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$
```

Factorial Revisited

```
-- 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).
```

```
-- factorial with if-exp

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

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

```
-- factorial with if-exp

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

assert(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$
```

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

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).
```