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>
Let’s start with something simple: Factorial

```plaintext
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!
The Factorial Revisited

Let’s rewrite this so everything is a value

```plaintext
1 -- factorial with if-exp

2 function fact with n do

3 return 1 if n==1 else n*fact(n-1).

4 end

5

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

We use a conditional expression to compute the return value.

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

```
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

```plaintext
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

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

fact(3) = 6;
```
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] => 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).
```
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.
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...

```plaintext
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).
```

Do this...

```plaintext
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).
```

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

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

fact(3) = 6;
```

Do this…

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

fact(3)=6;
```
Multi-Dispatch

Instead of this...

```plaintext
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).
```

Do this...

```plaintext
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).
```

Notice that we can pattern match on the structure of a list: E.g. []
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 \(x=3\)
Instead of using ‘hd’ and ‘tl’ we can use pattern matching with the head-tail pattern ‘[ h | t ]’.

\[
\begin{align*}
[ h \mid t ] & \leftarrow [1,2,3] \\
\text{h = 1} & \text{first element of list} \\
\text{t = [2,3]} & \text{rest of list without first element}
\end{align*}
\]
Head-Tail Pattern Matching

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

Instead of this…

```plaintext
-- sum the integer values on a list
function listsum
  with [] do
    0
  with l do
    hd(l)+listsum(tl(l)).
end

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

Do this…

```plaintext
-- sum the integer values on a list
function listsum
  with [] do
    0
  with [h|t] do
    h+listsum(t).
end

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.

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

```python
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]
```
Functional Style Programming in Python

- A recursive program with pattern matching in Python
- Functional programming is more explicit about the intentions of a program
- This is often called declarative programming
- Functional and logic programming are considered declarative programming paradigms

```
def listsum(l):
    acc = 0
    for v in l:
        acc += v
    return acc

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

```
def listsum(l):
    match l:
        case [] :
            return 0
        case (h,*t) :
            return h+listsum(t)

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

Imperative Programming  Functional Programming
If we need to match a value but we don’t care what that value is, we can use a wildcard pattern `_'

```plaintext
1 -- wild card pattern
2
3 function zero
4   with 0 do
5     "zero"
6   with _ do -- wild card
7     "something else"
8 end

1 assert(zero(0) == "zero").
2 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

1 assert(pair (1,1) == "pair with two ones").
2 assert(pair (3,4) == "pair with first component: 3").
3 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

```plaintext
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).
```
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.

```plaintext
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.
The empty list ‘[ ]’, single element list ‘[e]’, and the head-tail pattern ‘[x|y]’ are also structural patterns…

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

- We can nest arbitrary structures as patterns...

```plaintext
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
```
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:uinteger = 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

```haskell
-- the mergesort
load system io.

function mergesort
  with [] do
    []
  with [a] do
    [a]
  with l do
    function halve
      with [] do
        ([],[])  
      with [a] do
        ([a],[])  
      with [a|b|rest] do
        let (llist,rlist) = halve(rest).
        ([a]+llist,[b]+rlist)
    end
    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
    let (x,y) = halve(l).
    merge(mergesort(x),mergesort(y)).
  end
io @println(mergesort([3,2,1,0])).
```
Reading

Asteroid User Guide

- Functions

- Pattern Matching
Class Exercise

- Rewrite your solutions to the previous class exercise in the multi-dispatch style with pattern matching on the arguments.
  - 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 [ ] => [ ]
Assignment #3 – see BrightSpace