

CSE 3302 Programming Languages




Functional Programming Language: Haskell

Chengkai Li
Fall 2007

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 1


Topics



- Basics
- Types and classes
- Defining functions
- List comprehensions
- Recursive functions
- Higher-order functions

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 2


Notes



- Haskell is not free-format. It has certain layout rules.
- General rules to follow:
 - Indent the same amount for expressions at the same level
 - Don't use tab
- You need to use script (module) files to define functions, and use interactive environment to evaluate expressions (functions)


Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 3

Basics



Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 4

The Standard Prelude




When Hugs is started it first loads the library file `Prelude.hs`, and then repeatedly prompts the user for an expression to be evaluated.

For example:

```
> 2+3*4
14
> (2+3)*4
20
```

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 5

The Standard Prelude



The standard prelude also provides many useful functions that operate on lists. For example:

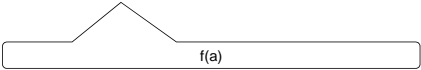
```
> length [1,2,3,4]
4
> product [1,2,3,4]
24
> take 3 [1,2,3,4,5]
[1,2,3]
```

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 6

Function Application

In Haskell, function application is denoted using space

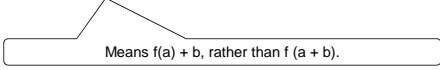
`f a`



Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 7

Moreover, function application is assumed to have higher priority than all other operators.

`f a + b`



Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 8

Examples

<u>Mathematics</u>	<u>Haskell</u>
<code>f(x)</code>	<code>f x</code>
<code>f(x,y)</code>	<code>f x y</code>
<code>f(g(x))</code>	<code>f (g x)</code>
<code>f(x,g(y))</code>	<code>f x (g y)</code>

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 9

Script

When developing a Haskell script, it is useful to keep two windows open, one running an editor for the script, and the other running Hugs.

Start an editor, type in the following two function definitions, and save the script as Test.hs: (file name, without `.hs`, must match module name)

```

module Test
  where

double x    = x + x

quadruple x = double (double x)

```

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 10

Leaving the editor open, load the script:

`:load Test.hs`

File Test.hs must be in the right path. Use "File >> Options" to change the path

Now both Prelude.hs and Test.hs are loaded, and functions from both scripts can be used:

```

> quadruple 10
40

> take (double 2) [1..6]
[1,2,3,4]


```

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 11

After a script is changed, it is automatically reloaded when you save the file. You can also use a reload command:

`> :reload`

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 12




Types and Classes

Lecture 18 – Functional Programming, Fall 2007

CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007

13



Types in Haskell


We use the notation $e :: T$ to mean that evaluating the expression e will produce a value of type T .

```
False      :: Bool
not        :: Bool -> Bool
not False  :: Bool
False && True :: Bool
```

Lecture 18 – Functional Programming, Fall 2007

CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007

14




Note:

- Every expression must have a valid type, which is calculated prior to evaluating the expression by type inference.
- Haskell programs are type safe, because type errors can never occur during evaluation;

Lecture 18 – Functional Programming, Fall 2007

CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007

15



Basic Types


Haskell has a number of basic types, including:

- Bool** - Logical values
- Char** - Single characters
- String** - Strings of characters
- Int** - integers

Lecture 18 – Functional Programming, Fall 2007

CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007

16



List Types

A list is sequence of values of the same type:

```
[False, True, False] :: [Bool]
['a', 'b', 'c', 'd'] :: [Char]
[['a'], ['b', 'c']] :: [[Char]]
```


In general:

$[T]$ is the type of lists with elements of type T .

Lecture 18 – Functional Programming, Fall 2007

CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007

17



Tuple Types

A tuple is a sequence of values of different types:

```
(False, True)      :: (Bool, Bool)
(False, 'a', True) :: (Bool, Char, Bool)
('a', (False, 'b')) :: (Char, (Bool, Char))
(True, ['a', 'b']) :: (Bool, [Char])
```

In general:

(T_1, T_2, \dots, T_n) is the type of n -tuples whose ith components have type T_i for any i in $1 \dots n$.

Lecture 18 – Functional Programming, Fall 2007

CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007

18

Function Types


A function is a mapping from values of one type to values of another type:

```
not      :: Bool → Bool
isDigit :: Char → Bool
(isDigit is in script Char.hs)
```

In general:

$T1 \rightarrow T2$ is the type of functions that map arguments of type $T1$ to results of type $T2$.

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 19

Note: 

⌘ The argument and result types are unrestricted. For example, functions with multiple arguments or results are possible using lists or tuples:

```
add      :: (Int,Int) → Int
add (x,y) = x+y

zeroto   :: Int → [Int]
zeroto n = [0..n]
```

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 20


Curried Functions

Functions with multiple arguments are also possible by returning functions as results:

```
add'     :: Int → (Int → Int)
add' x y = x+y
```

add' takes an integer x and returns a function. In turn, this function takes an integer y and returns the result x+y.

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 21

Note: 

⌘ add and add' produce the same final result, but add takes its two arguments at the same time, whereas add' takes them one at a time:

```
add      :: (Int,Int) → Int
add'     :: Int → (Int → Int)
```

⌘ Functions that take their arguments one at a time are called curried functions.

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 22

⌘ Functions with more than two arguments can be curried by returning nested functions:

```
mult     :: Int → (Int → (Int → Int))
mult x y z = x*y*z
```

mult takes an integer x and returns a function, which in turn takes an integer y and returns a function, which finally takes an integer z and returns the result x*y*z.

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 23

Curry Conventions


To avoid excess parentheses when using curried functions, two simple conventions are adopted:

- The arrow \rightarrow associates to the right.

```
Int → Int → Int → Int
```

Means $Int \rightarrow (Int \rightarrow (Int \rightarrow Int))$.

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 24



⌘ As a consequence, it is then natural for function application to associate to the left.


```
mult x y z
```

Means $((mult\ x)\ y)\ z$.

Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 25


Polymorphic Types



The function length calculates the length of any list, irrespective of the type of its elements.

```
> length [1,3,5,7]
4
> length ["Yes","No"]
2
> length [isDigit,isLower,isUpper]
3
```

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 26




This idea is made precise in the type for length by the inclusion of a type variable:

```
length :: [a] → Int
```

For any type a, length takes a list of values of type a and returns an integer.

A type with variables is called polymorphic.

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 27




Note:

⌘ Many of the functions defined in the standard prelude are polymorphic. For example:

```
fst :: (a,b) → a
head :: [a] → a
take :: Int → [a] → [a]
```

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 28

Overloaded Types




The arithmetic operator + calculates the sum of any two numbers of numeric type.

For example:

```
> 1+2
3
> 1.1 + 2.2
3.3
```

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 29



This idea is made precise in the type for + by the inclusion of a class constraint:

```
(+) :: Num a ⇒ a → a → a
```

For any type a in the class Num of numeric types, + takes two values of type a and returns another.

A type with constraints is called overloaded.

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 30

Classes in Haskell



A class is a collection of types that support certain operations, called the methods of the class.

Eq

Types whose values can be compared for equality and difference using

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

Lecture 18 – Functional Programming, Fall 2007

CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007

31

Haskell has a number of basic classes, including:



Eq - Equality types

Ord - Ordered types

Show - Showable types

Num - Numeric types

Lecture 18 – Functional Programming, Fall 2007

CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007

32

Example methods:



```
(==) :: Eq a => a -> a -> Bool
(<)  :: Ord a => a -> a -> Bool
show :: Show a => a -> String
(*)  :: Num a => a -> a -> a
```

Lecture 18 – Functional Programming, Fall 2007

CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007

33

Defining Functions



Lecture 18 – Functional Programming, Fall 2007

CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007

34

Conditional Expressions



As in most programming languages, functions can be defined using conditional expressions.

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

abs takes an integer n and returns n if it is non-negative and $-n$ otherwise.

Lecture 18 – Functional Programming, Fall 2007

CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007

35

Conditional expressions can be nested:



```
signum :: Int -> Int
signum n = if n < 0 then -1 else
           if n == 0 then 0 else 1
```

Note:

☞ In Haskell, conditional expressions must always have an else branch, which avoids any possible ambiguity problems with nested conditionals.

Lecture 18 – Functional Programming, Fall 2007

CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007

36

Guarded Equations

As an alternative to conditionals, functions can also be defined using guarded equations.

```
abs n | n >= 0 = n
      | otherwise = -n
```

As previously, but using guarded equations.

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 37

Guarded Equations

Guarded equations can be used to make definitions involving multiple conditions easier to read:

```
signum n | n < 0 = -1
         | n == 0 = 0
         | otherwise = 1
```

Note:

☞ The catch all condition otherwise is defined in the prelude by otherwise = True.

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 38

Pattern Matching

Many functions have a particularly clear definition using pattern matching on their arguments.

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

not maps False to True, and True to False.

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 39

Pattern Matching

Functions can often be defined in many different ways using pattern matching. For example

```
(&&) :: Bool → Bool → Bool
True && True = True
True && False = False
False && True = False
False && False = False
```

can be defined more compactly by

```
True && True = True      False && _ = False
_ && _ = False          True && b = b
```

☞ The underscore symbol _ is the wildcard pattern that matches any argument value.

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 40

List Patterns

In Haskell, every non-empty list is constructed by repeated use of an operator : called cons that adds a new element to the start of a list.

```
[1, 2, 3]
```

Means 1:(2:(3:[])).

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 41

List Patterns

The cons operator can also be used in patterns, in which case it deconstructs a non-empty list.

```
head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs
```

head and tail map any non-empty list to its first and remaining elements.

Lecture 18 – Functional Programming, Fall 2007 CSE3302 Programming Languages, UT-Arlington ©Chengkai Li, 2007 42

Lambda Expressions



A function can be constructed without giving it a name by using a lambda expression.

```
\x -> x+1
```

The nameless function that takes a number x and returns the result x+1.

Lecture 18 – Functional
Programming, Fall 2007

CSE3302 Programming Languages, UT-Arlington
©Chengkai Li, 2007

43

Why Are Lambda's Useful?



Lambda expressions can be used to give a formal meaning to functions defined using currying.

For example:

```
add x y = x+y
```

means

```
add = \x -> (\y -> x+y)
```

Lecture 18 – Functional
Programming, Fall 2007

CSE3302 Programming Languages, UT-Arlington
©Chengkai Li, 2007

44



Lambda expressions are also useful when defining functions that return functions as results.

For example,

```
compose f g x = f (g x)
```

is more naturally defined by

```
compose f g = \x -> f (g x)
```

Lecture 18 – Functional
Programming, Fall 2007

CSE3302 Programming Languages, UT-Arlington
©Chengkai Li, 2007

45