Fundamental Concepts

CS4450/7450

Professor William Harrison

February 6, 2017

Overview

- *Scripts* are collections of definitions.
- *Definitions* are equations describing mathematical functions.
- *Type signatures* describe the basic behavior of definitions.
- *Sessions* are interactions between the programmer and language system during which *expressions* are *evaluated*.
- Expressions evaluated in a session may refer to definitions given in a script.

GHCi Sessions

Starting the interpreter:

```
Athena:~ Bill$ ghci
```

This spits out the following:

```
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for
    help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

The prompt, “`Prelude>`”, signifies that GHCi has loaded the Haskell prelude, which is a script auto-loaded every time GHCi starts.

GHCi Sessions

Starting the interpreter:

```
Athena:~ Bill$ ghci
```

This spits out the following:

```
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for
    help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

The prompt, “`Prelude>`”, signifies that GHCi has loaded the Haskell prelude, which is a script auto-loaded every time GHCi starts.

GHCi Sessions

Starting the interpreter:

```
Athena:~ Bill$ ghci
```

This spits out the following:

```
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for
    help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

The prompt, "`Prelude>`", signifies that GHCi has loaded the Haskell prelude, which is a script auto-loaded every time GHCi starts.

Interactive Sessions

You interact with GHCi during a session.

```
Prelude> 42
  42
Prelude> 6 * 7
  42
Prelude> :type 42
  42 :: Num a => a
Prelude> :quit
  Leaving GHCi.
```

Writing Scripts

- “Script” = “Bunch of definitions in a file”.
- Create a script with a text editor (e.g., vi, emacs, notepad, etc.); the following saved in `Script.hs`.

```
square  :: Integer -> Integer
square x = x * x
smaller :: (Integer,Integer) -> Integer
smaller (x,y) = if x <= y then x else y
```

- Load the script while in GHCi:

```
Prelude> :load Script
[1 of 1] Compiling Main  ( Script.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

- Evaluate expressions using definitions in script:

```
*Main> square 14198724
201603763228176
*Main>
```

Writing Scripts

- “Script” = “Bunch of definitions in a file”.
- Create a script with a text editor (e.g., vi, emacs, notepad, etc.); the following saved in `Script.hs`.

```
square  :: Integer -> Integer
square x = x * x
smaller :: (Integer,Integer) -> Integer
smaller (x,y) = if x <= y then x else y
```

- Load the script while in GHCi:

```
Prelude> :load Script
[1 of 1] Compiling Main    ( Script.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

- Evaluate expressions using definitions in script:

```
*Main> square 14198724
201603763228176
*Main>
```

Writing Scripts

- "Script" = "Bunch of definitions in a file".
- Create a script with a text editor (e.g., vi, emacs, notepad, etc.); the following saved in `Script.hs`.

```
square  :: Integer -> Integer
square x = x * x
smaller :: (Integer,Integer) -> Integer
smaller (x,y) = if x <= y then x else y
```

- Load the script while in GHCi:

```
Prelude> :load Script
[1 of 1] Compiling Main  ( Script.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

- Evaluate expressions using definitions in script:

```
*Main> square 14198724
201603763228176
*Main>
```

Writing Scripts

- "Script" = "Bunch of definitions in a file".
- Create a script with a text editor (e.g., vi, emacs, notepad, etc.); the following saved in `Script.hs`.

```
square  :: Integer -> Integer
square x = x * x
smaller :: (Integer,Integer) -> Integer
smaller (x,y) = if x <= y then x else y
```

- Load the script while in GHCi:

```
Prelude> :load Script
[1 of 1] Compiling Main  ( Script.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

- Evaluate expressions using definitions in script:

```
*Main> square 14198724
201603763228176
*Main>
```

Evaluation

- Expressions are *evaluated* (a.k.a., *simplified*, *reduced*) by simplifying until no other simplification is possible.
- Non-deterministic: Usually a choice of simplifications

E.g., here are two "reduction sequences":

```
square (3+4)
      = square 7
      = 7 * 7
      = 49
```

```
square (3+4)
      = (3+4) * (3+4)
      = 7 * (3 + 4)
      = 7 * 7
      = 49
```

N.b., just "replacing equals for equals".

Evaluation

- Expressions are *evaluated* (a.k.a., *simplified*, *reduced*) by simplifying until no other simplification is possible.
- Non-deterministic: Usually a choice of simplification**s**

E.g., here are two "reduction sequences":

```
square (3+4)
     = square 7
     = 7 * 7
     = 49
```

```
square (3+4)
     = (3+4) * (3+4)
     = 7 * (3 + 4)
     = 7 * 7
     = 49
```

N.b., just "replacing equals for equals".

Evaluation

- Expressions are *evaluated* (a.k.a., *simplified*, *reduced*) by simplifying until no other simplification is possible.
- Non-deterministic: Usually a choice of simplification**s**

E.g., here are two "reduction sequences":

```
square (3+4)
     = square 7
     = 7 * 7
     = 49
```

```
square (3+4)
     = (3+4) * (3+4)
     = 7 * (3 + 4)
     = 7 * 7
     = 49
```

N.b., just "replacing equals for equals".

Evaluation

- Expressions are *evaluated* (a.k.a., *simplified*, *reduced*) by simplifying until no other simplification is possible.
- Non-deterministic: Usually a choice of simplification**s**

E.g., here are two "reduction sequences":

```
square (3+4)
      = square 7
      = 7 * 7
      = 49
```

```
square (3+4)
      = (3+4) * (3+4)
      = 7 * (3 + 4)
      = 7 * 7
      = 49
```

N.b., just "replacing equals for equals".

Termination & Non-termination

Not all evaluation terminates. Consider:

```
three :: Integer -> Integer
three x = 3
infinity :: Integer
infinity = infinity + 1
```

These are two possible reduction sequences.

```
three infinity                          three infinity
     = three (infinity + 1)                  = 3
     = three (infinity + 1 + 1)
                  ⋮
```

- Reduction sequences for an expression may include both terminating and non-terminating ones.
- *Order of evaluation matters!* Haskell's *lazy evaluation* finds a terminating red-seq if one exists. (Ch. 7).

Termination & Non-termination

Not all evaluation terminates. Consider:

```
three :: Integer -> Integer
three x = 3
infinity :: Integer
infinity = infinity + 1
```

These are two possible reduction sequences.

```
three infinity
     = three (infinity + 1)
     = three (infinity + 1 + 1)
                 ⋮
```

```
three infinity
     = 3
```

- Reduction sequences for an expression may include both terminating and non-terminating ones.
- *Order of evaluation matters!* Haskell's *lazy evaluation* finds a terminating red-seq if one exists. (Ch. 7).

Termination & Non-termination

Not all evaluation terminates. Consider:

```
three :: Integer -> Integer
three x = 3
infinity :: Integer
infinity = infinity + 1
```

These are two possible reduction sequences.

```
three infinity
     = three (infinity + 1)
     = three (infinity + 1 + 1)
                  ⋮
```

```
three infinity
     = 3
```

- Reduction sequences for an expression may include both terminating and non-terminating ones.
- *Order of evaluation matters!* Haskell's *lazy evaluation* finds a terminating red-seq if one exists. (Ch. 7).

Purity vs. Impurity

The following C program **returns** 6:

```
#include <stdio.h>

int main () {
   int x,i;
   x = 0;
   for (i=1; i<=3; i++) {
     x = x + i;
     printf("x is %d\n",x);
   }
   return x;
}
```

The following Haskell expressions **are** 6:

```
sum [1,2,3]
1 + 2 + 3
6
```

Purity vs. Impurity

The following C program **returns** 6:

```
#include <stdio.h>

int main () {
   int x,i;
   x = 0;
   for (i=1; i<=3; i++) {
     x = x + i;
     printf("x is %d\n",x);
   }
   return x;
}
```

The following Haskell expressions **are** 6:

```
sum [1,2,3]
1 + 2 + 3
6
```

Values and Expressions

- Impurity: C program returns `6` but what it does is not captured by its `int` type.
- Purity: Haskell expressions **denote** the value `6`.
- Name/Object distinction: think of each Haskell expression (e.g., `sum` $[1, 2, 3]$) as a name for an object (e.g., 6).

The distinction between syntax and semantics is critical to the study of programming languages.

Values and Expressions

- Impurity: C program returns `6` but what it does is not captured by its `int` type.
- Purity: Haskell expressions **denote** the value `6`.
- Name/Object distinction: think of each Haskell expression (e.g., `sum` $[1,2,3]$) as a name for an object (e.g., 6).

The distinction between syntax and semantics is critical to the study of programming languages.

Values and Expressions

- Impurity: C program returns `6` but what it does is not captured by its `int` type.
- Purity: Haskell expressions **denote** the value `6`.
- Name/Object distinction: think of each Haskell expression (e.g., `sum` $[1, 2, 3]$) as a name for an object (e.g., 6).

The distinction between syntax and semantics is critical to the study of programming languages.

Values and Expressions

- Impurity: C program returns `6` but what it does is not captured by its `int` type.
- Purity: Haskell expressions **denote** the value `6`.
- Name/Object distinction: think of each Haskell expression (e.g., `sum` $[1, 2, 3]$) as a name for an object (e.g., 6).

The distinction between syntax and semantics is critical to the study of programming languages.

Functions **are** Values

- Functional languages have all of the usual sorts of basic values: integers, floats, etc.
- They also have “first class” functions.
- “First class” means that functions can be passed and returned just as with a “normal” value
- Consider this definition:

```
add :: Integer -> Integer -> Integer
add x y = x + y
```

- When loaded into GHCi:

```
*Main> :type add
add :: Integer -> Integer -> Integer
*Main> :type add 9
add 9 :: Integer -> Integer
*Main> let foo = add 9
*Main> foo 6
15
```

Functions **are** Values

- Functional languages have all of the usual sorts of basic values: integers, floats, etc.
- They also have “first class” functions.
- “First class” means that functions can be passed and returned just as with a “normal” value
- Consider this definition:

```
add :: Integer -> Integer -> Integer
add x y = x + y
```

- When loaded into GHCi:

```
*Main> :type add
add :: Integer -> Integer -> Integer
*Main> :type add 9
add 9 :: Integer -> Integer
*Main> let foo = add 9
*Main> foo 6
15
```

Functions **are** Values

- Functional languages have all of the usual sorts of basic values: integers, floats, etc.
- They also have "first class" functions.
- "First class" means that functions can be passed and returned just as with a "normal" value
- Consider this definition:

```
add :: Integer -> Integer -> Integer
add x y = x + y
```

- When loaded into GHCi:

```
*Main> :type add
add :: Integer -> Integer -> Integer
*Main> :type add 9
add 9 :: Integer -> Integer
*Main> let foo = add 9
*Main> foo 6
15
```

Functions **are** Values

- Functional languages have all of the usual sorts of basic values: integers, floats, etc.
- They also have "first class" functions.
- "First class" means that functions can be passed and returned just as with a "normal" value
- Consider this definition:

```
add :: Integer -> Integer -> Integer
add x y = x + y
```

- When loaded into GHCi:

```
*Main> :type add
add :: Integer -> Integer -> Integer
*Main> :type add 9
add 9 :: Integer -> Integer
*Main> let foo = add 9
*Main> foo 6
15
```

Functions **are** Values

- Functional languages have all of the usual sorts of basic values: integers, floats, etc.
- They also have “first class” functions.
- “First class” means that functions can be passed and returned just as with a “normal” value
- Consider this definition:

```
add :: Integer -> Integer -> Integer
add x y = x + y
```

- When loaded into GHCi:

```
*Main> :type add
add :: Integer -> Integer -> Integer
*Main> :type add 9
add 9 :: Integer -> Integer
*Main> let foo = add 9
*Main> foo 6
15
```

First Class Functions

Recall the function:

```
square :: Integer -> Integer
square x = x * x
```

Here's a function that takes a function as input:

```
twice :: (Integer -> Integer) -> Integer -> Integer
twice f x = f (f x)
```

For example:

```
*Main> twice square 4
256
```

Q: What are the types of `twice square 4` and `twice square`?

First Class Functions

Recall the function:

```
square :: Integer -> Integer
square x = x * x
```

Here's a function that takes a function as input:

```
twice :: (Integer -> Integer) -> Integer -> Integer
twice f x = f (f x)
```

For example:

```
*Main> twice square 4
256
```

Q: What are the types of `twice square 4` and `twice square`?

An Aside on Types and Applications

- "−> associates to the right". That is, a type like

  ```
  Integer -> Integer -> Integer
  ```

  is really shorthand for:

  ```
  Integer -> (Integer -> Integer)
  ```

- Function application is to an input argument is expressing by juxtaposition
  - `square` 9 is a function application of `square` to 9.
- "Application associates to the left". That is, a function application like

  ```
  add 7 8
  ```

  is really shorthand for:

  ```
  ((add 7) 8)
  ```

An Aside on Types and Applications

- "->" associates to the right". That is, a type like

  ```
  Integer -> Integer -> Integer
  ```

  is really shorthand for:

  ```
  Integer -> (Integer -> Integer)
  ```

- Function application is to an input argument is expressing by juxtaposition
  - `square` 9 is a function application of `square` to 9.

- "Application associates to the left". That is, a function application like

  ```
  add 7 8
  ```

  is really shorthand for:

  ```
  ((add 7) 8)
  ```

An Aside on Types and Applications

- "−> associates to the right". That is, a type like

  ```
  Integer -> Integer -> Integer
  ```

  is really shorthand for:

  ```
  Integer -> (Integer -> Integer)
  ```

- Function application is to an input argument is expressing by juxtaposition
  - `square` 9 is a function application of `square` to 9.
- "Application associates to the left". That is, a function application like

  ```
  add 7 8
  ```

  is really shorthand for:

  ```
  ((add 7) 8)
  ```

Extensionality

- Consider the following functions:

```
double, double' :: Integer -> Integer
double x  = x + x
double' x = 2 * x
```

- These functions are equal — but why?
- They are equal because, for any `Integer i`,

```
double i == double' i
```

- This is called the principle of **extensionality**, which says that, *two functions are equal if they are equal on all inputs.*

Extensionality

- Consider the following functions:

```
double, double' :: Integer -> Integer
double x  = x + x
double' x = 2 * x
```

- These functions are equal — but why?
- They are equal because, for any `Integer i`,

```
double i == double' i
```

- This is called the principle of **extensionality**, which says that, *two functions are equal if they are equal on all inputs.*

Extensionality

- Consider the following functions:

```
double, double' :: Integer -> Integer
double x  = x + x
double' x = 2 * x
```

- These functions are equal — but why?
- They are equal because, for any `Integer i`,

```
double i == double' i
```

- This is called the principle of **extensionality**, which says that, *two functions are equal if they are equal on all inputs.*

Currying

- Consider the following functions:

```
smaller :: (Integer,Integer) -> Integer
smaller (x,y) = if x <= y then x else y
smallerc :: Integer -> Integer -> Integer
smallerc x y = if x <= y then x else y
```

- `smaller == smallerc`?
- `smaller` is a function that takes a single input—a pair of `Integer`s—while `smallerc` takes two inputs (each of which is an `Integer`).
- `smallerc` is in “curried” form, while `smaller` is in “uncurried” form.
- Named for American logician, Haskell Curry (1900-1982).

Currying

- Consider the following functions:

```
smaller :: (Integer,Integer) -> Integer
smaller (x,y) = if x <= y then x else y
smallerc :: Integer -> Integer -> Integer
smallerc x y = if x <= y then x else y
```

- `smaller == smallerc`?
- `smaller` is a function that takes a single input—a pair of `Integer`s—while `smallerc` takes two inputs (each of which is an `Integer`).
- `smallerc` is in “curried” form, while `smaller` is in “uncurried” form.
- Named for American logician, Haskell Curry (1900-1982).

Currying

- Consider the following functions:

```
smaller :: (Integer,Integer) -> Integer
smaller (x,y) = if x <= y then x else y
smallerc :: Integer -> Integer -> Integer
smallerc x y = if x <= y then x else y
```

- `smaller == smallerc`?
- `smaller` is a function that takes a single input—a pair of `Integer`s—while `smallerc` takes two inputs (each of which is an `Integer`).
- `smallerc` is in "curried" form, while `smaller` is in "uncurried" form.
- Named for American logician, Haskell Curry (1900-1982).

Currying

- Consider the following functions:

```
smaller :: (Integer,Integer) -> Integer
smaller (x,y) = if x <= y then x else y
smallerc :: Integer -> Integer -> Integer
smallerc x y = if x <= y then x else y
```

- `smaller == smallerc`?
- `smaller` is a function that takes a single input—a pair of `Integer`s—while `smallerc` takes two inputs (each of which is an `Integer`).
- `smallerc` is in “curried” form, while `smaller` is in “uncurried” form.
- Named for American logician, Haskell Curry (1900-1982).

Currying

- Consider the following functions:

```
smaller :: (Integer,Integer) -> Integer
smaller (x,y) = if x <= y then x else y
smallerc :: Integer -> Integer -> Integer
smallerc x y = if x <= y then x else y
```

- `smaller == smallerc`?
- `smaller` is a function that takes a single input—a pair of `Integer`s—while `smallerc` takes two inputs (each of which is an `Integer`).
- `smallerc` is in “curried” form, while `smaller` is in “uncurried” form.
- Named for American logician, Haskell Curry (1900-1982).

Currying Function

Currying can be written as a function:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f a b = f (a,b)
```

While `smaller` $\neq$ `smallerc`, they are equivalent in a sense:

```
curry smaller i j == smaller (i,j)
                  == if i<=j then i else j
                  == smallerc i j
```

Currying Function

Currying can be written as a function:

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f a b = f (a,b)
```

While `smaller` $\neq$ `smallerc`, they are equivalent in a sense:

```
curry smaller i j == smaller (i,j)
                  == if i<=j then i else j
                  == smallerc i j
```

Function Composition

- If we have `square`, we can define `quad`, which raises its argument to the 4th.

```
quad :: Integer -> Integer
quad x = square (square x)
   where square :: Integer -> Integer
         square x = x * x
```

- In high school algebra, this is just “function composition” of `square` with itself.
- In standard mathematical notation, this would be `square` ∘ `square`.
- In Haskell, this is:

```
quad :: Integer -> Integer
quad = square . square
   where square :: Integer -> Integer
         square x = x * x
```

Function Composition

- If we have `square`, we can define `quad`, which raises its argument to the 4th.

```
quad :: Integer -> Integer
quad x = square (square x)
   where square :: Integer -> Integer
         square x = x * x
```

- In high school algebra, this is just "function composition" of `square` with itself.
- In standard mathematical notation, this would be `square` $\circ$ `square`.
- In Haskell, this is:

```
quad :: Integer -> Integer
quad = square . square
   where square :: Integer -> Integer
         square x = x * x
```

Function Composition

- If we have `square`, we can define `quad`, which raises its argument to the 4th.

```
quad :: Integer -> Integer
quad x = square (square x)
   where square :: Integer -> Integer
         square x = x * x
```

- In high school algebra, this is just "function composition" of `square` with itself.
- In standard mathematical notation, this would be `square` $\circ$ `square`.
- In Haskell, this is:

```
quad :: Integer -> Integer
quad = square . square
   where square :: Integer -> Integer
         square x = x * x
```

Just FYI: Defining Composition

Composition operator `(.)` defined in the Haskell prelude as:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) a = f (g a)
```

Composition is associative:

```
f . (g . h) == (f . g) . h
```

Therefore, we can simply write: `f . g . h`

Just FYI: Defining Composition

Composition operator `(.)` defined in the Haskell prelude as:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) a = f (g a)
```

Composition is associative:

```
f . (g . h) == (f . g) . h
```

Therefore, we can simply write: `f . g . h`

Definitions with Guards

- Recall the definition of smaller:

```
smaller :: (Integer,Integer) -> Integer
smaller (x,y) = if x<=y then x else y
```

- An equivalent definition uses **guards**:

```
smaller :: (Integer,Integer) -> Integer
smaller (x,y) | x<=y       = x
              | otherwise = y
```

- The "| *test*" are guards.
- Each guard test is checked in order until a true one is found; corresponding branch is returned.
- The `else` case is the `otherwise` guard.
- Guards are *syntactic sugar*—any guarded definition could be translated into one using only `if − then − else`.

Definitions with Guards

- Recall the definition of smaller:

```
smaller :: (Integer,Integer) -> Integer
smaller (x,y) = if x<=y then x else y
```

- An equivalent definition uses **guards**:

```
smaller :: (Integer,Integer) -> Integer
smaller (x,y) | x<=y      = x
              | otherwise = y
```

- The "| *test*" are guards.
- Each guard test is checked in order until a true one is found; corresponding branch is returned.
- The `else` case is the `otherwise` guard.
- Guards are *syntactic sugar*—any guarded definition could be translated into one using only `if − then − else`.

Definitions with Guards

- Recall the definition of smaller:

```
smaller :: (Integer,Integer) -> Integer
smaller (x,y) = if x<=y then x else y
```

- An equivalent definition uses **guards**:

```
smaller :: (Integer,Integer) -> Integer
smaller (x,y) | x<=y      = x
              | otherwise = y
```

- The "| *test*" are guards.
- Each guard test is checked in order until a true one is found; corresponding branch is returned.
- The `else` case is the `otherwise` guard.
- Guards are *syntactic sugar*—any guarded definition could be translated into one using only `if − then − else`.

Recursion

- Recursion is the main engine in a functional language.
- Loops are, in fact, recursive constructs. The loop, `while (b) { c }` can be written as a recursive procedure:

```
procwhile b c = if b
                  then (c ; procwhile b c)
                  else skip
```

- No surprises:

$$n\,! = \begin{cases} 1 & \textit{when}\ (n == 0) \\ n \times ((n-1)!) & \textit{otherwise} \end{cases}$$

- ...can be written:

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

Recursion

- Recursion is the main engine in a functional language.
- Loops are, in fact, recursive constructs. The loop, `while` (b) { c } can be written as a recursive procedure:

```
procwhile b c = if b
                 then (c ; procwhile b c)
                 else skip
```

- No surprises:

$$n\,! = \begin{cases} 1 & \textit{when}\ (n == 0) \\ n \times ((n-1)!) & \textit{otherwise} \end{cases}$$

- ...can be written:

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

Recursion

- Recursion is the main engine in a functional language.
- Loops are, in fact, recursive constructs. The loop, `while` (b) { c } can be written as a recursive procedure:

```
procwhile b c = if b
                 then (c ; procwhile b c)
                 else skip
```

- No surprises:

$$\mathtt{n}\,! = \begin{cases} 1 & \mathit{when}\ (n == 0) \\ n \times ((n-1)!) & \mathit{otherwise} \end{cases}$$

- ...can be written:

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

Recursion

- Note that it may also written with guards:

```
fact :: Integer -> Integer
fact n | n==0      = 1
       | otherwise = n * (fact (n-1))
```

- Guards provide a convenient means for input checking:

```
fact :: Integer -> Integer
fact n | n<0       = error "Argghhh..."
       | n==0      = 1
       | otherwise = n * (fact (n-1))
```

- `error` provides a way of stopping execution. Use for debugging.

```
*Main> fact (-1)
*** Exception: Argghhh...
*Main>
```

Recursion

- Note that it may also written with guards:

```
fact :: Integer -> Integer
fact n | n==0      = 1
       | otherwise = n * (fact (n-1))
```

- Guards provide a convenient means for input checking:

```
fact :: Integer -> Integer
fact n | n<0       = error "Argghhh..."
       | n==0      = 1
       | otherwise = n * (fact (n-1))
```

- `error` provides a way of stopping execution. Use for debugging.

```
*Main> fact (-1)
*** Exception: Argghhh...
*Main>
```

Recursion

- Note that it may also written with guards:

```
fact :: Integer -> Integer
fact n | n==0      = 1
       | otherwise = n * (fact (n-1))
```

- Guards provide a convenient means for input checking:

```
fact :: Integer -> Integer
fact n | n<0       = error "Argghhh..."
       | n==0      = 1
       | otherwise = n * (fact (n-1))
```

- `error` provides a way of stopping execution. Use for debugging.

```
*Main> fact (-1)
*** Exception: Argghhh...
*Main>
```

Local Definitions

- Definitions as we've seen them thus far are at the “top level” (i.e., global)

```
foo :: A -> B  -- foo can call bar
foo a = ...
bar :: A -> C  -- bar can call foo
bar a = ...
```

- Some times we want local definitions (a.k.a., helper functions) for convenience. Local definitions are hidden.

```
foo :: A -> B          -- can call bar and boo
foo a = ...
   where boo x = ...
bar :: A -> C          -- can call foo, but not boo
bar a = ...
```

- Seen this already:

```
quad :: Integer -> Integer
quad x = square (square x)
   where square :: Integer -> Integer
         square x = x * x
```

Local Definitions

- Definitions as we've seen them thus far are at the "top level" (i.e., global)

```
foo :: A -> B  -- foo can call bar
foo a = ...
bar :: A -> C  -- bar can call foo
bar a = ...
```

- Some times we want local definitions (a.k.a., helper functions) for convenience. Local definitions are hidden.

```
foo :: A -> B            -- can call bar and boo
foo a = ...
   where boo x = ...
bar :: A -> C            -- can call foo, but not boo
bar a = ...
```

- Seen this already:

```
quad :: Integer -> Integer
quad x = square (square x)
   where square :: Integer -> Integer
         square x = x * x
```

Local Definitions

- Definitions as we've seen them thus far are at the "top level" (i.e., global)

```
foo :: A -> B  -- foo can call bar
foo a = ...
bar :: A -> C  -- bar can call foo
bar a = ...
```

- Some times we want local definitions (a.k.a., helper functions) for convenience. Local definitions are hidden.

```
foo :: A -> B            -- can call bar and boo
foo a = ...
   where boo x = ...
bar :: A -> C            -- can call foo, but not boo
bar a = ...
```

- Seen this already:

```
quad :: Integer -> Integer
quad x = square (square x)
   where square :: Integer -> Integer
         square x = x * x
```

Types, what are they good for?

The main point of types and type systems is to restrict a language to sensible expressions by “weeding out” nonsense.

Ex: the following are nonsense:

```
9 + True
square square
    ...
```

- *Type*: a collection of values.
- *Type system*: rules for assigning types to expressions.
- *Strong type system* for a language insists that every expression have a type.
- *Static type system* can compute the type of an expression (if it has one).

Types, what are they good for?

The main point of types and type systems is to restrict a language to sensible expressions by "weeding out" nonsense.
Ex: the following are nonsense:

```
9 + True
square square
   ...
```

- *Type*: a collection of values.
- *Type system*: rules for assigning types to expressions.
- *Strong type system* for a language insists that every expression have a type.
- *Static type system* can compute the type of an expression (if it has one).

Types, what are they good for?

The main point of types and type systems is to restrict a language to sensible expressions by "weeding out" nonsense.
Ex: the following are nonsense:

```
9 + True
square square
   ...
```

- *Type*: a collection of values.
- *Type system*: rules for assigning types to expressions.
- *Strong type system* for a language insists that every expression have a type.
- *Static type system* can compute the type of an expression (if it has one).

Types, what are they good for?

The main point of types and type systems is to restrict a language to sensible expressions by "weeding out" nonsense.
Ex: the following are nonsense:

```
9 + True
square square
    ...
```

- *Type*: a collection of values.
- *Type system*: rules for assigning types to expressions.
- *Strong type system* for a language insists that every expression have a type.
- *Static type system* can compute the type of an expression (if it has one).

Types, what are they good for?

The main point of types and type systems is to restrict a language to sensible expressions by "weeding out" nonsense.
Ex: the following are nonsense:

```
9 + True
square square
    ...
```

- *Type*: a collection of values.
- *Type system*: rules for assigning types to expressions.
- *Strong type system* for a language insists that every expression have a type.
- *Static type system* can compute the type of an expression (if it has one).

Types, what are they good for?

The main point of types and type systems is to restrict a language to sensible expressions by "weeding out" nonsense.
Ex: the following are nonsense:

```
9 + True
square square
    ...
```

- *Type*: a collection of values.
- *Type system*: rules for assigning types to expressions.
- *Strong type system* for a language insists that every expression have a type.
- *Static type system* can compute the type of an expression (if it has one).

Just FYI: Typing rules

- Every type system (incl. Haskell's) is defined by "inference rules".
- Ex: the rule for function application looks like:

$$\frac{e_1 :: A \to B \qquad e_2 :: A}{e_1\ e_2 :: B}$$

  where A and B can be any type.
- Rule is pronounced: "*if* e_1 *has type* $A \to B$ *and* e_2 *has type* A*, then the application* $e_1\, e_2$ *has type* B*.*"

$$\frac{\mathit{twice} :: \overbrace{(\mathit{Integer} \to \mathit{Integer})}^{A} \to \overbrace{\mathit{Integer} \to \mathit{Integer}}^{B} \qquad \mathit{square} :: \overbrace{\mathit{Integer} \to \mathit{Integer}}^{A}}{\mathit{twice}\ \mathit{square} :: \underbrace{\mathit{Integer} \to \mathit{Integer}}_{B}}$$

Just FYI: Typing rules

- Every type system (incl. Haskell's) is defined by "inference rules".
- Ex: the rule for function application looks like:

$$\frac{e_1 :: A \rightarrow B \quad e_2 :: A}{e_1\ e_2 :: B}$$

  where A and B can be any type.
- Rule is pronounced: "*if* e_1 *has type* $A \rightarrow B$ *and* e_2 *has type* A*, then the application* $e_1\,e_2$ *has type* B*.*"

$$\frac{\mathit{twice} :: \overbrace{(\mathit{Integer} \rightarrow \mathit{Integer})}^{A} \rightarrow \overbrace{\mathit{Integer} \rightarrow \mathit{Integer}}^{B} \qquad \mathit{square} :: \overbrace{\mathit{Integer} \rightarrow \mathit{Integer}}^{A}}{\mathit{twice}\ \mathit{square} :: \underbrace{\mathit{Integer} \rightarrow \mathit{Integer}}_{B}}$$

Parametric Polymorphism

- Consider the following transcript:

```
*Main> length ['x','y','z']
3
*Main> length [1,2,3]
3
```

- The first example would suggest that `length :: [Char] → Int` and the second example that `length :: [Int] → Int`.
- Which is it? `length` **must** have a single type.
- The solution: allow types to have "parameters":

```
*Main> :t length
length :: [a] -> Int
```

- This is read: for all types `a`, `length` has type `[a] → Int`.

Parametric Polymorphism

- Consider the following transcript:

```
*Main> length ['x','y','z']
3
*Main> length [1,2,3]
3
```

- The first example would suggest that `length` :: [`Char`] $\rightarrow$ `Int` and the second example that `length` :: [`Int`] $\rightarrow$ `Int`.
- Which is it? `length` **must** have a single type.
- The solution: allow types to have "parameters":

```
*Main> :t length
length :: [a] -> Int
```

- This is read: for all types `a`, `length` has type [`a`] $\rightarrow$ `Int`.

Parametric Polymorphism

- Consider the following transcript:

```
*Main> length ['x','y','z']
3
*Main> length [1,2,3]
3
```

- The first example would suggest that `length :: [Char] → Int` and the second example that `length :: [Int] → Int`.
- Which is it? `length` **must** have a single type.
- The solution: allow types to have “parameters”:

```
*Main> :t length
length :: [a] -> Int
```

- This is read: for all types `a`, `length` has type `[a] → Int`.

“Ad Hoc” Polymorphism & Type Classes

- For some operations, parametric polymorphism is too general. E.g., addition only makes sense applied to numerical values:

```
'a' + 'b'     -- makes no sense!
```

- Upshot: restrict to a class of sensible types:

```
*Main> :t (+)
(+) :: Num a => a -> a -> a
```

- Here `Num` is called a type class. All the familiar numbers are in `Num`: `Int`, `Integer`, `Float`, etc.
- The “`Num a =>`” is a class constraint. The type of $(+)$ is pronounced: for every `Num` type `a`, $(+)$ has type $a \rightarrow a \rightarrow a$.

"Ad Hoc" Polymorphism & Type Classes

- For some operations, parametric polymorphism is too general. E.g., addition only makes sense applied to numerical values:

```
'a' + 'b'     -- makes no sense!
```

- Upshot: restrict to a class of sensible types:

```
*Main> :t (+)
(+) :: Num a => a -> a -> a
```

- Here `Num` is called a type class. All the familiar numbers are in `Num`: `Int`, `Integer`, `Float`, etc.
- The "`Num a =>`" is a class constraint. The type of $(+)$ is pronounced: for every `Num` type `a`, $(+)$ has type $a \to a \to a$.

"Ad Hoc" Polymorphism & Type Classes

- For some operations, parametric polymorphism is too general. E.g., addition only makes sense applied to numerical values:

```
'a' + 'b'     -- makes no sense!
```

- Upshot: restrict to a class of sensible types:

```
*Main> :t (+)
(+) :: Num a => a -> a -> a
```

- Here `Num` is called a type class. All the familiar numbers are in `Num`: `Int`, `Integer`, `Float`, etc.
- The "`Num a =>`" is a class constraint. The type of $(+)$ is pronounced: for every `Num` type `a`, $(+)$ has type $\mathtt{a} \rightarrow \mathtt{a} \rightarrow \mathtt{a}$.