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

```
-- factorial with if-stmt

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

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

```
-- factorial with if-exp

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

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

We use a conditional expression to compute the return value

```
-- factorial with if-exp

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

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

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

```
-- factorial with if-exp

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

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] => 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

```
-- sum the integer values on a list

function listsum with l do
  0 if l==[] else hd(l)+listsum(tl(l)).
end

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

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

SML & Listsum

Asteroid

```
-- sum the integer values on a list

function listsum with l do
  0 if l==[] else hd(l)+listsum(tl(l)).
end

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…

```
-- factorial with if-exp

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

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

Do this…

```
-- factorial with multi-dispatch

function fact
  with 1 do -- function argument == 1
    1
  with n do -- function argument =/= 1
    n*fact(n-1).
end

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

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

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…

```
-- sum the integer values on a list

function listsum with l do
  0 if l==[] else hd(l)+listsum(tl(l)).
end

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

Do this…

```
-- 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).
```

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

Head-Tail Pattern Matching

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

Instead of this…

```
-- 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…

```
-- 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 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.

```
-- 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

We went from this…

```
-- sum the integer values on a list

function listsum with l do
  0 if l==[] else hd(l)+listsum(tl(l)).
end

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

To this…

```
-- 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: SML

- Head-Tail pattern matching is also available in SML

```
-- 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).
```

```
(* 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 '_'

```
-- wild card pattern

function zero
  with 0 do
    "zero"
  with _ do -- wild card
    "something else"
end

assert(zero(0) == "zero").
assert(zero(1) == "something else").
```

```
-- wild card pattern in structures

function pair
  with (1,1) do
    "pair with two ones"
  with (a,_) do -- wild card within structure
    "pair with first component: "+a
  with _ do
    "not a pair"
end

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

```
-- a function that determines whether a value
-- is an integer value or not

function isinteger
  with %integer do
    true
  with _ do
    false
end

assert(isinteger(1) == true).
assert(isinteger(1.0) == false).
```

Type Patterns

- We can limit the values that a variable can match by using a special conditional pattern: <var> : <type pattern>
  - E.g. x:%real – states that ‘x’ can only match floating point values.

```
-- the typed version of factorial
-- factorial is only defined over the integers

load system io.

function fact
   with 1 do
      1
   with n:%integer do
      n*fact(n-1)
   with _ do
      throw Error "not an integer value".
end

assert(fact(3) == 6).
try
   fact(3.0)
catch s do
   io @println s. -- catch the error
end
```

General Structural Patterns

- We have already seen the empty list '[ ]', single element list '[e]', and the head-tail pattern '[x|y]' as 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

- But we can nest arbitrary structures for 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
```

Patterns & Let

- Even though the ‘let’ statement looks like an assignment statement it is acuatally a pattern match statement of the form,
  - let <pattern> = <value>
- It tries to take the value on the right and pattern-match 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

```
-- examples of the let statement

let x = 1.  -- the variable x is the simplest pattern possible
let 1 = 1.  -- the 1 on the left is the pattern, on the right the value
let x:%integer = 1. -- type patterns work here too
let (x,y) = (1,2). -- pattern instantiated x=1 and y=2
let ((a,b),(c,d)) = ((1,2),(3,4)). -- pair of pairs
```

The MergeSort

- Putting this all together – the MergeSort

```
-- 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])).
```