Defining Functions

- When programming in a functional language we have basically two techniques to choose from when defining a new function:
  - Recursion
  - 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.

Here’s the ubiquitous factorial function:

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

- The syntax of a type signature is
  
  ```haskell
  fun_name :: argument_types
  fact takes one integer input argument and returns one integer result.
  ```

- The syntax of function declarations:
  ```haskell
  fun_name param_names = fun_body
  ```

- Here’s the ubiquitous factorial function:
  ```haskell
  fact :: Int -> Int
  fact n = if n == 0 then 1 else n * fact (n-1)
  ```
Conditional Expressions

- 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 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** ?:::

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

Example:

```plaintext
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:

```plaintext
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:

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

- Function application examples:

  ```plaintext
  fact 1  \Rightarrow 1
  fact 5  \Rightarrow 120
  fact (3+2) \Rightarrow 120
  ```
Multi-Argument Functions

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

  \[
  \text{add} :: (\text{Int},\text{Int}) \rightarrow \text{Int} \\
  \text{add} (x,y) = x+y
  \]

  \[
  > \text{add} (40,2) \\
  42
  \]

- Later, we'll learn about Curried Functions.

The \textit{error} Function

- \textit{error string} can be used to generate an error message and terminate a computation.

  \[
  \text{error string} \\
  \text{error string} \\
  \text{error string}
  \]

- This is similar to Java's exception mechanism, but a lot less advanced.

  \[
  \text{f :: Int -> Int} \\
  \text{f n = if n<0 then} \\
  \quad \text{error "illegal argument"} \\
  \quad \text{else if n <= 1 then} \\
  \quad \quad 1 \\
  \quad \text{else} \\
  \quad \quad n * f (n-1)
  \]

  \[
  > \text{f (-1)} \\
  \text{Program error: illegal argument}
  \]

Layout

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

  \[
  \text{myfunc :: Int -> Int} \\
  \text{myfunc x = if x == 0 then} \\
  \quad 0 \text{ else 99}
  \]

- 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.

  \[
  \text{f a + b} \\
  \]

- In math (and Java) we use parentheses to include arguments; in Haskell no parentheses are needed.

  \[
  > \text{f a + b} \\
  \text{means} \\
  > (\text{f a}) + b \\
  \text{since function application binds harder than plus.}
  \]
Here’s a comparison between mathematical notations and Haskell:

<table>
<thead>
<tr>
<th>Math</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x) )</td>
<td>( f \ x )</td>
</tr>
<tr>
<td>( f(x, y) )</td>
<td>( f \ x \ y )</td>
</tr>
<tr>
<td>( f(g(x)) )</td>
<td>( f \ (g \ x) )</td>
</tr>
<tr>
<td>( f(x, g(y)) )</td>
<td>( f \ x \ (g \ y) )</td>
</tr>
<tr>
<td>( f(x)g(y) )</td>
<td>( f \ x \ * \ g \ y )</td>
</tr>
</tbody>
</table>

**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).

```haskell
fact n =
  if n == 0 then 1               \( \leftarrow \) guard
  else n * fact (n-1)            \( \leftarrow \) general case
```

**Simulating Recursive Functions**

- We can visualize the evaluation of \( \text{fact} \ 3 \) 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 \texttt{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.
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.

When the guard is True we evaluate the base case and return back up through the layers of recursion.
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, \( \text{head:tail} \).
3. The recursive function application will often be on the tail, \( f \text{tail} \).
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: 

\[
\text{len :: } [\text{Int}] \rightarrow \text{Int}
\]

\[
\text{len } s = \begin{cases} 
0 & \text{if } s == [\ ] \\
\text{else } & 1 + \text{len} (\text{tail } s)
\end{cases}
\]

- 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 \(\text{len } [5,6]\)

\[
\text{len } s = \begin{cases} 
0 & \text{if } s == [\ ] \\
\text{else } & 1 + \text{len} (\text{tail } s)
\end{cases}
\]

\[
\text{len } [5,6] \Rightarrow \\
\begin{align*}
\text{if } [5,6]==[\ ] & \text{ then } 0 \\
\text{else } & 1 + \text{len} (\text{tail } [5,6]) \\
& \Rightarrow \\
& 1 + \text{len} [6] \\
& \Rightarrow \\
& 1 + (\text{if } [6]==[\ ] \text{ then } 0 \text{ else } 1 + \text{len} (\text{tail } [6])) \\
& \Rightarrow \\
& 1 + (1 + \text{len} (\text{tail } [6])) \\
& \Rightarrow \\
& 1 + (1 + \text{len } [6]) \\
& \Rightarrow \\
& 1 + (1 + (\text{if } [\ ]==[\ ] \text{ then } 0 \text{ else } 1 + \text{len} (\text{tail } [\ ]))) \\
& \Rightarrow \\
& 1 + (1 + 0) \\
& \Rightarrow 1 + 1 \\
& \Rightarrow 2
\end{align*}
\]

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