University of Auckland

415.360

Functional Programming

Christian Collberg
March 1, 1996

Defining Functions

Defining Functions I

- 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 II

- 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 III

- 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`

- $\boxed{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3}$ 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      ⇒ 6
if 1==2 then 5 else 6       ⇒ 6
5 + if 1==1 then 3 else 2   ⇒ 8
```

Defining Functions IV

- `fact` is defined recursively, i.e. the function body contains an application of the function itself.
- The syntax of function application is: $\boxed{\texttt{fun\_name arg}}$. 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
```

Slide 4–4

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

Slide 4–5

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.

Slide 4–6

Tree View of `fact 3` I

```
      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 ...
```

- We keep going deeper into the recursion (evaluating the **general case**) until the **guard** is evaluated to `True`.

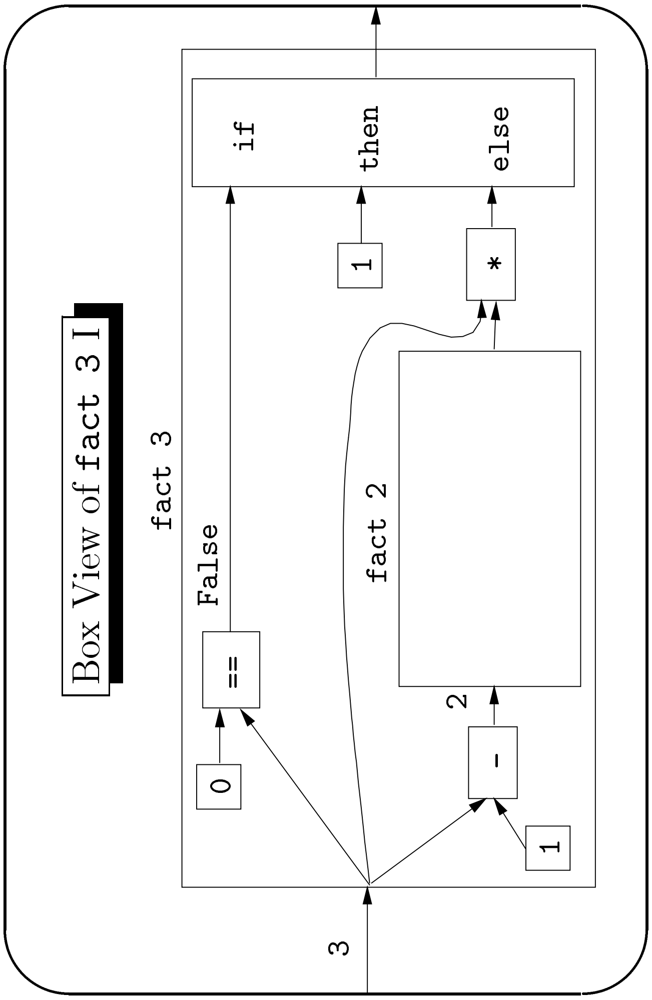
Slide 4–7

Tree View of `fact 3` II

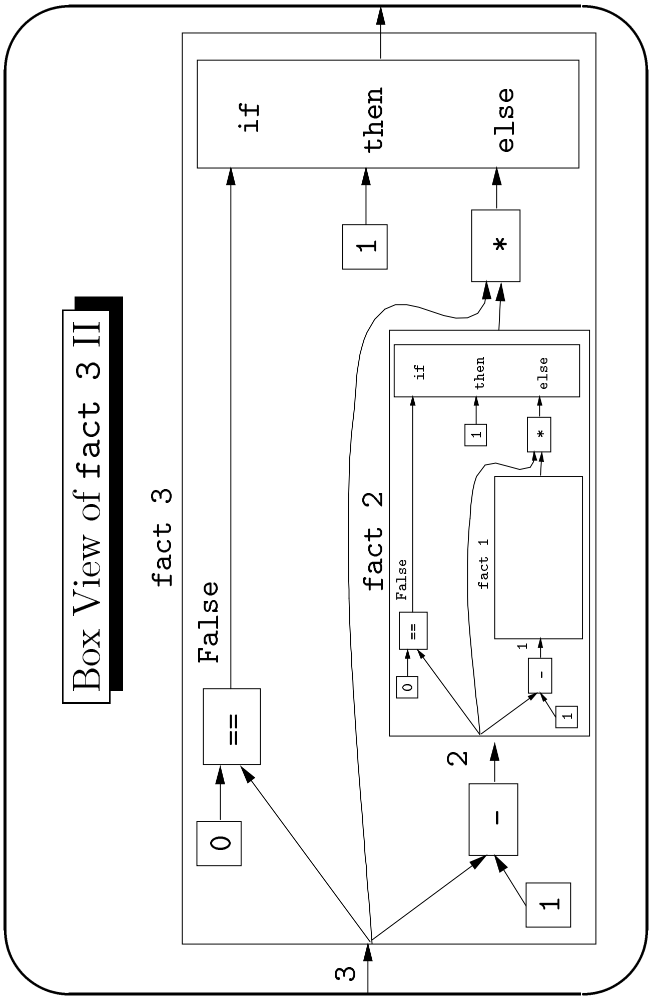
```
        fact 3
                            3*2=6

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

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

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

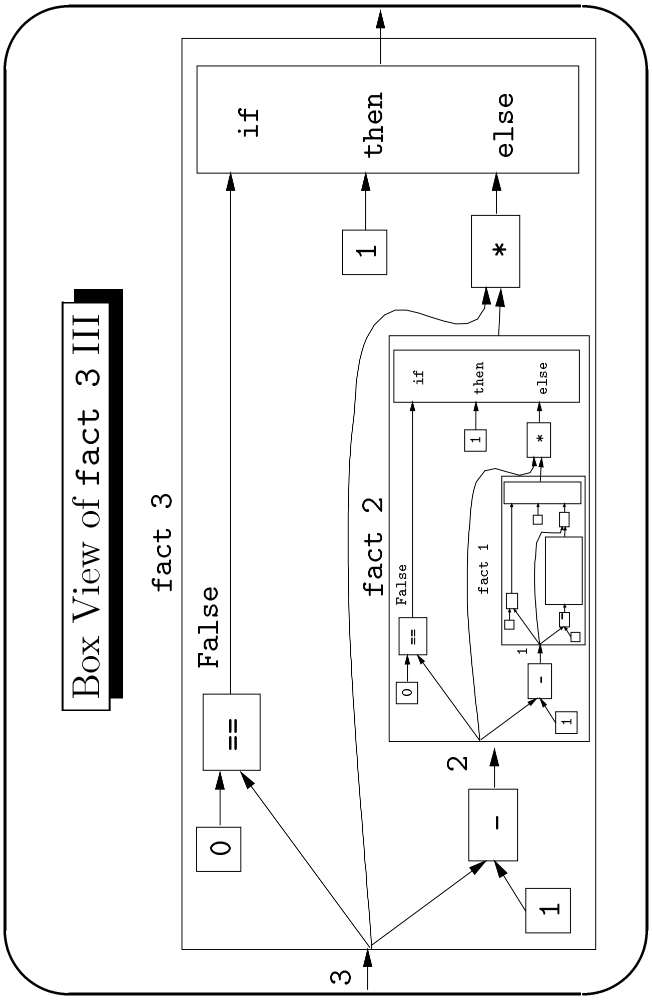
Slide 4–8



Slide 4–9



Slide 4–10



Slide 4–11

Reduction View of fact 3 I

```
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))  ⇑ ...
```

Slide 4–12

Reduction View of fact 3 II

```
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
```

Slide 4–13

Recursion Over Lists I

- 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`.

Slide 4–14

The `length` Function I

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:

```
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.

Slide 4–15

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
```

Slide 4–16

Tree View of len [5,6,7]

```
      len [5,6,7]
                        1+2=3
if [5,6,7]==[] then 0
       else 1 + len (tail [5,6,7])
          len [6,7]                   1+1=2
if [6,7]==[] then 0
       else 1 + len (tail [6,7])
          len [7]                     1+0=1
if [7]==[] then 0
       else 1 + len (tail [7])
          len []
                              0
if []==[] then 0
       else ...

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

Slide 4–17

Pattern Matching I

- 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
```

`fact` Revisited:

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

Slide 4–18

Pattern Matching II

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

```
isNice "Jenny" = "Definitely"
isNice "Jessica" = "Maybe"
isNice "Chris" = "No Way"
```

- We can use pattern matching as a design aid to help us make sure that we're considering all possible inputs.
- Pattern matching simplifies taking structured function arguments apart. Example:

```
fun (x:xs) = x ⊕ fun xs
      ⇔
fun xs = head xs ⊕ fun (tail xs)
```

Slide 4–19

Pattern Matching III

- 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
```

- 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 s···g else"
```

Slide 4–20

Pattern Matching IV – Integer Patterns

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

Pattern	Syntax	Example	Description
variable	var_name	fact n = ···	n matches any argument
constant	literal	fact 0 = ···	matches the value
wildcard	_	five _ = 5	_ matches any argument
(n+k) pat.	(n+k)	fact (n+1) = ···	(n+k) matches any integer $\geq k$

Slide 4–21

Pattern Matching V – List Patterns

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

Pattern	Syntax	Example	Description
cons	(x:xs)	len (x:xs) = ···	matches non-empty list
empty	[ ]	len [ ] = 0	matches the empty list
one-elem	[x]	len [x] = 1	matches a list with exactly 1 element.
two-elem	[x,y]	len [x,y] = 2	matches a list with exactly 2 elements.

Slide 4–22

The `sumlist` Function

Using conditional expr:

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

Using patterns:

```
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.

Slide 4–23

The `length` Function Revisited

Using conditional expr:

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

Using patterns:

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

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

Slide 4–24

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)` $\Rightarrow$ `Stack overflow`

  `fact' (-1)` $\Rightarrow$ `Program Error`

- The second pattern in `fact'` only matches positive integers (≥ 1). Hence there is no pattern that matches `-1`, and `fact'` fails for any negative argument.

Slide 4–25

Summary

- B&W reading list:
  1. Defining Functions: 8–11
  2. Pattern matching: 36–37,72–74
  3. Recursion: 104–105, 109–110. (Ignore induction for now.)
- 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).

Slide 4–26

Homework I

- 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`.

Template:

```
addints ::  Int -> Int
addints a = ···
```

Examples:

```
?  addints 5
   15

?  addints 2
   3
```

Slide 4–27

Homework II

- 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]`.

Template:

```
member ::  Int -> [Int] -> Bool
member x L = ···
```

Examples:

```
?  member 1 [1,2,3]
   True
?  member 3 [1,2,3]
   True
?  member 4 [1,2,3]
   False
```

Slide 4–28

Homework III

- Write a recursive function `memberNum x L` which returns the number of times `x` occurs in `L`.
- Use `memberNum` to write a function `unique L` which returns a list of elements from `L` that occurs exactly once.

Template:

```
memberNum ::  Int -> [Int] -> Int
unique ::  [Int] -> Int
```

Examples:

```
?  memberNum 5 [1,5,2,3,5,5]
   3
?  unique [2,4,2,1,4]
   1
```

Slide 4–29

Homework IV

- Ackerman's function is defined for nonnegative integers:

$$
\begin{aligned}
A(0,n) &= n+1 \\
A(m,0) &= A(m-1,1) \\
A(m,n) &= A(m-1,A(m,n-1))
\end{aligned}
$$

- Use pattern matching to implement Ackerman's function.
- Flag all illegal inputs using the built-in function `error S` which terminates the program and prints the string `S`.

Template & Examples:

```
ackerman ::  Int -> Int -> Int
ackerman 0 5 ⇒ 6
ackerman (-1) 5 ⇒ ERROR
```

Slide 4–30