# CS571: Programming Languages



Haskell



## Functional Programming

- Functional programming emphasizes the application of functions, in contrast to imperative programming, which emphasizes changes in state and the execution of sequential commands.
- A functional language is a language that supports and encourages programming in a functional style.



## Encoding "factor" Using Haskell

$$factor(n) = \begin{cases} 1 & if \ \ n = 0 \\ n * factor(n-1) & otherwise \end{cases}$$

```
fact x =
    if x == 0 then 1 else x * fact(x-1);
```



## Why Functional Programming

- Shorter, clearer, and more maintainable code.
- Easier to understand



## Haskell in Industry

- http://www.haskell.org/haskellwiki/Haskell_in_industry
- Haskell has a diverse range of use commercially
  - Aerospace and defense
  - Finance
  - Web startups
  - Hardware design firms
  - A lawnmower manufacturer

## Haskell in Industry

- AT & T: used in the Network Security division to automate processing of internet abuse complaints.
- Bank of America: used for backend data transformation and loading.
- Alcatel-Lucent: prototype narrowband software radio systems.
- Allston Trading: used Haskell for some of their trading infrastructures.
- ……



## Haskell

- Online tutorial
  http://haskell.org/tutorial/
  google "haskell tutorial"
- Software:
  - Hugs: a functional programming system based on Haskell.
  - Type hugs in bingsuns.binghamton.edu
  - Download hugs from
  - https://www.haskell.org/hugs/pages/downloading.htm
  - To exit hugs, type :q



## Software: Hugs

```
bingsun2% hugs
__   __ __  __  ____   ___
________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__|| __||      Copyright (c) 1994-2005
||---||         ___ ||          World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: March 2005
________________________________________

Haskell 98 mode: Restart with command line option -98 to enable
   extensions

Type :? for help
Hugs.Base>
```



## Hugs

| | |
|---|---|
| :load <filename> | load a file |
| :edit <filename> | edit file |
| <expr> | evaluate expression |
| :type <expr> | print type of expression |
| :cd dir | change directory |
| :gc | force garbage collection |
| :version | print Hugs version |
| :quit | exit Hugs |



## Type System

- Type checking happens at compile time.
- Every value has a type
- Values can be passed as arguments to functions
- Functions are values



## Case Sensitivity

- Haskell is case-sensitive
  - Function names and constants start with a lower case letter
  - Specific types start with a capital letter.

## Basic Types and definitions: Bool

- The Booleans: Bool
  - Values: True, False
  - Operators

| | |
|---|---|
| && | and |
| \|\| | or |
| not | not |

- Simple computation can be executed interactively

```
Hugs> True && False
False

Hugs> True || False
True

Hugs> not True
False
```



## Basic Types and definitions: Int

- The integers: Int
  - Values: e.g. 0, 45, -5, 273873
  - Operators: +, -, *, ^ (raise to the power), div (whole number division), mod (the remainder from whole number division), abs (absolute value), negate (change the sign).
  - The greatest value in Int is 2147483647.
  - For larger number, we need use Integer.



## Basic Types and definitions: Int

```
Hugs> 2*3

Hugs> div 13 5

Hugs> mod 13 5

Hugs> 2^3

Hugs> abs (-2.3)

Hugs> negate (-2.3)
```



## Basic Types and definitions: Int

```
Hugs> 2*3
6
Hugs> div 13 5
2
Hugs> mod 13 5
3
Hugs> 2^3
8
Hugs> abs (-2.3)
2.3
Hugs> negate (-2.3)
2.3
```



## Infix and Prefix

- "div" and "mod" need backquotes (`) when used as "infix" operators
  - 12 `mod` 5 is the same as mod 12 5
- Infix operators can be written before their arguments, by enclosing the operator in parenthesis.

(+) 2 3 = 2 + 3

```
Hugs> (+) 2 3
5
```



## Basic Types and Definitions: Relational Operators

- Relational operators: take two integers as input and return a Bool (True or False)

>, >=, ==, /=, <=, <

```
Hugs> 3 >= 2
True

Hugs> 3 /= 2
True

Hugs> 3 <= 2
False
```

## Basic Types and Definitions: Char, String

- The characters: Char
  'a', '3', '\t' (tab), '\n' (new line), '\\' (backslash)
  '\' ' (single quote ' ), '\" ' (double quote " )
- String is a list of Chars.
  - type String = [Char]
  - "abc" is shorthand for ['a','b','c']



## Basic Types and Definitions: Float

- Floating-point numbers: Float
  - Value: 0.132, -23.3
  - Operators: e.g., +, -, *, / (fractional division), ^, abs, ceiling/floor (convert a fraction to an integer), pi (the constant), sqrt (positive square root).
  - Haskell also allows literal floating-point numbers in scientific notation.

| | |
|---|---|
| 231.61e7 | 231.61 * 10^7 = 2,316,100,000 |
| 231.6e-2 | 231.61 * 10^(-2) = 2.3161 |



## Basic Types and Definitions: Float

```
Hugs > 2/3

Hugs > ceiling 1.23

Hugs > floor 1.23

Hugs > pi

Hugs > sqrt 4
```



## Basic Types and Definitions: Float

```
Hugs > 2/3
0.666666666666667

Hugs > ceiling 1.23
2
Hugs > floor 1.23
1
Hugs > pi

Hugs > sqrt 4
```



## Basic Types and Definitions: Float

```
Hugs > 2/3
0.666666666666667

Hugs > ceiling 1.23
2
Hugs > floor 1.23
1
Hugs > pi
3.14159265358979

Hugs > sqrt 4
2.0
```



## Script

- Script: filename.hs
  - '- -' specifies the beginning of a comment. Comments can also be enclosed by symbols '{-' and '-}' (multiple lines).

square.hs:

```
---- The function to square an integer
square :: Int -> Int
square n = n*n
```

Has type

- To load square.hs, type :1oad square.hs or :l square.hs
- To edit square.hs, type :e square.hs. Use ctrl-X to exit.

## Type

- Haskell can infer the type

```
square n = n*n
> :t square
> square :: Num a => a → a
```

Num (numeric type): integers and floating points



## Functions

- Functional programming: writing a program as a set of functions.

```
cube x = x * (square x)
square x = x * x

> sqare 5
25

> cube 5
125
```

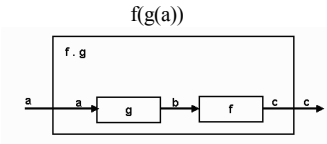


## Example: inc

- Increase an integer n by 1

```
inc :: Integer -> Integer
inc n = n+1

Main> inc (inc 3)
```



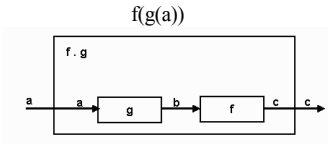



## Example: inc

- Increase an integer n by 1

```
inc :: Integer -> Integer
inc n = n+1

Main> inc (inc 3)
5
```






## Currying a Function

- Functions of two or more arguments
  - Uncurried form

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

Hugs> :load adduc.hs
Main> addUC(1,2)
3
```



## Currying a Function

- Functions of two or more arguments
  - Curried form -- the name curry derives from the person who popularized the idea: Haskell Curry
    - Take a single argument at a time

```
add :: Int -> Int -> Int
→is right associative, a → b → c means a → (b → c)

add x y = x+y
```

add x y is equivalent to (add x) y - applying add to yields a new function which is then applied to the second argument y

We can define inc using add:

## Currying a Function

- Functions of two or more arguments
  - Curried form -- the name curry derives from the person who popularized the idea: Haskell Curry
    - Take a single argument at a time

    add :: Int -> Int -> Int

    →is right associative, a → b → c means a → (b → c)

    add x y = x+y

    add x y is equivalent to (add x) y - applying add to yields a new function which is then applied to the second argument y

    We can define inc using add:

    inc = add 1



## Example (f.hs)

- Given the following haskell program

  **f x y = x + 2*y**

  What is the result of **f (f 1 2) (f 3 4)**?

  What is the result of **f (f 1 2) f 3 4**?



## Example (f.hs)

- Given the following haskell program

  **f x y = x + 2*y**

  What is the result of **f (f 1 2) (f 3 4)**?

  27

  What is the result of **f (f 1 2) f 3 4**?



## Example (f.hs)

- Given the following haskell program

  **f x y = x + 2*y**

  What is the result of **f (f 1 2) (f 3 4)**?

  27

  What is the result of **f (f 1 2) f 3 4**?

  ERROR - Cannot infer instance

  *** Instance : Num (a -> a -> a)

  *** Expression : f (f 1 2) f 3 4

  Reason: **f (f 1 2) f 3 4 = (f (f 1 2) f) 3 4**



## Excercise

- sumsquares(n,m) = $n^2 + m^2$



## Excercise

- sumsquares(n,m) = $n^2 + m^2$

  sq x = x*x

  sumsquares n m = sq n + sq m

## Excercise

- Write a haskell fuction isEven n that checks if n is an even number.



## Excercise

- Write a haskell fuction isEven n that checks if n is an even number.

```
isEven :: Int → Bool
isEven n = (n `mod` 2 ==0)
```



## Conditional Expressions

- We can write conditional expressions by means of the if…then…else constructor of Haskell.

```
max:: Int → Int → Int
max x y
        = if x >= y then x else y
```



## Guards/Conditions

- Used to give alternatives in the definitions of functions.
- A guard is a Boolean expression and these expressions are used to express various cases in the definition of a function.

```
max:: Int → Int → Int
max x y
        | x >= y        = x
        | otherwise     = y
```



## Example: maxThree

- Given three inputs, compute the maximum number.



## Example: maxThree

- Given three inputs, compute the maximum number.

```
maxThree:: Int → Int → Int → Int
    maxThree x y z
            | x >= y && x >= z  = x
            | y >= z            = y
            | otherwise         = z
```

## Factorial Revisit (fact.hs)

$$f : N_0 \rightarrow N_0$$

$$f(n) = \begin{cases} 1 & n = 0,1 \\ n * f(n-1) & otherwise \end{cases}$$

```
factorial :: Int → Int
factorial x
  | x < 0        = error "negative value"
  | x == 0       = 1
  | otherwise    = x * factorial (x-1)
```



## Factorial Revisit (fact.hs)

```
factorial :: Int → Int
factorial x
  | x < 0        = error "negative value"
  | x == 0       = 1
  | otherwise    = x * factorial (x-1)

Main> factorial (-1)

Main> factorial 4

Main> factorial 4.5
```



## Factorial Revisit (fact.hs)

```
factorial :: Int → Int
factorial x
  | x < 0        = error "negative value"
  | x == 0       = 1
  | otherwise    = x * factorial (x-1)

Main> factorial (-1)
Program error: negative value

Main> factorial 4

Main> factorial 4.5
```



## Factorial Revisit (fact.hs)

```
factorial :: Int → Int
factorial x
  | x < 0        = error "negative value"
  | x == 0       = 1
  | otherwise    = x * factorial (x-1)

Main> factorial (-1)
Program error: negative value

Main> factorial 4
24

Main> factorial 4.5
```



## Factorial Revisit (fact.hs)

```
factorial :: Int → Int
factorial x
  | x < 0        = error "negative value"
  | x == 0       = 1
  | otherwise    = x * factorial (x-1)

Main> factorial (-1)
Program error: negative value

Main> factorial 4
24

Main> factorial 4.5
ERROR - Cannot infer instance
*** Instance   : Fractional Int
*** Expression : factorial 4.5
```



## Factorial Revisit (fact.hs)

```
Main> factorial 15

Main> factorial 16

Main> factorial 20

Main> factorial 50
```

## Factorial Revisit (fact.hs)

Main> factorial 15
2004310016

Main> factorial 16

Main> factorial 20

Main> factorial 50



## Factorial Revisit (fact.hs)

Main> factorial 15
2004310016

Main> factorial 16
2004189184

Main> factorial 20

Main> factorial 50



## Factorial Revisit (fact.hs)

Main> factorial 15
2004310016

Main> factorial 16
2004189184

Main> factorial 20
-2102132736

Main> factorial 50



## Factorial Revisit (fact.hs)

Main> factorial 15
2004310016

Main> factorial 16
2004189184

Main> factorial 20
-2102132736

Main> factorial 50
0



## Factorial Revisit (fact.hs)

Main> factorial 15
2004310016

Main> factorial 16
2004189184

Main> factorial 20
-2102132736

Main> factorial 50
0

- The greatest number in Int is 2147483647. For large number, use Integer.



## Problems with Int

Change to factorial :: Int -> Int
factorial :: Integer -> Integer

Main> factorial 15
1307674368000

Main> factorial 16
20922789888000

Main> factorial 20
2432902008176640000

Main> factorial 50
30414093201713378043612608166064768844377641568960512000000000000

## Factorial Revisit (Cont.)

- If we do not specify the type of fact, then Haskell picks up the most general type

```
fact x
    | x < 0        = error "negative value"
    | x == 0       = 1
    | otherwise    = x * fact (x-1)
```

… > : t fact



## Factorial Revisit (Cont.)

- If we do not specify the type of fact, then Haskell picks up the most general type

```
fact x
    | x < 0        = error "negative value"
    | x == 0       = 1
    | otherwise    = x * fact (x-1)
```

… > : t fact

fact :: (Ord a, Num a) => a → a

The input must be in the intersection of classes Ord (characters, integers, floating points etc.) and Num.

…> fact 3.7



## Factorial Revisit (Cont.)

- If we do not specify the type of fact, then Haskell picks up the most general type

```
fact x
    | x < 0        = error "negative value"
    | x == 0       = 1
    | otherwise    = x * fact (x-1)
```

… > : t fact

fact :: (Ord a, Num a) => a → a

The input must be in the intersection of classes Ord (characters, integers, floating points etc.) and Num.

…> fact 3.7

Program error: negative value



## Pattern Notation

- You can also use patterns

```
factorial2 :: Integer -> Integer
factorial2 n | n < 0     = error "negative value"
factorial2 n | n == 0    = 1
factorial2 n | n > 0     = n * factorial2 (n-1)
```



## Precedence

- Function application has highest precedence:
- factorial n – 1 is (factorial n) – 1, not factorial (n-1)

Main> factorial 4 - 1

Main> factorial (4-1)



## Precedence

- Function application has highest precedence:
- factorial n – 1 is (factorial n) – 1, not factorial (n-1)

Main> factorial 4 - 1

23

Main> factorial (4-1)

## Precedence

- Function application has highest precedence:
- factorial n – 1 is (factorial n) – 1, not factorial (n-1)

```
Main> factorial 4 - 1
23
Main> factorial (4-1)
6
```



## Negative Numbers

- -12: prefix '-' can often get confused with the infix operator to subtract one number from another.
- Enclose the negative number in parentheses when it is used as a function argument:

```
> sq (-2)

> sq -2
```



## Negative Numbers

- -12: prefix '-' can often get confused with the infix operator to subtract one number from another.
- Enclose the negative number in parentheses when it is used as a function argument:

```
> sq (-2)
4
> sq -2
```



## Negative Numbers

- -12: prefix '-' can often get confused with the infix operator to subtract one number from another.
- Enclose the negative number in parentheses when it is used as a function argument:

```
> sq (-2)
4
> sq. -2
    ERROR - Cannot infer instance
    *** Instance   : Num (Int -> Int)
    *** Expression : square - 2
```



## Exercise

- Write a function iszero that returns true when its integer argument is 0 and false other wise
- Write a haskell function power2 n to compute $2^n$ **n >= 0** (use recursion)



## Exercise

- Write a function iszero that returns true when its integer argument is 0 and false other wise

```
iszero x
    |x==0 = True
    |otherwise = False
```

- Write a haskell function power2 n to compute $2^n$ **n >= 0** (use recursion)

## Exercise

- Write a function iszero that returns true when its integer argument is 0 and false other wise

```
iszero x
    |x==0 = True
    |otherwise = False
```

- Write a haskell function power2 n to compute $2^n$ **n >= 0** (use recursion)

```
power2 :: Int -> Int
power2 n
  | n == 0   = 1
  | n > 0    = 2 * power2 (n-1)
```



## Excercise

- Write a haskell function threeEqual that checks if three Integers are equal.



## Excercise

- Write a haskell function threeEqual that checks if three Integers are equal.

```
threeEqual :: Int → Int → Int → Bool
threeEqual x y z
                | x == y && x == z   = True
                | otherwise          = False
```

*Or*

```
threeEqual x y z = (x==y) && (y==z)
```



## Excercise

- Write a haskell function addall such that addall(n) = 1 + 2 + … + n (assume that n >= 1)



## Excercise

- Write a haskell function addall such that addall(n) = 1 + 2 + … + n (assume that n >= 1)

if n = 1 → addall(n) = 1
if n > 1 → addall(n) = n + addall(n-1)

```
addall n
    | n == 1        = 1
    | otherwise     = n + addall (n-1)
```