Functional Programming

- Here we explore aspects of functional programming in Asteroid
- Functional programming is defined by:
  - Programs exclusively consist of (recursive) definition of functions – no for-loops allowed!
  - Declarative approach to data structures – use pattern matching to access substructures.
  - Functions are first-class citizens of the language – giving rise to higher-order programming.
- We start with a look at pattern matching

Pattern Matching

In computer science, pattern matching is the act of checking a given sequence of tokens or a structure for the presence of the constituents of some pattern.

- Pattern matching in programming languages is an invention of functional programming languages.
- Easy access to sub-parts of a data structure.

Source: https://en.wikipedia.org/wiki/Pattern_matching

Tuples & Lists

- In Asteroid we can use pattern matching to extract the components of a tuple.

```
-- basic pattern matching on tuples

let x = (1,2). -- declare a tuple

-- extracting the components of x computationally
let a = x@0.
let b = x@1.
assert(a==1 and b==2).

-- extracting the components of x with pattern matching
let (p,q) = x.
assert(p==1 and q==2).
```

Pattern!

Tuples & Lists

- As an example of another programming language Python also supports pattern matching.

```python
# basic pattern matching on tuples

x = (1,2) # declare a tuple

# extracting the components of x computationally
a = x[0]
b = x[1]
assert(a==1 and b==2)

# extracting the components of x with pattern matching
(p,q) = x
assert(p==1 and q==2)
```

Tuples & Lists

- This also works with nested tuples.

```
-- pattern matching on nested structures

let (x,y) = ((1,2),3).
assert(x==(1,2) and y==3).
```

```
-- pattern matching on nested structures

let ((a,_),y) = ((1,2),3).
assert(a==1 and y==3).
```

Tuples & Lists

- The head-tail pattern [_ | _] is particularly useful to destructure lists – especially when we consider recursive algorithms.

```
-- the head-tail operator

let l = [1,2,3].
let [h|t] = l.
assert(h==1 and t==[2,3]).
```

```
let [h|t] = [1,2,3].
```

h = 1
first element of list

t = [2,3]
rest of list without first element

```
-- the head-tail operator

-- we can run the head-tail operator in reverse
-- to construct lists.
let [1,2,3] = [1|[2|[3|[]]]].

-- or in shorthand notation
let [1,2,3] = [1|2|3|[]].
```

Tuples & Lists

- In the functional programming language ML the head-tail operator is expressed with head::tail

```
val head::tail = [1,2,3];
```

head = 1
first element
of list

tail = [2,3]
rest of list
without first
element

Tuples & Lists

- In Python the head-tail operator is expressed with (head,*tail)

```
>>> (head,*tail) = [1,2,3]
>>> head
1
>>> tail
[2, 3]
>>>
```

Structures

- We can also pattern match on instances of structures – a convenient way to extract the components.

Pattern!

```
-- pattern matchine of structures

structure Point with
    data x.
    data y.
end

-- construct an instance of the structure
let p = Point(0,7).

-- extract the components of the structure with pattern matching
let Point(a,b) = p.
assert(a==0 and b==7).
```

Structures

- Structure pattern matching in Rust

```
// Defining a struct in Rust
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 }; // Initializing the struct

    let Point { x: a, y: b } = p; // Initializing variables a & b from the
struct fields
    assert_eq!(0, a);
    assert_eq!(7, b);
}
```

Source: https://oribenshir.github.io/afternoon_rusting/blog/enum-and-pattern-matching-part-1

Special Patterns

- Wild card patterns will match anything but do not retain what they matched.

```
-- wild card pattern matching

-- we want to ensure that the second component
-- is a pair but don't care what the pair looks like
let (x,(_,_)) = (1,(2,3)).                  -- succeeds
let (x,(_,_)) = (1,("hello","there")). -- succeeds
let (x,(_,_)) = (1,2,3).                    -- fails
```

Special Patterns

- Constraint Patterns allow us to use patterns in order to limit what is stored in a variable.

```
-- constraint patterns
load system "io".

-- define two different 'add' trees
let t1 = ("add",1,2).
let t2 = ("add",("*",3,2),2).

-- we use the pattern to make sure that it is
-- an 'add' tree that we store in 'tree'
let tree:("add",_,_) = t1.
println("matched tree: "+tree).

let tree:("add",_,_) = t2.
println("matched tree: "+tree).
```

In013 — -zsh — 65×12

```
lutz$ asteroid named-pattern.ast
matched tree: (add,1,2)
matched tree: (add,(*,3,2),2)
lutz$
```

Special Patterns

- Typematch patterns allow us to match on all values of a particular type.

```
-- typematch patterns

-- works with primitive types
let %integer = 1.
let %real = 2.0.
let %string = "Hello There!".

-- works with user defined types
structure A with
    data a.
    data b.
end

let %A = A(1,2).

-- typematch patterns together with constraint patterns
-- allow us to limit what is stored in a variable
let i:%integer = 1.
```

Special Patterns

- <u>Conditional patterns</u> allow us to test values during a pattern match.

```
-- conditional patterns

let i %if i > 0 = 1.  -- succeeds
let i %if i > 0 = -1. -- fails

-- we can combine typematch, constraint, and conditional
-- patterns to construct elaborate patterns.
-- Note: the parentheses are necessary
let (i:%integer) %if i > 0 = 1.  -- succeeds
let (i:%integer) %if i > 0 = -1. -- fails
```

Multi-Dispatch

- Multi-dispatch is a declarative way to look at function definitions using pattern matching on function arguments.
- Essentially, we are declaring one function body for each input pattern.

Multi-Dispatch

Computational Approach

```
-- traditional implementation of factorial

function fact with n do
  if n == 0 do
    return 1.
  else
    return n*fact(n-1).
  end
end

let v = fact(3).
assert(v==6).
```

Multi-Dispatch Approach

```
-- multi-dispatch implementation of factorial

function fact
  with 0 do
    return 1.
  orwith n do
    return n*fact(n-1).
  end

let v = fact(3).
assert(v==6).
```

Multi-Dispatch

- We have something similar in ML

```
fun fact n =
  if n=0 then 1 else n*fact(n-1);
```

```
fun fact 0 = 1
  | fact n = n*fact(n-1);
```

Recursion

- This is particularly interesting with more complicated input structures like lists.
- In functional programming iteration is not available therefore we need to use recursion
  - Base case
  - Recursive case

Recursion

```
-- counting elements on a list functional style

function count
  with [] do
    return 0.
  orwith [_|t] do
    return 1+count(t).
  end

assert(count [1,2,3] == 3).
```

```
-- double the values on a list

function double
  with [] do
    return [].
  orwith [h|t] do
    return [2*h] + double t.
  end

assert(double [1,2,3] == [2,4,6]).
```

Observation: The first ‘with’ clause is always the base case of the recursion.

Multi-Dispatch

Recursion with multiple base cases.

```
-- Quicksort

function qsort
    with [] do
        return [].
    orwith [a] do
        return [a].
    orwith [pivot|rest] do -- head-tail operator
        let less=[].
        let more=[].
        for e in rest do
            if e < pivot do
                let less = less + [e].
            else do
                let more = more + [e].
            end
        end

        return qsort less + [pivot] + qsort more.
    end

assert(qsort [3,2,1,0] == [0,1,2,3]).
```