CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

CS4450/7450
Chapter 4: Syntax in Functions
Principles of Programming Languages

Dr. William Harrison

University of Missouri

September 13, 2016

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

What is a Pattern?

```
data I = A | B | C
foo :: I -> String
foo A = "One"
foo B = "Two"
foo C = "Three"
```

A *pattern* is anything in the argument position of a function definition.

What is a Pattern?

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

```
data I = A | B | C
foo :: I -> String
foo A = "One"
foo B = "Two"
foo C = "Three"
```

A *pattern* is anything in the argument position of a function definition. There are:

- variable patterns, wildcard patterns, constructor patterns, as-patterns

...and bigger patterns are composed of smaller patterns.

Wildcard Patterns

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

The underscore "_" is a wildcard pattern. They match anything.

```
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

Wildcards are good to use to indicate that you don't care about the value it matches.

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

Variable Patterns

Variable patterns match anything:

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)
```

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

Variable Patterns

Variable patterns match anything:

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)
```

In the following application, a and b are bound to (5,6) and (7,8), respectively.

```
addVectors (5,6) (7,8)
```

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

Variable Patterns

Variable patterns match anything:

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)
```

In the following application, a and b are bound to (5,6) and (7,8), respectively.

```
addVectors (5,6) (7,8)
```

Can also express structure of the input directly using patterns:

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

Constructor Patterns

Recall that lists have two constructors:

```
data [a] = [] | (a : [a])
```

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

Constructor Patterns

Recall that lists have two constructors:

```
data [a] = [] | (a : [a])
```

Constructors, when appearing in argument position, are patterns:

```
length :: (Num b) => [a] -> b
length []     = 0
length (_:xs) = 1 + length xs
```

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

Composite Patterns

Patterns can be composed to make bigger patterns, thereby
giving you more expressiveness in matching values:

```
tell :: (Show a) => [a] -> String
tell []       = "The list is empty"
tell (x:[])   = "The list has one element: " ++ show
    x
tell (x:y:[]) = "The list has two elements: " ++ show
    x ++ " and " ++ show y
tell (x:y:_)  = "This list is long. The first two
    elements are: " ++ show x ++ " and " ++ show y
```

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

As Patterns

Here, "as" is @

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of "
     ++ all ++ " is " ++ [x]
```

```
ghci> capital "Dracula"
"The first letter of Dracula is D"
```

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

Guards

It's easy enough to write the maximum function using
`if-then-else`:

```
max :: Float -> Float -> Float
max a b = if a<b then b else a
```

It's easy enough to write the maximum function using
`if-then-else`:

```
max :: Float -> Float -> Float
max a b = if a<b then b else a
```

Another way to define the identical function is with *guards*:

```
max :: Float -> Float -> Float
max a b | a<b      = b
        | otherwise = a
```

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

Why use guards: Readability.

This is much more readable:

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi <= 18.5 = "underweight"
    | bmi <= 25.0 = "normal"
    | bmi <= 30.0 = "overweight"
    | otherwise   = "obese"
```

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

Why use guards: Readability.

This is much more readable:

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi <= 18.5 = "underweight"
    | bmi <= 25.0 = "normal"
    | bmi <= 30.0 = "overweight"
    | otherwise   = "obese"
```

...than this:

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi = if bmi <= 18.5
                then "underweight"
                  else if bmi <= 25.0
                        then "normal"
                  else if bmi <= 30.0
                        then "overweight"
                  else
                        "obese"
```

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

Where clauses
let you define local variables

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "underweight"
    | weight / height ^ 2 <= 25.0 = "normal"
    | weight / height ^ 2 <= 30.0 = "overweight"
    | otherwise                   = "obese"
```

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

Where clauses
let you define local variables

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "underweight"
    | weight / height ^ 2 <= 25.0 = "normal"
    | weight / height ^ 2 <= 30.0 = "overweight"
    | otherwise                   = "obese"

bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= 18.5 = "underweight"
    | bmi <= 25.0 = "normal"
    | bmi <= 30.0 = "overweight"
    | otherwise   = "obese"
  where
    bmi = weight / height ^ 2
      -- calculate bmi once , use value repeatedly
```

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

Let definitions
…are just like where clauses

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let
        sideArea = 2 * pi * r * h
        topArea  = pi * r ^2
    in
        sideArea + 2 * topArea
```

- Variables defined in a `let` or `where` clauses are local
- E.g., `sideArea` and `topArea` can be used only in the body of the `let`/`where`.

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

Case Expressions

General form of a case expression:

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

Case Expressions

General form of a case expression:

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...


head :: [a] -> a
head []    = error "empty list"
head (x:_) = x
```

CS4450

Bill Harrison

Pattern
Matching

Guards in
Patterns

Where Clauses

Let Bindings

Case
Expressions

Case Expressions

General form of a case expression:

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```

```
head :: [a] -> a
head []    = error "empty list"
head (x:_) = x
```

A way to define the identical function:

```
head :: [a] -> a
head xs = case xs of
               []    -> error "empty list"
               (x:_) -> x
```