Chapter 15

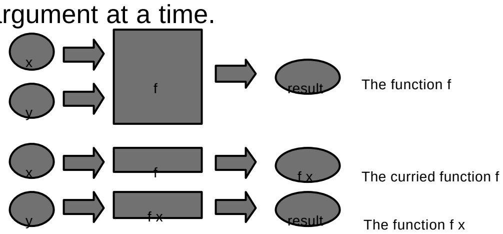
Functional Programming

Topics

- Reduction and Currying
- Recursive definitions
- Local definitions
- Type Systems
  - Strict typing
  - Polymorphism
- Types Classes
- Types
  - Booleans
  - Characters
  - Enumerations
  - Tuples
  - Strings

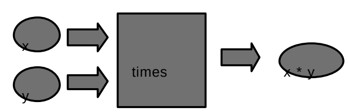
Currying

- Viewing a function with two or more arguments as a function that takes one argument at a time.



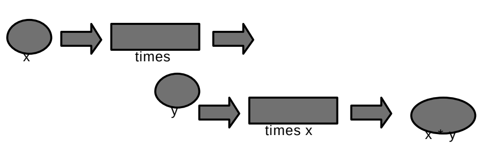
Currying: example

- The uncurried function `times` takes two numbers as inputs and return their multiplication.



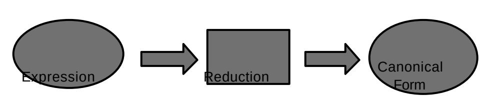
Currying: example

- The curried function `times` takes a number `x` and return the function (`times x`).
- (`times x`) takes a number `y` and returns the number (`x * y`).



Reduction

- Reduction is the process of converting a functional expression to its canonical form by repeatedly applying reduction rules



Reduction Rules

- There are two kinds of reduction rules:
  - Build-in definitions
    - For example the arithmetic operations
  - User supplied definitions

Recursive Definitions

- Definitions can also be recursive.
- Example:

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

  - This definition of `fact` is not completely satisfactory: if it is applied to a negative integer, then the computation never terminates.
  - For negative numbers, `fact x = ⊥`.
    - It is better if the computation terminated with a suitable error message rather than proceeding indefinitely with a futile computation.

Recursive Definitions

```
fact :: Integer → Integer
fact n
      | n < 0  =  error "negative argument"
      | n == 0 = 1
      | n > 0  = n * fact(n-1)
```

- The predefined function error takes a string as argument; when evaluated it causes immediate termination of the evaluator and displays the given error message.

```
? fact (-1)
Program error: negative argument
```


Local Definitions

- In mathematical descriptions there are expressions qualified by a phrase of the form "where …".
  - `f(x,y) = (a+1)(a+2)`, where `a = (x+y)/2`
- Example:

```
f :: (Float,Float) → Float
f(x,y) = (a+1) * (a+2) where a = (x+y)/2
```

  - The special word `where` is used to introduce a local definition whose context (or scope) is the expression on the RHS of the definition of `f`.

Local Definitions

- When there are two or more local definitions, there are two styles:

```
f :: (Float,Float) → Float
f(x,y) = (a+1) * (b+2)
         where a = (x+y)/2
               b = (x+y)/3

f :: (Float,Float) → Float
f(x,y) = (a+1) * (b+2)
         where a = (x+y)/2; b = (x+y)/3
```


Local Definitions

- A local definition can be used in conjunction with a definition that relies on guarded equations.:

```
f :: Integer → Integer → Integer
f x y =
      | x ≤ 10 = x + a
      | x > 10 = x-a
        where a = square(y+a)
```

  - The `where` clause qualifies both guarded equations.

Type Systems

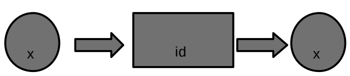
- Programming languages have either:
  - No type systems
    - Lisp, Prolog, Basic, etc
  - A strict type system
    - Pascal, Modula2
  - A polymorphic type systems
    - ML, Mirada, Haskell, Java, C++

Strong Typing Principle

- Every expression must have a type
  - 3 has type `Int`
  - `'A'` has type `Char`
- The type of a compound expression can be deduced from its constituents alone.
  - `('A',1+2)` has type `(Char, Int)`
- An expression which does not have a sensible type is illegal.
  - `'A'+3` is illegal

Strict Typing

- Every expression has a unique concrete type.
  - Although this system is good for trapping errors, it is too restrictive.



- What type should be given to `id`?
  - Is it `Int→Int`?, `Char→Char`?, `(Int,Bool)→(Int,Bool)`
- With strict typing we have to define separate versions of `id` for each type.

Polymorphism

- Polymorphism allows the definition of certain functions to be used with different types.
- Without polymorphism we would have to write different versions of the function for each possible type (type declaration is different but the body is the same).
- Polymorphism results in simpler, more general, reusable and concise programs.

Type Classes

- A curried multiplication can be used with two different type signatures:

  (x) :: Integer → Integer → Integer
  (x) :: Float → Float → Float

- So, it can be assigned a polymorphic type:

  (x) :: α → α → α

  - This type is too general (two characters or two booleans should not be multiplied).

Type Classes

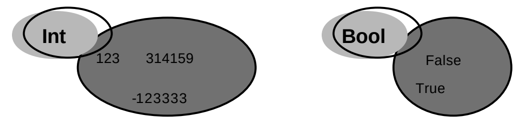
- Group together kindred types into *type classes*.
  - `Integer` and `Float` belong to the same class, the class of numbers.

    (x) :: Num α ⇒ α → α → α
- There are other kindred types apart from numbers.
  - The types whose value can be displayed, the types whose value can be compared for equality, the type whose value can be enumerated, etc.

Types

- In addition to defining functions and constants, functional languages allows to define types to build new and useful types from existing ones.
- The universe of values is divided into organized collections, called *types*.
  - Integer, Float, Double, booleans, characters, lists, trees, etc.
  - An infinity variety of other types can be put together: Integer → Float, (Float, Float), etc.

Types



- Each type has associated with it certain operations which are not meaningful for other types.

Type Declaration

- The type of an expression is declared using the following convention:

  `expression :: type`

  - Example: `e :: t`
    - Reads: "the expression e has the type `t`"

    - `pi :: Double`
    - `Square :: Integer → Integer`

Types

- *Strong typing*: the value of an expression depends only on the values of its component expressions, so does its type.
- Consequence of strong typing
  - Any expression which cannot be assigned a sensible type is not well formed and is rejected by the computer before evaluation (illegal expressions).

Types

```
quad :: Integer → Integer
quad x = square square x
```

- Advantage of strong typing
  - Enables a range of errors to be detected before evaluation.

- There are two stages of analysis when a expression is submitted for evaluation.

Types

- The expression is checked to see whether it conforms to the correct syntax laid down for constructing expressions.
  - No: the computer signals a *syntax error*
  - Yes: perform the second stage of evaluation

- The expression is analysed to see if it posses a sensible type
  - Fails: the computer signals a *type error*.
  - Yes: the expression is evaluated.

Classification of Types

- Basic/Simple Types
  - Contain primitive values
- User-defined Types
  - Contain user-defined values
- Derived Types
  - Contain more complex values

Simple Data Types: booleans

- Used to define the truth value of a conditional expression.
  - There are two truth values, `True` and `False`.
  - These two values comprise the datatype `Bool` of boolean values.
  - `True`, `False` and `Bool` begin with a capital letter.
  - The datatype `Bool` can be introduce with a *datatype declaration:*

```
data Bool = False | True
```


Simple Data Types: booleans

- Having introduce `Bool`, it is possible to define functions that take boolean arguments by *pattern matching*.
  - Example: the negation function

```
not :: Bool → Bool
not False = True
not True = False
```

    - To simplify expressions of the form `not e`: first `e` is reduced to normal form.
      - If e cannot be reduced to normal form then the value of `not e` is undefined
      - `not ⊥ = ⊥` then `not` is strict.

Simple Data Types: booleans

- There are not two but thee boolean values: `True`, `False`, and ⊥.
- Every datatype declaration introduces an extra anonymous value, the undefined value of the datatype.
- More examples: conjunction, disjunction.

Simple Data Types: booleans

- This is how pattern matching works:

```
⊥ ∧ True = ⊥
⊥ ∧ False = ⊥
False ∧ ⊥ = False
True ∧ ⊥ = ⊥
```

  - ∧ is strict in its LHS, but nonstrict in its RHS argument.

Booleans: equality operators

- There are two equality operators = = and ≠

```
(==) :: Bool → Bool → Bool
x == y = (x∧y) ∨ (not x ∧ not y)
(≠) :: Bool → Bool → Bool
x ≠ y = not(x == y)
```

- The symbol == is used to denote a computable test for equality.
- The symbol = is used both in definitions and its normal mathematical sense.

Booleans: equality operators

- The main purpose of introducing an equality test is to be able to use it with a range of different types.
  - (==) and (≠) are *overloaded operations*.
- The proper way to introduce them is first to declare a type class `Eq` consisting of all those types for which (==) and (≠) are to be defined.

Booleans: equality operators

```
class Eq α where
  (=),(≠) :: α → α → Bool
```

- To declare that a certain type is an instance of the type class `Eq`, an *instance declaration* is needed.

```
instance Eq Bool where
(x == y) = (x ∧ y) ∨ (not x ∧ not y)
(x ≠ y) = not(x == y)
```


Booleans: comparison operators

- Booleans can also be compared.
  - Comparison operations are also overloaded and make sense with elements from a number of different types.

```
 class (Eq α) ⇒ Ord α where
(<),(≤),(≥),(>) :: α → α → Bool
(x ≤ y) = (x < y) ∨ (x == y)
(x ≥ y) = (x > y) ∨ (x == y)
(x > y) = not(x ≤ y)
```


Booleans: comparison operators

- `Bool` could be an instance of `Ord`:

```
instance Ord Bool where
      False ≤  False = False
      False ≤  True = True
      True ≤  False = False
      True ≤  True = False
```


Example: leap years

- Define a function to determine whether a year is a leap year or not.
  - A leap year is divisible by 4, except that if it is divisible by 100, then it must also be divisible by 400.

```
leapyear :: Int → Bool
leapyear y = (y mode 4 == 0) ∧
             (y mode 100 ≠ 0 ∨ (y mode 400 == 0)
```

  - Using conditional expressions:

```
leapyear y = if (y mode 100==0)
             then (y mode 400 ==0)
             else (y mode 4 == 0)
```


Characters

- Characters are denoted by enclosing them in single quotation marks.
  - Remember: the character `'7'` is different from the decimal number `7`.
- Two primitive functions are provided for processing characters, `ord` and `chr`.
  - Their types are:

```
ord :: Char → Int
chr :: Int → Char
```

Characters

- The function `ord` converts a character `c` to an integer `ord c` in the range `0 ≤ ord c ≤ 256`
- The function `chr` does the reverse, converting an integer back into the character it represents.
- Thus `chr (ord c) = c` for all characters `c`.

```
? ord'b'                    ? chr98
  98                          'b'
? chr(ord'b'+1)
  'c'
```


Characters

- Characters can be compared and tested for equality.

```
instance Eq Char where
  (x == y) = (ord x == ord y)

instance Ord Char where
  (x < y) = (ord x < ord y)

? '0' < '9'             ? 'A' < 'Z'
True                    True
```


Characters: simple functions

- Three functions for determining whether a character is a digit, lower-case letter, or upper-case letter:

```
isDigit,isLower,isUpper :: Char → Bool
isDigit c = ('0' ≤ c) ∧ (c ≤ '9')
isLower c = ('a' ≤ c) ∧ (c ≤ 'z')
isUpper c = ('A' ≤ c) ∧ (c ≤ 'Z')
```

- A function for converting lower-case letter to upper-case:

```
capitalise :: Char → Char
capitalise c = if isLower c then
                  chr(offset+ord c) else c
                 where offset = ord 'A' – ord 'a'
```


Enumerations

- They are user-defined types.
- Example:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

  - This definition binds the name `Day` to a new type that consists of eight distinct values, seven of which are represented by the given constants and the eight by the undefined value ⊥
    - The seven new constants are called the constructors of the datatype Day.
    - By convention, constructor names and the new name begin with an upper-case letter.

Enumerations

- It is possible to compare elements of type `Day`, so `Day` can be declared as an instance of the type classes `Eq` and `Ord`.
  - A definition of (==) and (<) based on pattern matching would involve a large number of equations.
- Better idea. Code elements of `Day` as integers, and use integer comparison instead.

Enumerations

- Since the same idea can be employed with other enumerated types, a new type class `Enum` is declared
  - `Enum` describes types whose elements can be enumerated.

```
class Enum α where
  fromEnum :: α → Int
  toEnum   :: Int → α
```

  - A type is declared an instance of `Enum` by giving definition of `toEnum` and `fromEnum`, functions that convert between elements of the type and `Int`.

Enumerations: example

- Day is a member of `Enum`:

```
instance Enum Day where
     fromEnum Sun = 0
     fromEnum Mon = 1
     fromEnum Tue = 2
     fromEnum Wed = 3
     fromEnum Thu = 4
     fromEnum Fri = 5
     fromEnum Sat = 6
```


Enumerations: example

- Given fromEnum on Day:

```
instance Eq Day where
(x == y) = (fromEnum x == fromEnum y)

instance Ord Day where
(x < y) = (fromEnum x < fromEnum y)
```


Enumerations: example

```
workday   :: Day → Bool
workday d = (Mon ≤ d) ∧ (d ≤ Fri)

restday   :: Day → Bool
restday d = (d==Sat) ∨ (d==Sun)

dayafter   :: Day → Day
dayafter d = toEnum((fromEnum d+1) mod 7)
```


Automatic instance declarations

- Haskell provides a mechanism for declaring a type as an instance of `Eq`, `Ord`, and `Enum` in one declaration.

```
data Day =   Sun | Mon | Tue | Wed |
             Thu | Fri | Sat
             deriving (Eq,Ord,Enum)
```

  - The deriving clause causes the evaluator to generate instance declarations of the named type classes automatically.

Tuples

- One way of combining types to form new ones is by pairing them.
  - Example: `(Integer, Char)` consists of all pairs of values `(x,c)` for which `x` is an arbitrary-precision integer, and `c` is a character.
- Like other types, the type (α,β) contains an additional value $\perp$

Tuples: practical example

- A function returns a pair of numbers, the two real roots of a quadratic equation with coefficients (a,b,c):

```
roots :: (Float, Float, Float) → (Float,Float)
roots (a,b,c)
      | a == 0    = error "not quadratic"
      | e < 0     = error "complex roots"
      | otherwise = ((-b-r)/d,(-b+r)/d)
      where r = sqrt e
            d = 2*a
            e = b*b-4*a*c
```

Other Types

- A type can be declared by typing its constants or with values that depend on those of other types.

```
data Either = Left Bool | Right Char
```

  - This declares a type `Either` whose values are denoted by expressions of the form `Left b`, where b is a boolean, and `Right c`, where c is a character.
  - There are 3 boolean values (including ⊥) and 257 characters (including ⊥), so there are 261 distinct values of the type `Either`; these include `Left ⊥`, `Right ⊥`, and ⊥

Other Types

- In general:

```
data Either α β = Left α | Right β
```

- The names `Left` and `Right` introduces two constructors for building values of type `Either`, these constructors are nonstrict functions with types:

```
Left  :: α → Either α β
Right :: β → Either α β
```


Other Types

- Assuming that values of types α and β can be compared, comparison on that type `Either α β` can be added as an instance declaration:

```
instance (Eq α,Eq β ) ⇒ Eq(Either α β) where
        Left x == Left y     = (x==y)
        Left x == Right y    = False
        Right x == Left y    = False
        Right x == Right y  = (x==y)
instance (Ord α,Ord β ) ⇒ Ord(Either α β) where
        Left x < Left y      = (x<y)
        Left x < Right y     = True
        Right x < Left y     = False
        Right x < Right y    = (x<y)
```


Type Synonyms

- *Type synonym declaration*: a simple notation for giving alternative names to types.
- Example:

```
roots :: (Float, Float, Float) → (Float,Float)
```

  - As an alternative, two type synonyms could be used

```
type Coeffs = (Float, Float, Float)
type Roots  = (Float,Float)
```


Type Synonyms

  - This declarations do not introduce new types but merely alternative names for existing types.

```
roots :: Coeffs → Roots
```

  - This new description is shorter and more informative.
- Type synonyms can be general.

```
type Pairs α     = (α,α)
type Automorph α = α → α
type Flag α      = (α,Bool)
```


Type Synonyms

- Type synonyms cannot be declared in terms of each other since every synonym must be expressible in terms of existing types.
- Synonyms can be declared in terms of another synonym.

```
type Bools = PairBool
```

- Synonyms and declarations can be mixed

```
data OneTwo α = One α | Two(Pairs α)
```

Strings

- A list of characters is called a *string*.
- The type `String` is a synonym type:
```
type String = [Char]
```
- Syntax: the characters of a string are enclosed in double quotation marks.
- `‘a’` vs. `“a”`
  - the former is a character
  - the latter is a list of characters that happens to contain only one element.

Strings

- Strings cannot be declared separately as instances of `Eq` and `Ord` because they are just synonyms.
  - They inherit whatever instances are declared for general lists.
- Comparison on strings follow the normal lexicographic ordering.
```
? “hello” < “hallo”
False
? “Jo” < “Joanna”
True
```


Strings

- Haskell provides a primitive command for printing strings.
```
putStr :: String → IO()
```
  - Evaluating the command `putStr` causes the string to be printed literally.
```
? putStr “Hello World”
Hello World
? putStr “This sentence contains \n a newline”
This sentence contains
a newline
```


The type class Show

- Haskell provides a special type class `Show` to display information of different kinds and formats.
```
class Show α where
  showsPrec :: Int → α → String → String
```
  - The function `showsPrec` is provided for displaying values of type α
  - Using `showsPrec` it is possible to define a simpler function that takes a value and converts it to a string.
```
show :: Show α ⇒ α → String
```


The type class Show

- Example: if `Bool` is declares to be a member of `Show` and `show` is defined for booleans as
```
show False = "False"
show True  = "True"
? putStr(show True)
True
```
- Some instances of `Show` are provided as primitive.
```
? putStr("The year is "++ show(3*667))
The year is 2001
```