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 ···
  ```

Defining Functions...

- 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
  ```

- **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:

  ```haskell
  if False then 5 else 6  ⇒  6
  if 1==2 then 5 else 6  ⇒  6
  5 + if 1==1 then 3 else 2  ⇒  8
  ```

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.
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 gofer interpreter goes through when evaluating a function. In our notation boxed subexpressions are substituted or evaluated in the next reduction.
- Note that the Gofer interpreter may not go through exactly the same steps as shown in our simulations. More about this later.

Fact 3

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

if 2==0 then 1
  else 2 * fact (2-1)
    fact 1

if 1==0 then 1
  else 1 * fact (1-1)
    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.

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

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  ⩽ guard
  else
    n * fact (n-1)  ⩽ general case
```

• 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).
• When the guard is True we evaluate the base case and return back up through the layers of recursion.
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 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 Gofer:

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

- 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 fact 3

\[
\begin{align*}
fact 3 & \Rightarrow \\
\text{if } 3 == 0 & \text{ then } 1 \text{ else } 3 \ast \text{fact } (3-1) & \Rightarrow \\
\text{if False} & \text{ then } 1 \text{ else } 3 \ast \text{fact } (3-1) & \Rightarrow \\
3 & \ast \text{fact } (3-1) & \Rightarrow \\
3 & \ast \text{fact } 2 & \Rightarrow \\
3 & \ast \text{if } 2 == 0 & \text{ then } 1 \text{ else } 2 \ast \text{fact } (2-1) & \Rightarrow \\
3 & \ast \text{if False} & \text{ then } 1 \text{ else } 2 \ast \text{fact } (2-1) & \Rightarrow \\
3 & \ast (2 \ast \text{fact } (2-1)) & \Rightarrow \\
3 & \ast (2 \ast \text{fact } 1) & \Rightarrow \\
3 & \ast (2 \ast \text{if } 1 == 0 & \text{ then } 1 \text{ else } 1 \ast \text{fact } (1-1)) & \Rightarrow \ldots
\end{align*}
\]

Reduction View of fact 3...

\[
\begin{align*}
3 & \ast (2 \ast \text{if } 1 == 0 & \text{ then } 1 \text{ else } 1 \ast \text{fact } (1-1)) & \Rightarrow \\
3 & \ast (2 \ast \text{if False} & \text{ then } 1 \text{ else } 1 \ast \text{fact } (1-1)) & \Rightarrow \\
3 & \ast (2 \ast (1 \ast \text{fact } (1-1))) & \Rightarrow \\
3 & \ast (2 \ast (1 \ast \text{fact } 0)) & \Rightarrow \\
3 & \ast (2 \ast (1 \ast \text{if } 0 == 0 & \text{ then } 1 \text{ else } 0 \ast \text{fact } (0-1))) & \Rightarrow \\
3 & \ast (2 \ast (1 \ast \text{if True} & \text{ then } 1 \text{ else } 0 \ast \text{fact } (0-1))) & \Rightarrow \\
3 & \ast (2 \ast (1 \ast 1)) & \Rightarrow \\
3 & \ast (2 \ast 1) & \Rightarrow \\
3 & \ast 2 & \Rightarrow \\
6
\end{align*}
\]
Pattern Matching

- Gofer has a notation (called patterns) for defining functions that is more convenient than conditional (if-then-else) expressions.
- Patterns are particularly useful when the function has more than two cases.

Pattern Syntax:

```
function_name pattern_1 = expression_1
function_name pattern_2 = expression_2
...
function_name pattern_n = expression_n
```

Pattern Matching...

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

Fact Revisited:

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

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
```
Pattern Matching...

- When a function $f$ is applied to an argument, Gofer looks at each definition of $f$ until the argument matches one of the patterns.

  not True = False
  not False = True

Pattern Matching...

- Pattern matching allows us to have alternative definitions for a function, depending on the format of the actual parameter. Example:

  isNice "Jenny" = "Definitely"
  isNice "Johanna" = "Maybe"
  isNice "Chris" = "No Way"

Pattern Matching...

- We can use pattern matching as a design aid to help us make sure that we're considering all possible inputs.

  In most cases a function definition will consist of a number of mutually exclusive patterns, followed by a default (or catch-all) pattern:

    diary "Monday" = "Woke up"
    diary "Sunday" = "Slept in"
    diary anyday = "Did something else"

    diary "Sunday" ⇒ "Slept in"
    diary "Tuesday" ⇒ "Did something else"

Pattern Matching...

  - Pattern matching simplifies taking structured function arguments apart. Example:

    fun (x:xs) = x ⊕ fun xs
    ⇔
    fun xs = head xs ⊕ fun (tail xs)
The sumlist Function

Using conditional expr:

```haskell
sumlist :: [Int] -> Int
sumlist xs = if xs == [] then 0
            else head xs + sumlist(tail xs)
```

Using patterns:

```haskell
sumlist :: [Int] -> Int
sumlist [] = 0
sumlist (x:xs) = x + sumlist xs
```

- Note that patterns are checked top-down! The ordering of patterns is therefore important.

Pattern Matching – Integer Patterns

- There are several kinds of integer patterns that can be used in a function definition.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Syntax</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>var_name</td>
<td>fact n = ···</td>
<td>n matches any argument</td>
</tr>
<tr>
<td>constant</td>
<td>literal</td>
<td>fact 0 = ···</td>
<td>matches the value</td>
</tr>
<tr>
<td>wildcard</td>
<td>_</td>
<td>five _ = 5</td>
<td>_ matches any argument</td>
</tr>
<tr>
<td>(n+k) pat.</td>
<td>(n+k)</td>
<td>fact (n+1) = ···</td>
<td>(n+k) matches any integer ≥ k</td>
</tr>
</tbody>
</table>

The length Function Revisited

Using conditional expr:

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

Using patterns:

```haskell
len :: [Int] -> Int
len [] = 0
len (x:xs) = 1 + len xs
```

- Note how similar `len` and `sumlist` are. Many recursive functions on lists will have this structure.

Pattern Matching – List Patterns

- There are also special patterns for matching and (taking apart) lists.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Syntax</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cons</td>
<td>(x:xs)</td>
<td>len (x:xs) = ···</td>
<td>matches non-empty list</td>
</tr>
<tr>
<td>empty</td>
<td>[]</td>
<td>len [] = 0</td>
<td>matches the empty list</td>
</tr>
<tr>
<td>one-elem</td>
<td>[x]</td>
<td>len [x] = 1</td>
<td>matches a list with exactly 1 element.</td>
</tr>
<tr>
<td>two-elem</td>
<td>[x,y]</td>
<td>len [x,y] = 2</td>
<td>matches a list with exactly 2 elements.</td>
</tr>
</tbody>
</table>
The fact Function Revisited

Using conditional expr:

fact n = if n == 0 then 1 else n * fact (n-1)

Using patterns:

fact' :: Int -> Int
fact' 0 = 1
fact' (n+1) = (n+1) * fact' n

• Are fact and fact’ identical?
  fact (-1) ⇒ Stack overflow
  fact' (-1) ⇒ Program Error

• The second pattern in fact’ only matches positive integers (≥ 1).

Summary

• Functional languages use recursion rather than iteration to express repetition.

• We have seen two ways of defining a recursive function: using conditional expressions (if-then-else) or pattern matching.

• A pattern can be used to take lists apart without having to explicitly invoke head and tail.

• Patterns are checked from top to bottom. They should therefore be ordered from specific (at the top) to general (at the bottom).

Homework

• Define a recursive function addints that returns the sum of the integers from 1 up to a given upper limit.

• Simulate the execution of addints 4.

addints :: Int -> Int
addints a = ...

? addints 5
15

? addints 2
3

Homework...

• Define a recursive function member that takes two arguments – an integer x and a list of integers L – and returns True if x is an element in L.

• Simulate the execution of member 3 [1,4,3,2].

member :: Int -> [Int] -> Bool
member x L = ...

? member 1 [1,2,3]
True

? member 4 [1,2,3]
False
Homework...

- Write a recursive function \texttt{memberNum} \( x \) \( L \) which returns the number of times \( x \) occurs in \( L \).
- Use \texttt{memberNum} to write a function \texttt{unique} \( L \) which returns a list of elements from \( L \) that occurs exactly once.

\begin{verbatim}
memberNum :: Int -> [Int] -> Int
unique :: [Int] -> Int

? memberNum 5 [1,5,2,3,5,5]
  3
? unique [2,4,2,1,4]
  1
\end{verbatim}

Homework...

- Ackerman’s function is defined for nonnegative integers:

\[
A(0, n) = n + 1 \\
A(m, 0) = A(m - 1, 1) \\
A(m, n) = A(m - 1, A(m, n - 1))
\]

- Use pattern matching to implement Ackerman’s function.
- Flag all illegal inputs using the built-in function \texttt{error} \( S \) which terminates the program and prints the string \( S \).

\begin{verbatim}
ackerman :: Int -> Int -> Int
ackerman 0 5 ⇒ 6
ackerman (-1) 5 ⇒ ERROR
\end{verbatim}