# CSc 372

## Comparative Programming Languages

## 7 : Haskell — Patterns

Department of Computer Science
University of Arizona

collberg@gmail.com



Christian Collberg

## Pattern Matching

- Haskell 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)
```

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

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

## Pattern Matching. . .

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

```
not True = False
not False = True
```

## Pattern Matching. . .

- 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 – 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 $= \cdots$ | n matches any argument |
| constant | literal | fact 0 $= \cdots$ | matches the value |
| wildcard | _ | five _ $= 5$ | _ matches any argument |
| (n+k) pat. | (n+k) | fact (n+1) $= \cdots$ | (n+k) matches any integer $\geq k$ |

## Pattern Matching – 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. |

## 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.

## 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.

## 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 ($\geq 1$).

## 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 xs = ···

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

## Homework

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

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

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

# Homework

- 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`.

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