Local Declarations, Overloading

Johannes Borgström
johannes.borgstrom@it.uu.se

Program Design and Data Structures

Based on notes by Tjark Weber, Lars-Henrik Eriksson, Pierre Flener, Sven-Olof Nyström

Today

- Local Declarations, Scope
- Overloading
- Revision

Local Declarations, Scope

Local Declarations

It is possible to bind identifiers locally with an expression of the form

Local Declarations

It is possible to bind identifiers locally with an expression of the form

```
let
  declaration1
  declaration2
  ...
in
  expression
```

Local Declarations

It is possible to bind identifiers locally with an expression of the form

```
let
  declaration1
  declaration2
  ...
in
  expression
```

The identifiers declared between `let` and `in` are bound only until the end of the following expression.

Local Declarations: Example

```
Prelude> let x = 42 in x + 1
43

Prelude> x

<interactive>:...: Not in scope: `x'
```

Scope

The **scope** of a binding is the part of the program where the bound name can be used to refer to the corresponding value. We say that within its scope, the binding is **visible**.

Scope

The **scope** of a binding is the part of the program where the bound name can be used to refer to the corresponding value. We say that within its scope, the binding is **visible**.

Different programming languages have different scoping rules.

Scope

The **scope** of a binding is the part of the program where the bound name can be used to refer to the corresponding value. We say that within its scope, the binding is **visible**.

Different programming languages have different scoping rules.

In Haskell, the scope of a top-level declaration is typically the entire program (i.e., a name that is declared at the top level may be used anywhere in the program code). The scope of a local declaration only extends to the end of the corresponding expression.

Shadowing

Local declarations **shadow** any other declaration of the same name, i.e., they render it inaccessible (within the scope of the local declaration):

Shadowing

Local declarations **shadow** any other declaration of the same name, i.e., they render it inaccessible (within the scope of the local declaration):

```
x = 0
y = let x = 1 in x + 10
```

Shadowing

Local declarations **shadow** any other declaration of the same name, i.e., they render it inaccessible (within the scope of the local declaration):

```
x = 0
y = let x = 1 in x + 10
```

```
Prelude> (x, y)
(0,11)
```

Shadowing: Some Remarks

Shadowing can be confusing. It is usually *not* done on purpose.

Shadowing: Some Remarks

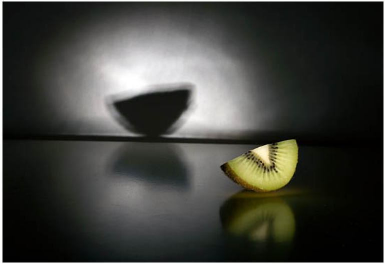
Shadowing can be confusing. It is usually *not* done on purpose.

- In principle, shadowing could be avoided by renaming so that all declarations use distinct names.

Shadowing: Some Remarks

Shadowing can be confusing. It is usually *not* done on purpose.

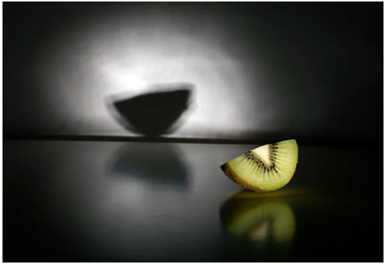
- In principle, shadowing could be avoided by renaming so that all declarations use distinct names.
- However, in realistic programs shadowing typically does happen in some places (more or less by accident).



Shadowing: Some Remarks

Shadowing can be confusing. It is usually *not* done on purpose.

- In principle, shadowing could be avoided by renaming so that all declarations use distinct names.
- However, in realistic programs shadowing typically does happen in some places (more or less by accident).



You need to know about shadowing because you should be able to read and understand realistic programs. Still, it is usually a good idea to avoid shadowing in your own code (by using distinct names).

Formal Arguments Bind Into the Function Body

The scope of a formal argument is the body of the function. Within the body, the argument shadows any earlier declaration of the same name.

Formal Arguments Bind Into the Function Body

The scope of a formal argument is the body of the function. Within the body, the argument shadows any earlier declaration of the same name.

```
x = "shadowed"

add1 x = x + 1

Prelude> x
"shadowed"
Prelude> add1 42
43
```

Formal Arguments can be Shadowed

Bindings within a function, e.g., local declarations, shadow formal arguments of the same name.

Formal Arguments can be Shadowed

Bindings within a function, e.g., local declarations, shadow formal arguments of the same name.

```
shadowedArgument x = let x=0 in x+1
```

```
Prelude> shadowedArgument 42
1
```

What's in a Name?

What's in a name? That which we call a rose
By any other name would smell as sweet.

W. Shakespeare, Romeo and Juliet (1597)

What's in a Name?

What's in a name? That which we call a rose
By any other name would smell as sweet.

W. Shakespeare, Romeo and Juliet (1597)

In most programming languages (including Haskell), the semantics of programs does not depend on the specific names that you choose.

What's in a Name?

What's in a name? That which we call a rose
By any other name would smell as sweet.

W. Shakespeare, Romeo and Juliet (1597)

In most programming languages (including Haskell), the semantics of programs does not depend on the specific names that you choose.

Names are only interpreted by human readers of the program. They serve to **explain** the purpose of values, functions, etc.

What's in a Name?

What's in a name? That which we call a rose
By any other name would smell as sweet.

W. Shakespeare, Romeo and Juliet (1597)

In most programming languages (including Haskell), the semantics of programs does not depend on the specific names that you choose.

Names are only interpreted by human readers of the program. They serve to **explain** the purpose of values, functions, etc.

Therefore, it is important to use **descriptive** names.

Renaming

It is possible to replace the names in a program (in a systematic manner) without changing the semantics of the program at all.

Renaming

It is possible to replace the names in a program (in a systematic manner) without changing the semantics of the program at all.

```
x = 0
y = let x = 1 in x + 10

Prelude> (x, y)
(0,11)
```

Renaming

It is possible to replace the names in a program (in a systematic manner) without changing the semantics of the program at all.

```
x = 0
y = let x = 1 in x + 10

Prelude> (x, y)
(0,11)
```

Renaming

It is possible to replace the names in a program (in a systematic manner) without changing the semantics of the program at all.

```
x = 0
y = let x = 1 in x + 10

Prelude> (x, y)
(0,11)
```

```
z = 0
y = let x = 1 in x + 10

Prelude> (z, y)
(0,11)
```

Renaming

It is possible to replace the names in a program (in a systematic manner) without changing the semantics of the program at all.

```
x = 0
y = let x = 1 in x + 10

Prelude> (x, y)
(0,11)
```

```
z = 0
y = let x = 1 in x + 10

Prelude> (z, y)
(0,11)
```

(Of course, names that are declared outside the program, such as +, cannot be replaced in your code only.)

Renaming is Cheating

Knowingly submitting a (changed) copy of some fellow student's solution is cheating.

Renaming is Cheating

Knowingly submitting a (changed) copy of some fellow student's solution is cheating.

We use software that can detect changed copies—even if you rename all identifiers!

Renaming is Cheating

Knowingly submitting a (changed) copy of some fellow student's solution is cheating.

We use software that can detect changed copies—even if you rename all identifiers!

Don't rename to cheat. (You won't cheat your way to a successful career anyway. Better try to pick up some effective study techniques.)

Overloading

Overloading

Many functions in Haskell (e.g., `+`, `-`, `*`, `/`, `<`, `>`, `==`, ...) are available for different types.

Overloading

Many functions in Haskell (e.g., +, −, ∗, /, <, >, ==, . . .) are available for different types.

```
Prelude> 1 < 2
True
Prelude> 2.72 < 3.14
True
Prelude> "foo" < "bar"
False
```

Overloading

Many functions in Haskell (e.g., +, −, ∗, /, <, >, ==, . . .) are available for different types.

```
Prelude> 1 < 2
True
Prelude> 2.72 < 3.14
True
Prelude> "foo" < "bar"
False
```

These are really *three different* functions (that just happen to have the same name, <, for convenience).

Overloading

Many functions in Haskell (e.g., +, −, ∗, /, <, >, ==, . . .) are available for different types.

```
Prelude> 1 < 2
True
Prelude> 2.72 < 3.14
True
Prelude> "foo" < "bar"
False
```

These are really *three different* functions (that just happen to have the same name, <, for convenience).

We say that +, −, ∗, /, <, >, ==, . . . are **overloaded**.

Overloaded Values

Even values from base types can be overloaded.

Overloaded Values

Even values from base types can be overloaded.

For instance, literals `0`, `1`, ... can represent fixed (`Int`) or arbitrary precision (`Integer`) integers.

Overloaded Values

Even values from base types can be overloaded.

For instance, literals `0`, `1`, ... can represent fixed (`Int`) or arbitrary precision (`Integer`) integers.

Another example:

```
Prelude> maxBound :: Int
9223372036854775807
Prelude> maxBound :: Char
'\1114111'
```

Overloading: Type Inference

Sometimes, it is clear from the context (i.e., from the surrounding program code) what the concrete type of an overloaded function (or other value) is.

Overloading: Type Inference

Sometimes, it is clear from the context (i.e., from the surrounding program code) what the concrete type of an overloaded function (or other value) is.

For instance:

```
Prelude> "foo" < "bar"
False
```

Here, < has type `String->String->Bool`.

Overloading: Type Inference

Sometimes, it is clear from the context (i.e., from the surrounding program code) what the concrete type of an overloaded function (or other value) is.

For instance:

```
Prelude> "foo" < "bar"
False
```

Here, < has type `String->String->Bool`.

```
Prelude> Data.Char.toUpper maxBound
'\1114111'
```

Here, `maxBound` has type `Char`.

Using Overloaded Functions

You can use overloaded functions to implement your own functions:

Using Overloaded Functions

You can use overloaded functions to implement your own functions:

```
sumOfSquares x = x*x + x*x
```

Using Overloaded Functions

You can use overloaded functions to implement your own functions:

```
sumOfSquares x = x*x + x*x
```

Since + and $*$ are available for different types (e.g., `Integer`, `Double`, ...), also `sumOfSquares` is available for these types:

Using Overloaded Functions

You can use overloaded functions to implement your own functions:

```
sumOfSquares x = x*x + x*x
```

Since + and $*$ are available for different types (e.g., `Integer`, `Double`, ...), also `sumOfSquares` is available for these types:

```
Prelude> sumOfSquares 3
18
Prelude> sumOfSquares 3.0
18.0
```

Some Standard Type Classes

Overloaded values live in **type classes**. A type class is a collection of types that support the same operation(s).

Some Standard Type Classes

Overloaded values live in **type classes**. A type class is a collection of types that support the same operation(s).

- `Eq` — types having equality/inequality operator
- `Ord` — types whose values can be ordered
- `Show` — types whose values can be converted **to** strings
- `Read` — types whose values can be converted **from** strings
- `Bounded` — types with both an upper and a lower bound.
- `Num` — numbers

The Eq typeclass

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
```

The Eq typeclass

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
```

- Allows checking values for equality.

The Eq typeclass

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
```

- Allows checking values for equality.
- Defined for base types, lists, tuples,...

The Eq typeclass

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
```

- Allows checking values for equality.
- Defined for base types, lists, tuples,...
- Used in pattern matching (value patterns).

The Eq typeclass

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
```

- Allows checking values for equality.
- Defined for base types, lists, tuples,...
- Used in pattern matching (value patterns).
- Using == signals that this type class is to be used—that is, that the type of its arguments must belong to class `Eq`

The Ord typeclass

```haskell
class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a
```

The Ord typeclass

```
class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a
```

- Allows to order values.

The Ord typeclass

```
class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a
```

- Allows to order values.
- Defined for base types, lists, tuples,...

The Ord typeclass

```
class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a
```

- Allows to order values.
- Defined for base types, lists, tuples,...
- Used for sorting values.

The Ord typeclass

```
class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a
```

- Allows to order values.
- Defined for base types, lists, tuples,...
- Used for sorting values.
- Assumes that values can be checked for equality: `(Eq a) => Ord a`.
    - `Ord` is a *subclass* of `Eq`.
    - `Eq` is a *superclass* of `Ord`.

The Show and Read typeclasses

Type class `Show` is for converting values to strings, for printing/displaying, sending across network, storing in files, ...

```
class Show a where
  show :: a -> String
```

The Show and Read typeclasses

Type class `Show` is for converting values to strings, for printing/displaying, sending across network, storing in files, ...

```
class Show a where
  show :: a -> String
```

Type class `Read` is for reading values from strings, for user input, sending across network, storing in files, ...

```
class Read a where
  read :: String -> a
```

The Bounded typeclass

```
class Bounded a where
  minBound, maxBound :: a
```

The Bounded typeclass

```
class Bounded a where
  minBound, maxBound :: a
```

- Examples: `Bool`, `Int`, `Char`, and tuples of bounded types.

The Num Typeclass

```
class (Eq a, Show a) => Num a where
  (+),(-),(*) :: a -> a -> a
  negate       :: a -> a
  abs, signum  :: a -> a
  fromInteger  :: Integer -> a
```

The Num Typeclass

```
class (Eq a, Show a) => Num a where
  (+),(-),(*) :: a -> a -> a
  negate       :: a -> a
  abs, signum  :: a -> a
  fromInteger  :: Integer -> a
```

- Standard arithmetic operations (except division).

The Num Typeclass

```
class (Eq a, Show a) => Num a where
  (+),(-),(*) :: a -> a -> a
  negate      :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a
```

- Standard arithmetic operations (except division).
- Examples: `Int`, `Integer`, `Double`, `Rational`, `Complex`, ...

The Num Typeclass

```
class (Eq a, Show a) => Num a where
  (+),(-),(*) :: a -> a -> a
  negate       :: a -> a
  abs, signum  :: a -> a
  fromInteger  :: Integer -> a
```

- Standard arithmetic operations (except division).
- Examples: `Int`, `Integer`, `Double`, `Rational`, `Complex`, ...
- Can always be compared for equality (`Eq`) and printed (`Show`).

The Num Typeclass

```
class (Eq a, Show a) => Num a where
  (+),(-),(*) :: a -> a -> a
  negate       :: a -> a
  abs, signum  :: a -> a
  fromInteger  :: Integer -> a
```

- Standard arithmetic operations (except division).
- Examples: `Int`, `Integer`, `Double`, `Rational`, `Complex`, ...
- Can always be compared for equality (`Eq`) and printed (`Show`).
- Are not always ordered! (Why?)

The Num Typeclass

```
class (Eq a, Show a) => Num a where
  (+),(-),(*) :: a -> a -> a
  negate       :: a -> a
  abs, signum  :: a -> a
  fromInteger  :: Integer -> a
```

- Standard arithmetic operations (except division).
- Examples: `Int`, `Integer`, `Double`, `Rational`, `Complex`, ...
- Can always be compared for equality (`Eq`) and printed (`Show`).
- Are not always ordered! (Why?)
- Has subclasses for standard division (/) and integral division (`mod`,`div`).

Reading Types with Class

```haskell
zero :: (Eq a, Num p) => a -> p
zero x = case x of y | y == x -> 0
```

Reading Types with Class

```
zero :: (Eq a, Num p) => a -> p
zero x = case x of y | y == x -> 0
```

The type of `zero` can be read as:

- for any concrete types `a` and `p`,

Reading Types with Class

```
zero :: (Eq a, Num p) => a -> p
zero x = case x of y | y == x -> 0
```

The type of `zero` can be read as:

- for any concrete types `a` and `p`,
- if `a` belongs to the typeclass `Eq` and `p` belongs to the typeclass `Num`,

Reading Types with Class

```
zero :: (Eq a, Num p) => a -> p
zero x = case x of y | y == x -> 0
```

The type of `zero` can be read as:

- for any concrete types `a` and `p`,
- if `a` belongs to the typeclass `Eq` and `p` belongs to the typeclass `Num`,
- then `zero` takes values of type `a` to values of type `p`.

Reading Types with Class

```
zero :: (Eq a, Num p) => a -> p
zero x = case x of y | y == x -> 0
```

The type of `zero` can be read as:

- for any concrete types `a` and `p`,
- if `a` belongs to the typeclass `Eq` and `p` belongs to the typeclass `Num`,
- then `zero` takes values of type `a` to values of type `p`.

Reading Types with Class

```
zero :: (Eq a, Num p) => a -> p
zero x = case x of y | y == x -> 0
```

The type of `zero` can be read as:

- for any concrete types `a` and `p`,
- if `a` belongs to the typeclass `Eq` and `p` belongs to the typeclass `Num`,
- then `zero` takes values of type `a` to values of type `p`.

Compare the logical formula

$$\forall a. \forall p. \mathrm{Eq}(a) \wedge \mathrm{Num}(p) \implies (a \implies p)$$

Revision

Basic Types in Haskell

- Integers (`Integer`): `0, 1, 2, -10, 42, ...`
- Floating point numbers (`Double`): `0.0, -10.4, 3.14159, 1.234e6, ...`
- Characters (`Char`): `'a', 'B', '1', ...`
- Strings (`String`): `"Hello", "PKD", "1", ...`
- Booleans (`Bool`): `True`, `False`
- Unit (`()`): has only one value, written `()`
- ...

Note that `1`, `1.0`, `'1'` and `"1"` are all different.

Numeric Operations

For numeric types (`Num a`):

Functions	Type	Semantics
`+`, `-`, `*`	`a->a->a`	arithmetic operations
`==`, `/=`	`a->a->Bool`	equality, inequality
`<`, `<=`, `>`, `>=`	`a->a->Bool`	comparison
`-`, `negate`	`a->a`	negation (e.g., `-(2-3)` $\longrightarrow$ 1)
`abs`	`a->a`	absolute value (e.g., `abs (-1)` $\longrightarrow$ 1)

Numeric Operations

For numeric types (`Num a`):

Functions	Type	Semantics
`+`, `-`, `*`	`a->a->a`	arithmetic operations
`==`, `/=`	`a->a->Bool`	equality, inequality
`<`, `<=`, `>`, `>=`	`a->a->Bool`	comparison
`-`, `negate`	`a->a`	negation (e.g., `-(2-3)` $\longrightarrow$ 1)
`abs`	`a->a`	absolute value (e.g., `abs (-1)` $\longrightarrow$ 1)

For integral types (`Integral a`):

Functions	Type	Semantics
`div`, `mod`	`a->a->a`	modulo (e.g., `mod 7 3` $\longrightarrow$ 1)

String Operations

Functions	Type[1]	Semantics
`++`	`String->String->String`	concatenation
`length`	`String->Int`	length
`head`	`String->Char`	first character
`last`	`String->Char`	last character
`tail`	`String->String`	the string without its first char
`init`	`String->String`	the string without its last char
`take`	`Int->String->String`	the string's first n characters
`drop`	`Int->String->String`	the string without its first n chars
`!!`	`String->Int->Char`	the string's n^{th} character

[1] These operations actually work for lists of any type, e.g. `[Int]`.

Conversions

Function	Type
`fromInteger`	`(Num a) => Integer -> a`
`toInteger`	`(Integral a) => a -> Integer`
`round`	`(RealFrac a, Integral b) => a -> b`
`truncate`	`(RealFrac a, Integral b) => a -> b`
`floor`	`(RealFrac a, Integral b) => a -> b`
`ceiling`	`(RealFrac a, Integral b) => a -> b`
`toEnum`	`(Enum a) => Int -> a`
`fromEnum`	`(Enum a) => a -> Int`
`show`	`(Show a) => a -> String`
`read`	`(Read a) => String -> a`

Operator Precedence and Fixity

How does Haskell know that `1 + 2 * 3` should be evaluated as `1 + (2 * 3)` rather than `(1 + 2)* 3`?

Every infix operator has a **precedence**. Higher precedences bind more tightly.

Operator Precedence and Fixity

How does Haskell know that `1 + 2 * 3` should be evaluated as `1 + (2 * 3)` rather than `(1 + 2)* 3`?

Every infix operator has a **precedence**. Higher precedences bind more tightly.

How does Haskell know that `3 - 2 - 1` should be evaluated as `(3 - 2)- 1` rather than `3 - (2 - 1)`?

Infix operators may be **left**- or **right-associative**, meaning operations are grouped from the left (or right, respectively).

Comparison operators are not associative (what should `x == y >= z` mean?).

Operator Precedence and Fixity

Prec	Left assoc	Non-assoc	Right assoc
9	!!		.
8			^, ^^, **
7	*, /, `div`, `mod`, `rem`, `quot`		
6	+, -		
5			:, ++
4		==, /=, <, <=, >, >=	
3			&&
2			||
1	>>, >>=		
0			$, $!, `seq`

Pairs and Tuples

```
("PKD", 2019) :: (String, Integer)

(1, 2, 3) :: (Integer, Integer, Integer)

(3, "three", 3.0) :: (Integer, String, Double)
```

Pairs and Tuples

```
("PKD", 2019) :: (String, Integer)

(1, 2, 3) :: (Integer, Integer, Integer)

(3, "three", 3.0) :: (Integer, String, Double)
```

The function `(,)` takes two arguments and returns the pair (2-tuple) that consists of these arguments (in the given order).

If-Then-Else, Declarations

```
if condition then trueValue else falseValue
```

If-Then-Else, Declarations

```
if condition then trueValue else falseValue
```

Value declarations associate a value to an identifier.

```
x = 1
myPi = 3.14159
timesPi x = x * myPi

Prelude> x + x
2
Prelude> timesPi 2
6.28318
```

Functions

Function declarations consist of the name, arguments, and body.

```
takeLast n str = drop (length str - n) str
```

Haskell uses a sophisticated algorithm to infer the type of functions.

Functions

Function declarations consist of the name, arguments, and body.

```
takeLast n str = drop (length str - n) str
```

Haskell uses a sophisticated algorithm to infer the type of functions.

Functions can also be anonymous.

```
Prelude> (\n s -> drop (length s - n) s) 5 "computer"
"puter"
```

Functions

Function declarations consist of the name, arguments, and body.

```
takeLast n str = drop (length str - n) str
```

Haskell uses a sophisticated algorithm to infer the type of functions.

Functions can also be anonymous.

```
Prelude> (\n s -> drop (length s - n) s) 5 "computer"
"puter"
```

Operators are just functions written between their arguments. We can convert between operators and functions using `( )` and `` `f` ``.

```
Prelude> (+) 3 5
8
Prelude> 3 `takeLast` "PKD2019"
"019"
```

Function Specifications

In this course, our function specifications will consist of

1. Function name and arguments
2. Function type — argument(s) and result
3. Description — what is the purpose of this function?
4. What the function assumes (precondition)
5. What the function computes (return value)
6. Side effects
7. Examples of function usage

Function Specifications

In this course, our function specifications will consist of

1. Function name and arguments
2. Function type — argument(s) and result
3. Description — what is the purpose of this function?
4. What the function assumes (precondition)
5. What the function computes (return value)
6. Side effects
7. Examples of function usage

Specifications may be given at different levels of detail.

Specifications may be given in natural language, or more formally.

Patterns

A **pattern** is either a constant, identifier, underscore (_) or a “skeleton” for a datatype (e.g., tuples) where the skeleton components are patterns.

Patterns

A **pattern** is either a constant, identifier, underscore (_) or a "skeleton" for a datatype (e.g., tuples) where the skeleton components are patterns.

Identifiers and underscore match any value. A constant pattern matches only an equal value (checked using ==). Skeletons match if the value is of the same shape, and all component patterns match.

Patterns are checked in the order they appear in the source file.

Patterns

A **pattern** is either a constant, identifier, underscore (_) or a "skeleton" for a datatype (e.g., tuples) where the skeleton components are patterns.

Identifiers and underscore match any value. A constant pattern matches only an equal value (checked using ==). Skeletons match if the value is of the same shape, and all component patterns match.

Patterns are checked in the order they appear in the source file.

Guarded patterns are of the form `pattern | expr`. Here, `expr` must have type `Bool` and will usually refer to names from `pattern`.

Uses of Patterns

Function declarations:

```
name pattern1 = expression1
name pattern2 = expression2
...
name patternN = expressionN
```

Uses of Patterns

Function declarations:

```
name pattern1 = expression1
name pattern2 = expression2
...
name patternN = expressionN
```

Case expressions:

```
case expr of
  pattern1 -> expression1
  pattern2 -> expression2
  ...
  patternN -> expressionN
```

Haskell Sources

By convention, we use the extension `.hs` for files that contain Haskell source code: e.g., `example.hs`.

Haskell Sources

By convention, we use the extension `.hs` for files that contain Haskell source code: e.g., `example.hs`.

- Top-level expressions are not allowed.
- Usually, each declaration is given on a separate line.
- The order of declarations doesn't matter.
- Multiple declarations of the same identifier are not allowed.