

CSc 372

# Comparative Programming Languages

## 5 : Haskell — Function Definitions

Department of Computer Science  
University of Arizona

[collberg@gmail.com](mailto:collberg@gmail.com)

Copyright © 2011 Christian Collberg

# Defining Functions

- When programming in a functional language we have basically two techniques to choose from when defining a new function:
  - ① Recursion
  - ② Composition
- Recursion is often used for basic “low-level” functions, such that might be defined in a function library.
- Composition (which we will cover later) is used to combine such basic functions into more powerful ones.
- Recursion is closely related to **proof by induction**.

# Defining Functions...

- Here's the ubiquitous factorial function:

```
fact :: Int -> Int
fact n = if n == 0 then
          1
        else
          n * fact (n-1)
```

- The first part of a function definition is the **type signature**, which gives the **domain** and **range** of the function:

```
fact :: Int -> Int
```

- The second part of the definition is the **function declaration**, the implementation of the function:

```
fact n = if n == 0 then ...
```

# Defining Functions...

- The syntax of a type signature is

```
fun_name :: argument_types
```

fact takes one integer input argument and returns one integer result.

- The syntax of function declarations:

```
fun_name param_names = fun_body
```

# Conditional Expressions

- `if e1 then e2 else e3` is a **conditional expression** that returns the value of `e2` if `e1` evaluates to `True`. If `e1` evaluates to `False`, then the value of `e3` is returned. Examples:

`if True then 5 else 6`             $\Rightarrow$  5

`if False then 5 else 6`            $\Rightarrow$  6

`if 1==2 then 5 else 6`            $\Rightarrow$  6

`5 + if 1==1 then 3 else 2`        $\Rightarrow$  8

- Note that this is different from Java's or C's **if-statement**, but just like their **ternary operator** `?::`:

```
int max = (x>y)?x:y;
```

# Conditional Expressions...

- Example:

```
abs :: Int -> Int
```

```
abs n = if n>0 then n else -n
```

```
sign :: Int -> Int
```

```
sign n = if n<0 then -1 else  
         if n==0 then 0 else 1
```

- Unlike in C and Java, you can't leave off the else-part!

# Guarded Equations

- An alternative way to define conditional execution is to use guards:

```
abs :: Int -> Int
abs n | n >= 0 = n
      | otherwise = -n
```

```
sign :: Int -> Int
sign n | n < 0 = -1
      | n == 0 = 0
      | otherwise = 1
```

- The pipe symbol is read **such that**.
- **otherwise** is defined to be **True**.
- Guards are often easier to read — it's also easier to verify that you have covered all cases.

# Defining Functions...

- `fact` is defined recursively, i.e. the function body contains an application of the function itself.
- The syntax of function application is: `fun_name arg`. This syntax is known as “juxtaposition”.
- We will discuss multi-argument functions later. For now, this is what a multi-argument function application (“call”) looks like:

```
fun_name arg_1 arg_2 ... arg_n
```

- Function application examples:

```
fact 1      ⇒ 1
```

```
fact 5      ⇒ 120
```

```
fact (3+2)  ⇒ 120
```



# Multi-Argument Functions

- A simple way (but usually not **the right way**) of defining an multi-argument function is to use tuples:

```
add :: (Int,Int) -> Int  
add (x,y) = x+y
```

```
> add (40,2)
```

```
42
```

- Later, we'll learn about **Curried Functions**.

# The `error` Function

- `error string` can be used to generate an error message and terminate a computation.
- This is similar to Java's exception mechanism, but a lot less advanced.

```
f :: Int -> Int
f n = if n < 0 then
      error "illegal argument"
    else if n <= 1 then
      1
    else
      n * f (n-1)
```

```
> f (-1)
```

```
Program error: illegal argument
```

# Layout

- A function definition is finished by the first line not indented more than the start of the definition

```
myfunc :: Int -> Int
myfunc x = if x == 0 then
            0 else 99
```

```
myfunc :: Int -> Int
        myfunc x = if x == 0 then
                    0 else 99
```

```
myfunc :: Int -> Int
myfunc x = if x == 0 then
0 else 99
```

- The last two generate a **Syntax error in expression** when the function is loaded.

# Function Application

- Function application (“calling a function with a particular argument”) has higher priority than any other operator.
- In math (and Java) we use parentheses to include arguments; in Haskell no parentheses are needed.

> `f a + b`

means

> `(f a) + b`

since function application binds harder than plus.

# Function Application...

- Here's a comparison between mathematical notations and Haskell:

Math	Haskell
$f(x)$	<code>f x</code>
$f(x, y)$	<code>f x y</code>
$f(g(x))$	<code>f (g x)</code>
$f(x, g(y))$	<code>f x (g y)</code>
$f(x)g(y)$	<code>f x * g y</code>

# Recursive Functions

# Simple Recursive Functions

- Typically, a recursive function definition consists of a **guard** (a boolean expression), a **base case** (evaluated when the guard is True), and a **general case** (evaluated when the guard is False).

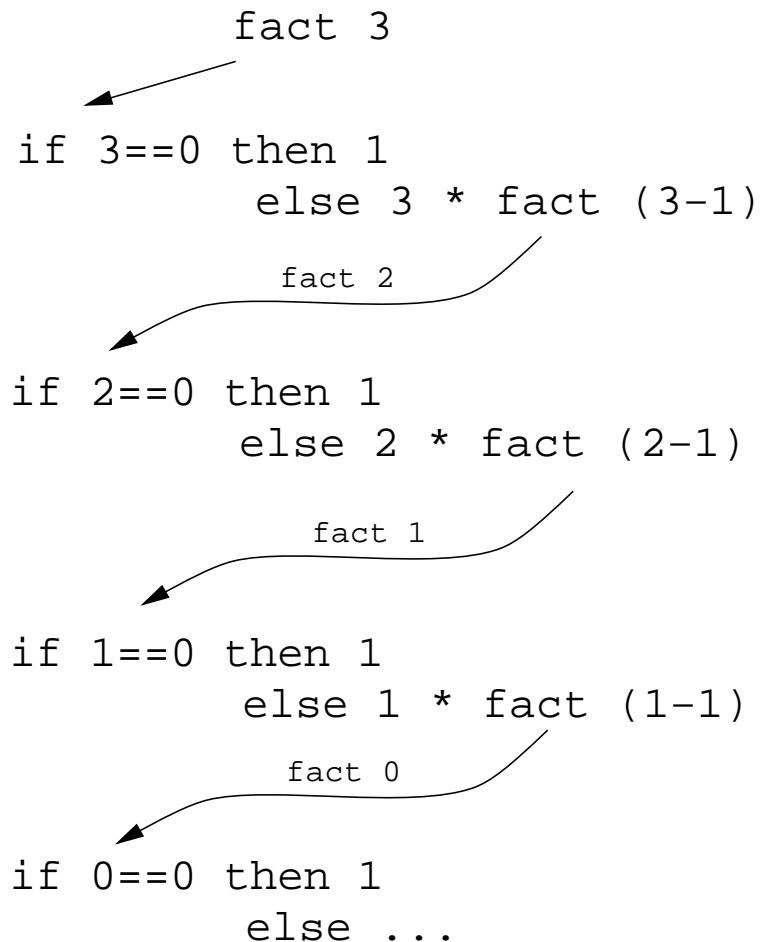
```
fact n =  
  if n == 0 then           ⇐ guard  
    1                       ⇐ base case  
  else  
    n * fact (n-1)         ⇐ general case
```

# Simulating Recursive Functions

- We can visualize the evaluation of `fact 3` using a **tree** view, **box** view, or **reduction** view.
- The tree and box views emphasize the flow-of-control from one level of recursion to the next
- The reduction view emphasizes the **substitution** steps that the hugs interpreter goes through when evaluating a function. In our notation boxed subexpressions are substituted or evaluated in the next reduction.
- Note that the Haskell interpreter may not go through exactly the same steps as shown in our simulations. More about this later.

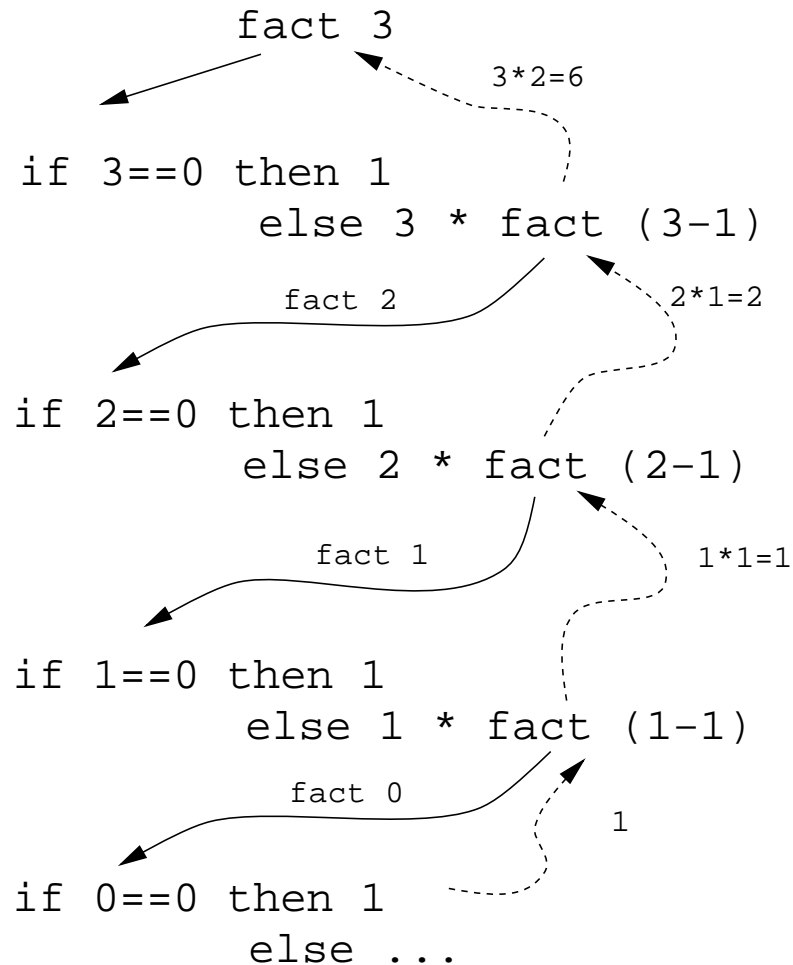


# Tree View of fact 3



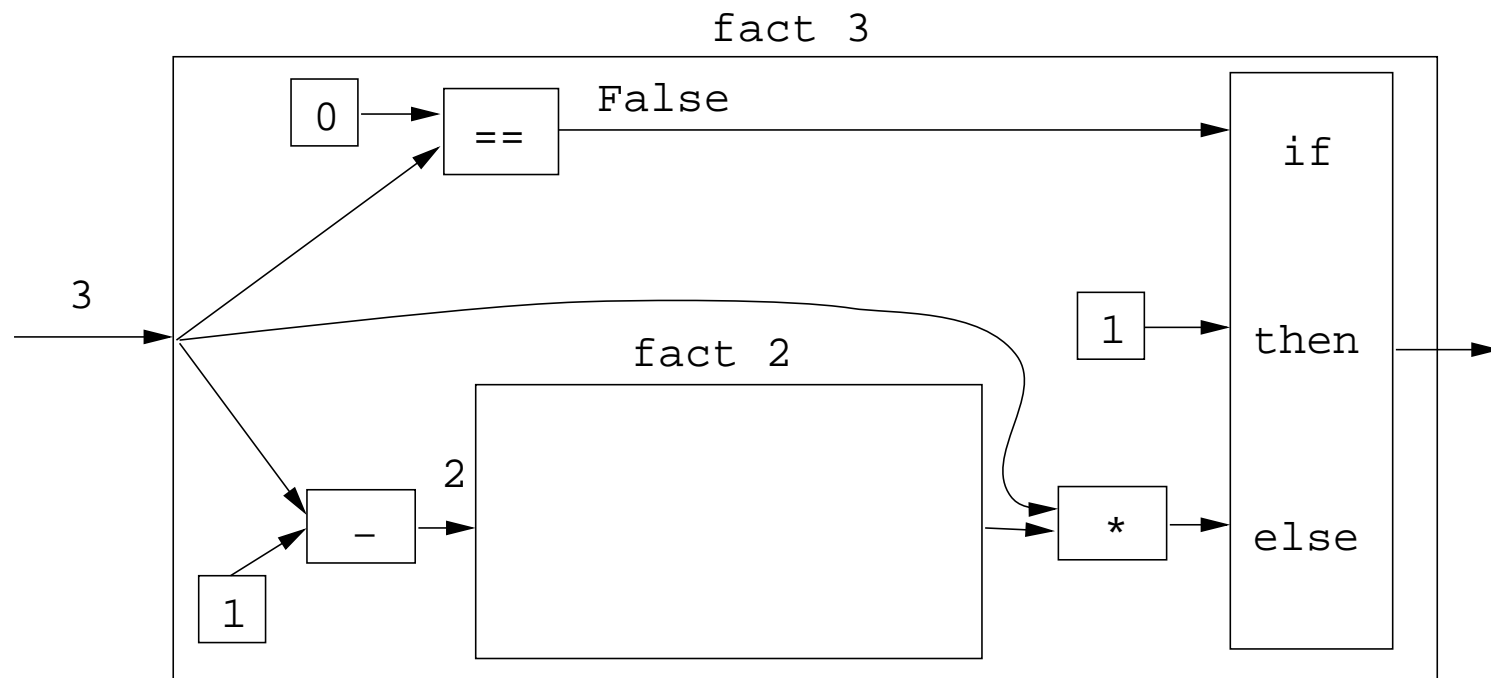
- This is a **Tree View** of **fact 3**.
- We keep going deeper into the recursion (evaluating the **general case**) until the **guard** is evaluated to True.

# Tree View of fact 3

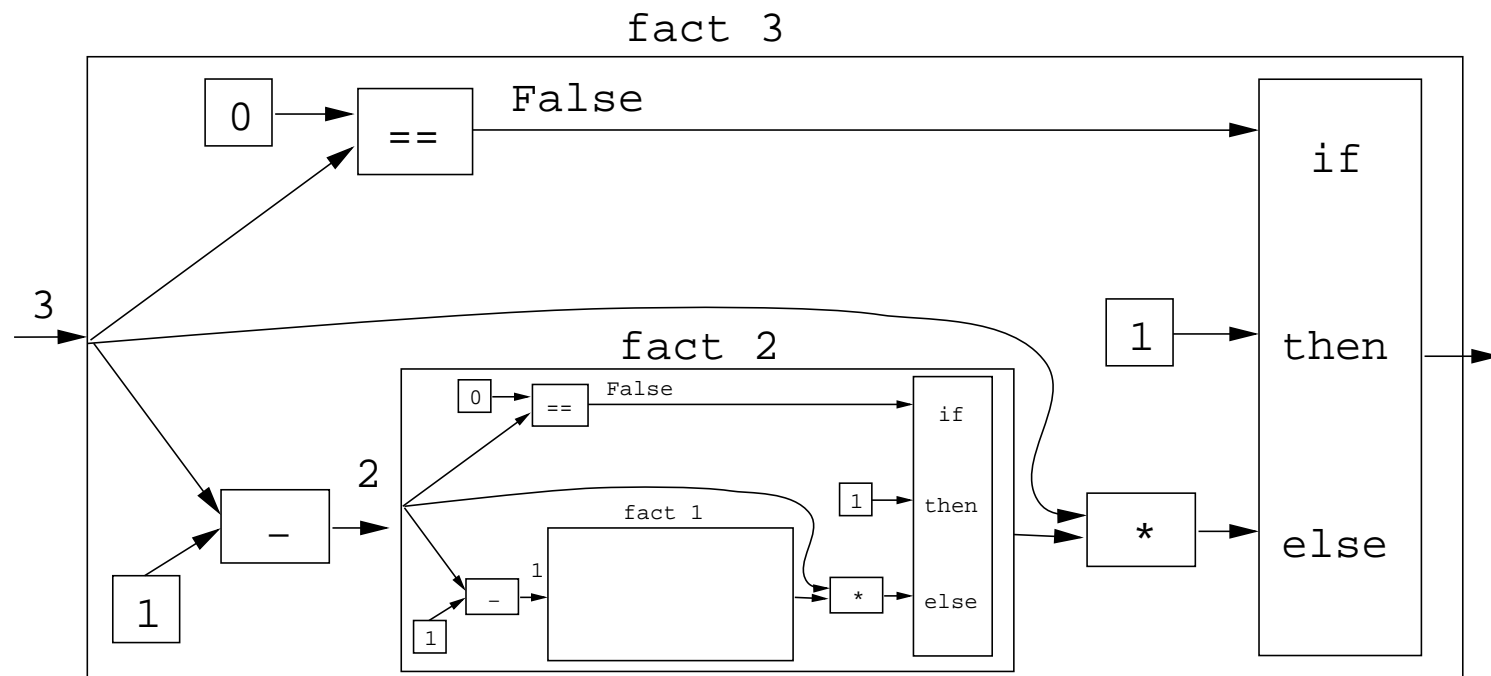


- When the guard is True we evaluate the **base case** and return back up through the layers of recursion.

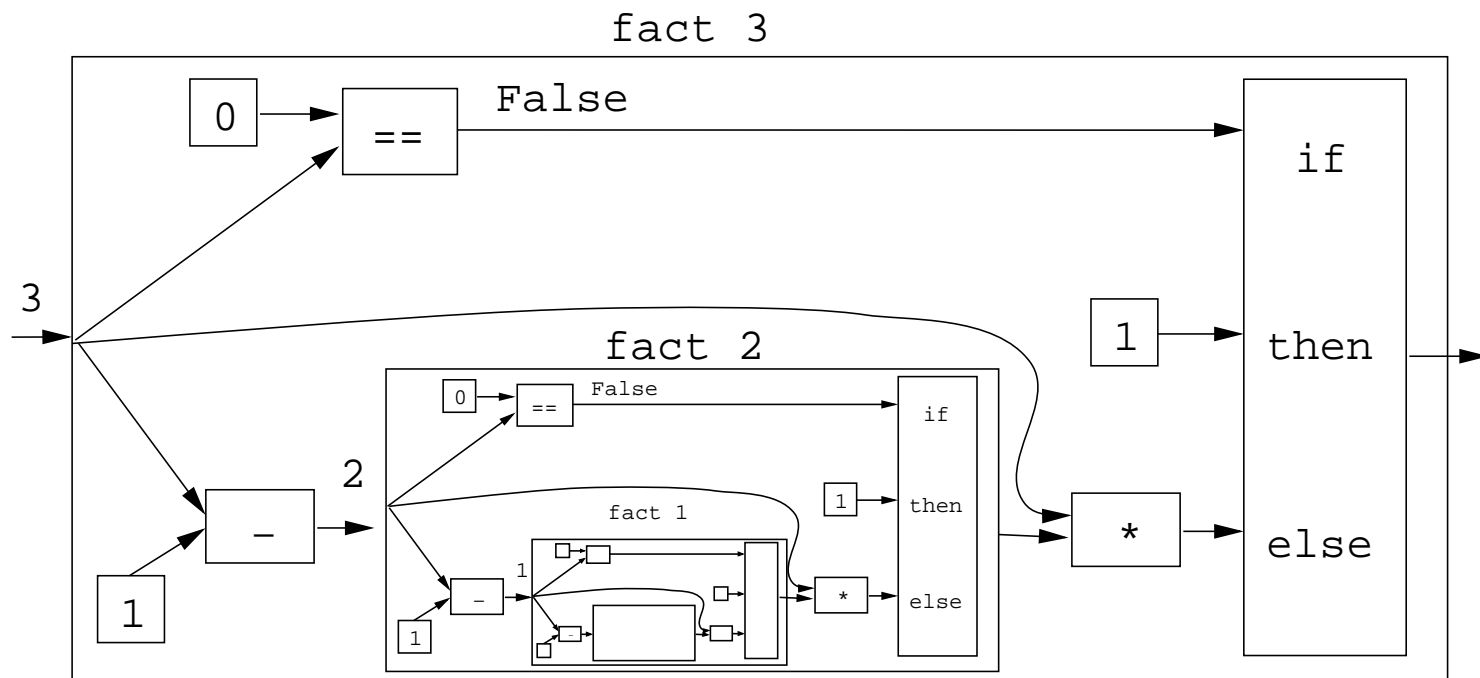
# Box View of fact 3



# Box View of fact 3...



# Box View of fact 3...



# Reduction View of fact 3

```
fact 3 ⇒  
if 3 == 0 then 1 else 3 * fact (3-1) ⇒  
if False then 1 else 3 * fact (3-1) ⇒  
3 * fact (3-1) ⇒  
3 * fact 2 ⇒  
3 * if 2 == 0 then 1 else 2 * fact (2-1) ⇒  
3 * if False then 1 else 2 * fact (2-1) ⇒  
3 * (2 * fact (2-1)) ⇒  
3 * (2 * fact 1) ⇒  
3 * (2 * if 1 == 0 then 1 else 1 * fact (1-1))  
⇒ ...
```

# Reduction View of fact 3...

```
3 * (2 * if 1 == 0 then 1 else 1 * fact (1-1)) ⇒  
3 * (2 * if False then 1 else 1 * fact (1-1)) ⇒  
3 * (2 * (1 * fact (1-1))) ⇒  
3 * (2 * (1 * fact 0)) ⇒  
3 * (2 * (1 * if 0 == 0 then 1 else 0 * fact (0-1))) ⇒  
3 * (2 * (1 * if True then 1 else 0 * fact (0-1))) ⇒  
3 * (2 * (1 * 1)) ⇒  
3 * (2 * 1) ⇒  
3 * 2 ⇒  
6
```

# Recursion Over Lists

- In the `fact` function the guard was `n==0`, and the recursive step was `fact(n-1)`. I.e. we subtracted 1 from `fact`'s argument to make a simpler (smaller) recursive case.
- We can do something similar to recurse over a list:
  - ① The guard will often be `n==[ ]` (other tests are of course possible).
  - ② To get a smaller list to recurse over, we often split the list into its head and tail, `head:tail`.
  - ③ The recursive function application will often be on the tail, `f tail`.



# The length Function

\_\_\_\_\_ In English: \_\_\_\_\_

*The length of the empty list [ ] is zero. The length of a non-empty list S is one plus the length of the tail of S.*

\_\_\_\_\_ In Haskell: \_\_\_\_\_

```
len :: [Int] -> Int
len s = if s == [ ] then
        0
      else
        1 + len (tail s)
```

- We first check if we've reached the end of the list `s == [ ]`. Otherwise we compute the length of the tail of s, and add one to get the length of s itself.

# Reduction View of len [5,6]

```
len s = if s == [ ] then 0 else 1 + len (tail s)
```

```
len [5,6] ⇒
```

```
  if [5,6]==[ ] then 0 else 1 + len (tail [5,6]) ⇒
```

```
  1 + len (tail [5,6]) ⇒
```

```
  1 + len [6] ⇒
```

```
  1 + (if [6]==[ ] then 0 else 1 + len (tail [6])) ⇒
```

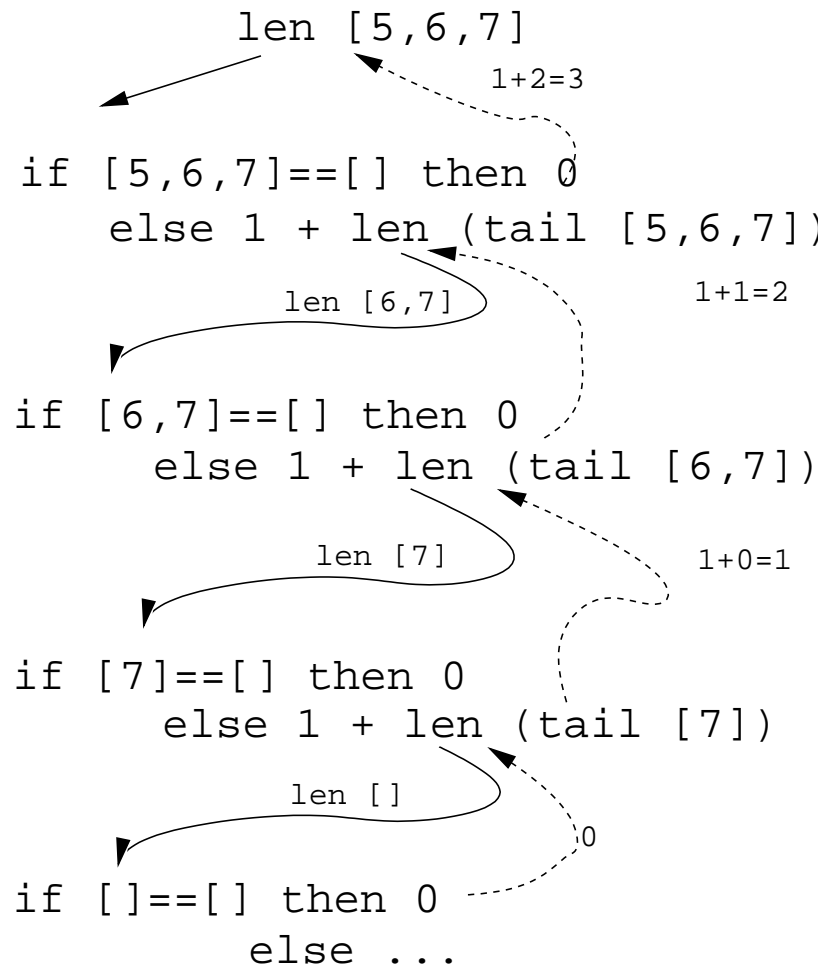
```
  1 + (1 + len (tail [6])) ⇒
```

```
  1 + (1 + len [ ]) ⇒
```

```
  1 + (1 + (if [ ]==[ ] then 0 else 1+len (tail [ ]))) ⇒
```

```
  1 + (1 + 0)) ⇒ 1 + 1 ⇒ 2
```

# Tree View of len [5,6,7]



```
len :: [Int] -> Int
len s = if s==[] then 0
        else 1+len(tail s)
```

- Tree View of len [5,6,7]