1 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.

2 Defining Functions...

- Here’s the ubiquitous factorial function:

  ```haskell
  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:

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

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

  ```haskell
  fact n = if n == 0 then ... ```
3 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
  ```

- `if e_1 then e_2 else e_3` is a conditional expression that returns the value of `e_2` if `e_1` evaluates to `True`. If `e_1` evaluates to `False`, then the value of `e_3` is returned. Examples:

  ```
  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
  ```

4 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 \cdots arg_n
  ```

- Function application examples:

  ```
  fact 1  \Rightarrow 1
  fact 5  \Rightarrow 120
  fact (3+2) \Rightarrow 120
  ```

5 Standard 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 1\hspace{1cm} \Leftarrow \text{guard}
  else n \times \text{fact}(n-1) \hspace{1cm} \Leftarrow \text{general case}
  ```
6 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.

7 Tree View of fact 3

```
    fact 3
       ↓
define fact 3 = if 3==0 then 1
          else 3 * fact (3-1)
    define fact 2 = if 2==0 then 1
                  else 2 * fact (2-1)
    define fact 1 = if 1==0 then 1
                  else 1 * fact (1-1)
    define fact 0 = if 0==0 then 1
                  else ...
```

- 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.

8 Tree View of fact 3

```
    fact 3
       ↓
define fact 3 = if 3==0 then 1
          else 3 * fact (3-1)
    define fact 2 = if 2==0 then 1
                  else 2 * fact (2-1)
    define fact 1 = if 1==0 then 1
                  else 1 * fact (1-1)
    define fact 0 = if 0==0 then 1
                  else ...
```

- When the guard is True we evaluate the base case and return back up through the layers of recursion.
9 Box View of fact 3

10 Box View of fact 3...

11 Box View of fact 3...

12 Reduction View of fact 3

\[
\text{fact 3} \Rightarrow \\
\text{if } 3 == 0 \text{ then } 1 \text{ else } 3 * \text{fact (3-1)} \Rightarrow \\
\text{if False then } 1 \text{ else } 3 * \text{fact (3-1)} \Rightarrow \\
3 * \text{fact (3-1)} \Rightarrow \\
3 * \text{fact 2} \Rightarrow \\
3 * \text{if } 2 == 0 \text{ then } 1 \text{ else } 2 * \text{fact (2-1)} \Rightarrow \\
3 * \text{if False then } 1 \text{ else } 2 * \text{fact (2-1)} \Rightarrow \\
3 * (2 * \text{fact (2-1)}) \Rightarrow \\
3 * (2 * \text{fact 1}) \Rightarrow \\
3 * (2 * \text{if } 1 == 0 \text{ then } 1 \text{ else } 1 * \text{fact (1-1)}) \\
\Rightarrow \ldots
\]
13 Reduction View of fact 3...

\[ 3 \times (2 \times \text{if } 1 == 0 \text{ then } 1 \text{ else } 1 \times \text{fact } (1-1)) \Rightarrow \\
3 \times (2 \times \text{if False then } 1 \text{ else } 1 \times \text{fact } (1-1)) \Rightarrow \\
3 \times (2 \times (1 \times \text{fact } (1-1))) \Rightarrow \\
3 \times (2 \times (1 \times \text{fact } 0)) \Rightarrow \\
3 \times (2 \times (1 \times \text{if } 0 == 0 \text{ then } 1 \text{ else } 0 \times \text{fact } (0-1))) \Rightarrow \\
3 \times (2 \times (1 \times \text{if True then } 1 \text{ else } 0 \times \text{fact } (0-1))) \Rightarrow \\
3 \times (2 \times (1 \times 1)) \Rightarrow \\
3 \times (2 \times 1) \Rightarrow \\
3 \times 2 \Rightarrow \\
6 \]

14 Recursion Over Lists

• In the fact function the guard was \( n==0 \), and the recursive step was \text{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, \( f \text{ tail} \).

15 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:

\[
\text{len} :: [\text{Int}] \rightarrow \text{Int} \\
\text{len } s = \text{if } s == [ ] \text{ then } 0 \\
\text{else} \\
\quad 0 \\
\quad 1 + \text{len } (\text{tail } s)
\]

• 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.

16 Reduction View of len \([5,6]\)

\[
\text{len } s = \text{if } s == [ ] \text{ then } 0 \text{ else } 1 + \text{len } (\text{tail } s) \\
\text{len } [5,6] \Rightarrow \\
\text{if } [5,6]==[ ] \text{ then } 0 \text{ else } 1 + \text{len } (\text{tail } [5,6]) \Rightarrow \\
\quad 1 + \text{len } (\text{tail } [5,6]) \Rightarrow \\
\quad 1 + \text{len } [6] \Rightarrow \\
\quad 1 + (\text{if } [6]==[ ] \text{ then } 0 \text{ else } 1 + \text{len } (\text{tail } [6])) \Rightarrow \\
\quad 1 + (1 + \text{len } (\text{tail } [6])) \Rightarrow \\
5 \]
1 + (1 + \text{len } []) \Rightarrow \\
1 + (1 + (\text{if } []==[] \text{ then } 0 \text{ else } 1+\text{len } (\text{tail } []))) \Rightarrow \\
1 + (1 + 0) \Rightarrow 1 + 1 \Rightarrow 2

17 Tree View of \text{len } [5,6,7]

- Tree View of \text{len } [5,6,7]

\text{len} :: \text{[Int]} \to \text{Int}
\text{len } s = \text{if } s==[] \text{ then } 0 \\
\text{else } 1+\text{len} (\text{tail } s)