

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: []

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).
```

Head-Tail Pattern Matching

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

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 approach is that the interpreter does a lot of work for you for free:
 - It executes the body that matches the function argument
 - It pattern matches the head-tail pattern to the function argument instantiating 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;
```


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").
```

The MergeSort

- Something a bit more complicated.

```
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 (l1list:[],r1list) do
22         r1list
23       with (l1list,r1list:[]) do
24         l1list
25       with (first:[a|l1list],second:[b|r1list]) do
26         [a]+merge(l1list,second)
27         if a < b
28         else [b]+merge(first,r1list)
29       end
30     let (x,y) = halve(l).
31     merge(mergesort(x),mergesort(y)).
32   end
33
34   io @println(mergesort([3,2,1,0])).
```