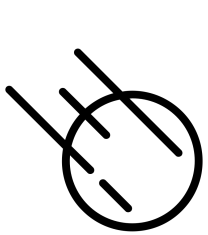
```
load system io.
load system type.

let x = type @tointeger (io @input "Please enter an integer: ").

if x < 0 do
    let x = 0.
    io @println "Negative, changed to zero".
elif x == 0 do
    io @println "Zero".
elif x == 1 do
    io @println "One".
else do
    io @println "Something else".
end
```

ln005/loop3.ast

```
load system io.
load system util

let indexes = ["first","second","third"].
let birds = ["turkey","duck","chicken"].

for (ix,bird) in util @zip (indexes,birds) do
    io @println ("the "+ix+" bird is a "+bird).
end
```

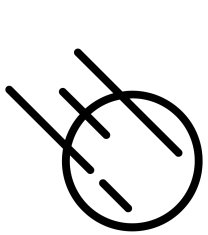
ln005/loop2.ast

```
load system io.

let l = ["bmw", "volkswagen", "mercedes"].

repeat
    let [element|l] = l.
    io @println element.
until l is [].
```



# Pattern Matching in Control Structures

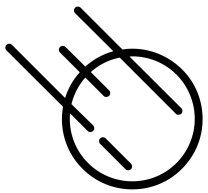
- Pattern matching lies at the heart of Asteroid
  - Imperative programming and pattern matching cannot really be separated in Asteroid even though they belong to different programming paradigms
- We saw some of Asteroid's pattern matching ability when we discussed the let statement.
- Some of the true power of pattern matching is revealed when using it within control structures



# Pattern Matching in If Statements

- In if statements we can use the is predicate to do pattern matching.
- Example: write a function that accepts a single value. If the value is a triple, print out its component values. If the value is a pair, print out its component values. Otherwise, print out an error message.

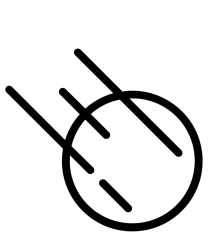
```
load system io.
load system type.

function print_components with value do
   if type @gettype value == "tuple" and len value == 3 do
      io @println ("Components of triple: "+value@0+","+value@1+","+value@2).
   elif type @gettype value == "tuple" and len value == 2 do
      io @println ("Components of pair: "+value@0+","+value@1).
   else do
      io @println "Error: Not a triple or pair".
   end
end

print_components (1,2).
```



# Pattern Matching in If Statements

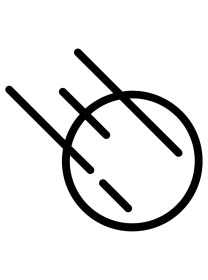
- This has a much nicer solution with pattern matching using the is predicate within the if clauses.

```
load system io.

function print_components with value do
    if value is (x,y,z) do
        io @println ("Components of triple: "+x+","+y+","+z).
    elif value is (x,y) do
        io @println ("Components of pair: "+x+","+y).
    else do
      io @println "Error: Not a triple or pair".
    end
end

print_components (1,2).
```



# Pattern Matching in For Loops

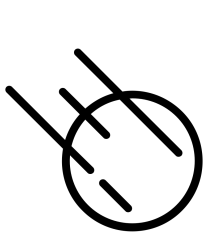
- Example: Write a program that constructs a list of Person objects where each object has a name and an age field. Then iterate over this list and write out the name of the persons whose names contain a lowercase ‘p’.



# Pattern Matching in For Loo

```
load system io.

structure Person with
    data name.
    data age.
end

-- define a list of persons
let people = [
    Person("George", 32),
    Person("Sophie", 46),
    Person("Oliver", 21)
].

-- print names that contain 'p'
for person in people do
    if "p" in person @name @explode () do
        io @println (person @name).
    end
end
```

ln005/pmloop1a.ast



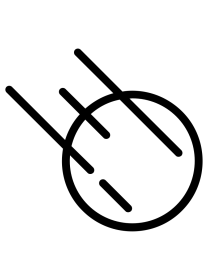
# Pattern Matching in For Loops

```
load system io.

structure Person with
    data name.
    data age.
end

-- define a list of persons
let people = [
    Person("George", 32),
    Person("Sophie", 46),
    Person("Oliver", 21)
].

-- print names that contain 'p'
for Person(name if name is ".*p.*", _) in people do
    io @println name.
end
```

Pattern matching

- Here we pattern match the Person object in the for loop,
- then use a regular expression to see if the name of that person matches our requirement that it contains a lower case ‘p’.
- The output is Sophie.

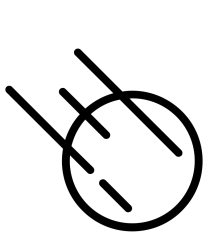
ln005/pmloop1b.ast



# Declarative Programming

- The differences between the non-pattern-match approach and the pattern-match approach are very subtle
- In general, pattern matching makes the code more readable because the developer's intentions and the structure of the data are directly visible
  - We often talk about **declarative programming**

> Declarative programming is a programming paradigm in which the programmer describes what the program should accomplish, rather than how to accomplish it. In a declarative program, the focus is on the logic of the computation, rather than the control flow.



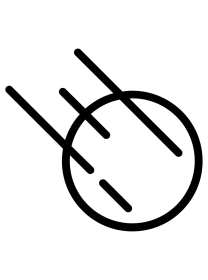
# Declarative Programming

- Pattern matching is considered a declarative programming technique.
- In pattern matching, the programmer specifies patterns that data can match against, rather than explicitly specifying how to manipulate the data.
  - This allows the programmer to express the logic of the computation in a more direct and readable way,
  - by describing what the expected inputs look like and what should be done with them,
  - rather than describing how to manipulate the data step-by-step.

```
-- print names that contain 'p'
for person in people do
    if "p" in person @name @explode () do
        io @println (person @name).
    end
end
```

Vs.

```
-- print names that contain 'p'
for Person(name if name is ".*p.*", _) in people do
    io @println name.
end
```



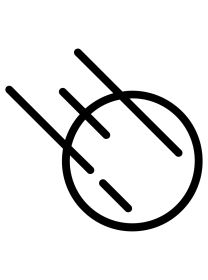
# Declarative Programming

- If we look carefully at our if-else example, we can see the declarative characteristics also
  - Patterns vs. data access/manipulation logic

```
if type @gettype value == "tuple" and len value == 3 do
   io @println ("Components of triple: "+value@0+","+value@1+","+value@2).
elif type @gettype value == "tuple" and len value == 2 do
   io @println ("Components of pair: "+value@0+","+value@1).
```

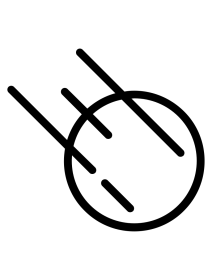
Vs.

```
if value is (x,y,z) do
    io @println ("Components of triple: "+x+","+y+","+z).
elif value is (x,y) do
    io @println ("Components of pair: "+x+","+y).
```



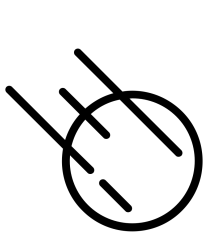
# Pattern Matching in Try-Catch Statements

- Exception handling in Asteroid is very similar to exception handling in many of the other modern programming languages available today with one major difference:
  - Exception objects can be any kind of object
  - In catch statements the exception objects are pattern matched



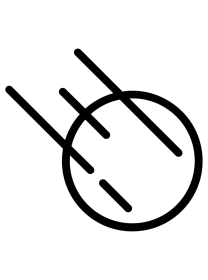
# Pattern Matching in Try-Catch Statements

- Idea: write a program that generates a random value between 0 and 1.  If the value is greater or equal to 0.5 then throw a Head object otherwise throw a Tail object.



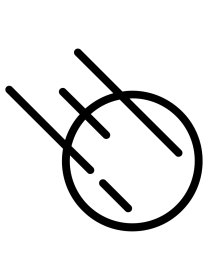
# Pattern Matching in Try-Catch Statements

```
load system io.
load system random.
load system type.

structure Head with
   data val.
end

structure Tail with
   data val.
end

try
   let i = random @random ().
   if i >= 0.5 do
      throw Head(i).
   else do
      throw Tail(i).
   end
catch Head(v) do
   io @println ("you win with "+type @tostring (v,type @stringformat (4,2))).
catch Tail(v) do
   io @println ("you loose with "+type @tostring (v,type @stringformat (4,2))).
end
```



# Pattern Matching in Try-Catch Statements

- Asteroid also provides built-in Exception objects
- All Asteroid and system errors are mapped into these object
- See the Asteroid user guide section “More on Exceptions”
  - asteroid-lang.readthedocs.io/en/latest/User%20Guide.html#more-on-exceptions



# Reading

- Asteroid user guide section “Flow of Control”
  - asteroid-lang.readthedocs.io/en/latest/User%20Guide.html#flow-of-control
- Asteroid reference guide
  - https://asteroid-lang.readthedocs.io/en/latest/Reference%20Guide.html