

# Functional Programming

- Functional programming is defined by:
  - Programs exclusively consist of recursive definitions of functions
  - Everything is a value – no statements allowed
    - We do allow:
      - Function definition statements 😊
      - Let statements for giving names to expressions
      - Return statements
  - Declarative approach to data via the use of pattern matching.
  - Functions as first-class citizens
    - This gives rise to higher-order programming.
- Functional Asteroid is called with '-F' switch
  - `asteroid -F <program>`

# The Factorial Revisited

- Let's start with something simple: Factorial

```
1  -- factorial with if-stmt
2
3  function fact with n do
4    if n == 1 do
5      return 1.
6    else
7      return n * fact(n-1).
8    end
9  end
10
11 assert(fact(3) == 6).
```

The problem is that if statements are not supported in the functional programming paradigm – they do not compute a value!

```
[lutz$ asteroid -F fact-stmt.ast
error: fact-stmt.ast: 4: if statement is not supported in functional mode
lutz$ █
```

# The Factorial Revisited

- Let's rewrite this so everything is a value

We use a conditional expression to compute the return value

```
1  -- factorial with if-exp
2
3  function fact with n do
4    return 1 if n==1 else n*fact(n-1).
5  end
6
7  assert(fact(3) == 6).
```

Since functions are only allowed to compute return values there is no need for the explicit 'return'.

```
1  -- factorial with if-exp
2
3  function fact with n do
4    1 if n==1 else n*fact(n-1).
5  end
6
7  assert(fact(3) == 6).
```

```
[lutz$ asteroid -F fact-exp.ast
lutz$ █
```

# SML

- SML is one of the classic functional languages next to Lisp.
- A web-based implementation of SML is available here,
  - <https://sosml.org>

## Asteroid

```
1  -- factorial with if-exp
2
3  function fact with n do
4    1 if n==1 else n*fact(n-1).
5  end
6
7  assert(fact(3) == 6).
```

## SML

```
(* factorial using if expression *)
fun fact n = if n=1 then 1 else n*fact(n-1);

fact(3) = 6;
```

# Lists: Listsum

- Let's see how functional programming works with lists
  - Remember: no loops!
  - Everything has to be done with recursion
- Program: Assume we are given a list of integer values, sum all the integer values on the list. E.g.  $[1,2,3] \Rightarrow 6$
- We need to use recursion.
  - Base case
  - Recursive step

# Lists: Listsum

- Notice the recursion in our solution,
  - Base case: `[] => 0`
  - Recursive step: pull the first element off the list and add it to the result of the recursive call over the rest of the list,
    - `hd(l)+listsum(tl(l))`
    - `hd` – first element
    - `tl` – rest of list

```
1  -- sum the integer values on a list
2
3  function listsum with l do
4    0 if l==[] else hd(l)+listsum(tl(l)).
5  end
6
7  assert(listsum([1,2,3]) == 6).
```

```
[lutz$ asteroid -F list-sum.ast
lutz$
```

# SML & Listsum

## Asteroid

```
1  -- sum the integer values on a list
2
3  function listsum with l do
4    0 if l==[] else hd(l)+listsum(tl(l)).
5  end
6
7  assert(listsum([1,2,3]) == 6).
```

## SML

```
(* sum integer values on a list *)
fun listsum l = if l=[] then 0 else hd(l)+listsum(tl(l));

listsum([1,2,3]) = 6;
```

# Class Exercise

- Write a program that given a list will count the number of elements on the list.
  - E.g. `[1,2,3] => 3`, and `[] => 0`
- Write a program that given a list of integer values **will return a list** where each value on the list is double the value of the original value.
  - E.g. `[1,2,3] => [2,4,6]`, and `[] => []`
- All programs need to be written in functional Asteroid and need to be run with the '-F' flag in place.



# Multi-Dispatch

- Since most functional programs consist of recursive functions all these functions will have a top-level 'if-else' expression to deal with the base vs recursive step.
- That style of programming gets tiring very fast and the code is not very readable.
- The solution: Multi-Dispatch
  - Introduce one function body for each of the steps.

# Multi-Dispatch

Instead of this...

```
1  -- factorial with if-exp
2
3  function fact with n do
4    1 if n==1 else n*fact(n-1).
5  end
6
7  assert(fact(3) == 6).
```

Advantage: implicit testing  
or pattern matching of the  
function arguments!

Do this...

```
1  -- factorial with multi-dispatch
2
3  function fact
4    with 1 do -- function argument == 1
5      1
6    with n do -- function argument /= 1
7      n*fact(n-1).
8    end
9
10  assert(fact(3) == 6).
```

# Multi-Dispatch: SML

Instead of this...

```
(* factorial using if expression *)  
fun fact n = if n=1 then 1 else n*fact(n-1);  
  
fact(3) = 6;
```

Do this...

```
(* factorial with multi-dispatch *)  
fun fact 1 = 1  
  | fact n = n*fact(n-1);  
  
fact(3)=6;
```


# Multi-Dispatch

Instead of this...

```
1  -- sum the integer values on a list
2
3  function listsum with l do
4    0 if l==[] else hd(l)+listsum(tl(l)).
5  end
6
7  assert(listsum([1,2,3]) == 6).
```

Notice that we can pattern match on the structure of a list: E.g. []

Do this...

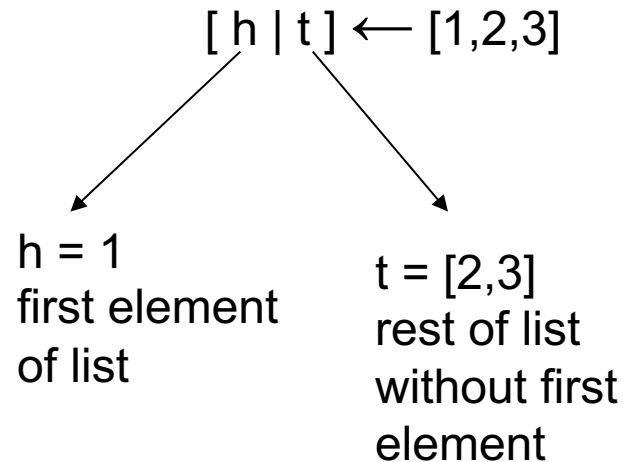
```
1  -- sum the integer values on a list
2
3  function listsum
4    with [] do 
5      0
6    with l do
7      hd(l)+listsum(tl(l)).
8    end
9
10  assert(listsum([1,2,3]) == 6).
```

# Pattern Matching

- In programming values have structure
  - Lists are comprised of a sequence of elements
  - Pairs are made up of two ordered values: first component and second component
  - Integers are single values without a decimal part
- In pattern matching we state the expected structure of a value as a pattern possibly with variables
  - If the pattern matches the expected value, then we say that the pattern-match was successful, and variables will be bound to parts of the value that they matched.
  - Example:  $(a,b) \leftarrow (1,2)$  with  $a=1$  and  $b=2$
  - Example:  $1 \leftarrow 1$
  - Example:  $x \leftarrow 3$  with  $x=3$

# Head-Tail Pattern Matching

- Instead of using 'hd' and 'tl' we can use pattern matching with the head-tail pattern '[ h | t ]'.



# Head-Tail Pattern Matching

- In listsum the head-tail pattern takes care of the analysis of the list!

Instead of this...

```
1  -- sum the integer values on a list
2
3  function listsum
4    with [] do
5      0
6    with l do
7      hd(l)+listsum(tl(l)).
8    end
9
10 assert(listsum([1,2,3]) == 6).
```

Do this...

```
1  -- sum the integer values on a list
2
3  function listsum
4    with [] do
5      0
6    with [h|t] do
7      h+listsum(t).
8    end
9
10 assert(listsum([1,2,3]) == 6).
```

# Head-Tail Pattern Matching

- The hallmark of this multi-dispatch approach is that the interpreter does a lot of work for you for free:
  - It executes the body that matches the function argument
  - If the head-tail pattern matches the function argument it instantiates the first element in variable *h* and the rest of the list in variable *t*.

```
1  -- sum the integer values on a list
2
3  function listsum
4    with [] do
5      0
6    with [h|t] do
7      h+listsum(t).
8    end
9
10 assert(listsum([1,2,3]) == 6).
```



# Head-Tail Pattern Matching

We went from this...

```
1  -- sum the integer values on a list
2
3  function listsum with l do
4    0 if l==[] else hd(l)+listsum(tl(l)).
5  end
6
7  assert(listsum([1,2,3]) == 6).
```

To this...

```
1  -- sum the integer values on a list
2
3  function listsum
4    with [] do
5      0
6    with [h|t] do
7      h+listsum(t).
8    end
9
10  assert(listsum([1,2,3]) == 6).
```

# Head-Tail Pattern Matching: SML

- Head-Tail pattern matching is also available in SML

```
1  -- sum the integer values on a list
2
3  function listsum
4    with [] do
5      0
6    with [h|t] do
7      h+listsum(t).
8    end
9
10 assert(listsum([1,2,3]) == 6).
```

```
(* listsum head-tail pattern matching *)|
fun listsum [] = 0
  | listsum (h::t) = h+listsum(t);

listsum([1,2,3])=6;
```

# Head-Tail Pattern Matching: Python

- Python also supports head-tail pattern matching...

```
Python 3.9.6 (default, Sep 13 2022, 22:03:16)
[Clang 14.0.0 (clang-1400.0.29.102)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> (h,*t) = [1,2,3]
>>> h
1
>>> t
[2, 3]
>>> █
```

# Wildcard Pattern

- If we need to match a value but we don't care what that value is, we can use a wildcard pattern '\_'

```
1  -- wild card pattern
2
3  function zero
4    with 0 do
5      "zero"
6    with _ do -- wild card
7      "something else"
8  end
9
10 assert(zero(0) == "zero").
11 assert(zero(1) == "something else").
```

```
1  -- wild card pattern in structures
2
3  function pair
4    with (1,1) do
5      "pair with two ones"
6    with (a,_) do -- wild card within structure
7      "pair with first component: "+a
8    with _ do
9      "not a pair"
10  end
11
12 assert(pair (1,1) == "pair with two ones").
13 assert(pair (3,4) == "pair with first component: 3").
14 assert(pair (1,2,3) == "not a pair").
```

# Type Patterns

- Type patterns match all the values of a particular type.
- Type patterns are written with the ‘%’ followed by the type name.
- A type pattern that matches all integer values is %integer.
- Type patterns can appear anywhere where patterns can appear.
- All built-in types are supported: %integer, %real, %string, %list, %tuple, %boolean
- User defined type patterns are %<name of the structure>.
  - For example if you created a structure called MyStruct then the associated type pattern is %MyStruct and will only match objects instantiated from MyStruct

```
1  -- a function that determines whether a value
2  -- is an integer value or not
3
4  function isinteger
5      with %integer do
6          true
7      with _ do
8          false
9  end
10
11  assert(isinteger(1) == true).
12  assert(isinteger(1.0) == false).
```

# Conditional Patterns

- We can limit the values that a variable can match by using a special conditional pattern: `<var> : <pattern>`
  - `x:%real` – states that 'x' can only match floating point values.
  - `q:(%integer,%integer)` – states the 'q' can only match pairs of integer values.

```
1  -- the typed version of factorial
2  -- factorial is only defined over the integers
3
4  load system io.
5
6  function fact
7      with 1 do
8          1
9          with n:%integer do
10             n*fact(n-1)
11         with _ do
12             throw Error "not an integer value".
13     end
14
15     assert(fact(3) == 6).
16     try
17         fact(3.0)
18     catch s do
19         io @println s. -- catch the error
20     end
```

# Structural Patterns

- Structural patterns means pattern matching on structure in addition to values.
- On the previous slide we saw one instance of that:
  - (%integer,%integer) – match pairs of integer values.

# Structural Patterns

- The empty list '[ ]', single element list '[e]', and the head-tail pattern '[x|y]' are also structural patterns...

```
function halve
  with [] do
    ([],[])
  with [a] do
    ([a],[])
  with [a|b|rest] do
    let (llist,rlist) = halve(rest).
    ([a]+llist,[b]+rlist)
end
```

Here [ a | b | rest ] is the same as [ a | [ b | rest ] ].



# Structural Patterns

- We can nest arbitrary structures as patterns...

```
function merge
  with ([],rlist) do
    | rlist
    with (llist,[]) do
      | llist
      with ([a|llist],[b|rlist]) do
        | [a]+merge(llist,[b]+rlist) if a < b
        | else [b]+merge([a]+llist,rlist)
      end
    end
  end
```

# Patterns & Let

- Even though the 'let' statement looks like an assignment statement it is actually a pattern-match statement of the form,
  - let <pattern> = <value>
- It takes the value on the right and pattern-matches it against the pattern on the left.
- If the pattern contains variables, they will be instantiated in the current namespace.
- All patterns we have discussed so far are also valid as let statement patterns

```
1  -- examples of the let statement
2
3  let x = 1.  -- the variable x is the simplest pattern possible
4  let 1 = 1.  -- the 1 on the left is the pattern, on the right the value
5  let x:%integer = 1. -- type patterns work here too
6  let (x,y) = (1,2). -- pattern instantiated x=1 and y=2
7  let ((a,b),(c,d)) = ((1,2),(3,4)). -- pair of pairs
8  let [a|b] = [1,2,3]. -- head-tail pattern match
```

# The MergeSort

- Putting this all together – the MergeSort

```
1  -- the mergesort
2
3  load system io.
4
5  function mergesort
6    with [] do
7      []
8    with [a] do
9      [a]
10   with l do
11     function halve
12       with [] do
13         ([],[])
14       with [a] do
15         ([a],[])
16       with [a|b|rest] do
17         let (l1list,r1list) = halve(rest).
18         ([a]+l1list,[b]+r1list)
19     end
20   function merge
21     with ([],rlist) do
22       rlist
23     with (l1list,[]) do
24       l1list
25     with ([a|l1list],[b|r1list]) do
26       [a]+merge(l1list,[b]+r1list) if a < b
27       else [b]+merge([a]+l1list,r1list)
28     end
29     let (x,y) = halve(l).
30     merge(mergesort(x),mergesort(y)).
31   end
32
33   io @println(mergesort([3,2,1,0])).
```

# Reading

- Asteroid User Guide

- Functions

- <https://asteroid-lang.readthedocs.io/en/latest/User%20Guide.html#functions>

- Pattern Matching

- <https://asteroid-lang.readthedocs.io/en/latest/User%20Guide.html#pattern-matching>

# Class Exercise

- Rewrite your solutions to the previous class exercise in the multi-dispatch style with pattern matching on the arguments.
- Work with the teammates from the previous exercise.
- You can start with the solution I posted in BrightSpace.

# Assignment

- All the example programs in this slide pack are available in the VM,
  - <https://replit.com/@lutzhamel/asteroid-csc301-ln013>
- Assignment #3 – see BrightSpace