Patterns, actions, tuples

- Layout
- Recursion
- Patterns
- Sale applications
- `fromIntegral`
- Output and actions
- Type synonyms
- Tuples

Notes, Ch 19, 20, 22, 23

Layout

The **layout** of Haskell scripts is used to tell where a function ends and another begins.

off-side rule

```
bigger m n
  |m >= n      = m
  |otherwise   = n

threeTimes n = 3*n
```

Valid syntax:

answer = 20; done = False

Pattern matching

Function to return `True` when both arguments have the same value and `False` otherwise.

```
eqOp ::Bool -> Bool -> Bool
```

```
eqOp True True  = True
eqOp False False = True
eqOp True False  = False
eqOp False True = False
```

Wildcard

```
isZero :: Int -> Bool
isZero 0 = True
isZero _ = False
```

Patterns

A function can be defined in many ways

```
eqOp :: Bool -> Bool -> Bool
```

can be defined more compactly by:

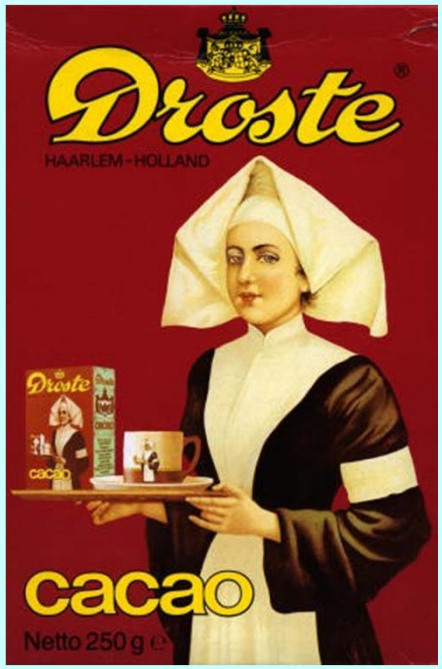
```
eqOp True True     = True
eqOp False False   = True
eqOp  _ _          = False
```

Patterns

A pattern can be:

A literal	`0  'a'  True  False`
A variable	any argument value will match this
Wild card	any argument value will match this

Recursion



Recursive functions

Functions defined in terms of themselves are called **recursive**

```
6!= 6x5x4x3x2x1=720
          0!   = 1
          n!   = n * (n-1)!
```

Assume we are working with non negative integers

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

base case

recursive case

Recursive functions

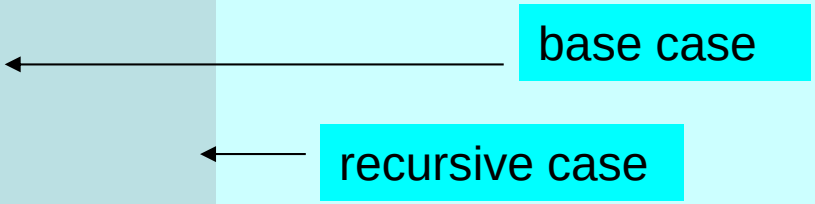
```
fact  :: Int -> Int
fact n
   |n < 0     = error "Negative input—not allowed"
   |n == 0     = 1
   |otherwise = n * fact (n-1)
```

Pattern matching

The factorial function using patterns:

--assume positive integers input

```
factP  :: Int -> Int
factP 0 = 1
factP n = n * factP (n-1)
```



Guards & pattern matching

```
factP  :: Int -> Int
factP n|n <= 0 = 1
factP n        = n * factP (n-1)
```

```
factP  :: Int -> Int
factP n|n<0 = error "Negative input—not allowed"
factP 0     = 1
factP n     = n * factP (n-1)
```

Sale application

Write a Haskell program to display sales data for a week.

```
Prelude> :load "saleAPP.hs"
[1 of 1] Compiling Main             ( saleAPP.hs, interpreted )
Ok, modules loaded: Main.
*Main> printTable  6
    Week     Sales
    0        15
    1        5
    2        7
    3        18
    4        7
    5        0
    6        5
    Total =  57
    Mean=    8.142858
*Main>
```

Sale application

```
{-author ID 91919191, Calude E.
version 21, 159.202--Demo sale application-see Notes, Ch 20
type at prompt:  printTable 4, i.e printTable and an Int
value <= maxWeekVal  -}
maxWeekVal::Int
maxWeekVal=6

isNotValidWeek::Int ->Bool
isNotValidWeek n = (n<0 ||n>maxWeekVal)
sales :: Int -> Int
sales 0 = 15
sales 1 =  5
sales 2 =  7
sales 3 =  18
sales 4 =  7
sales 5 =  0
sales 6 =  5
```

Sale application

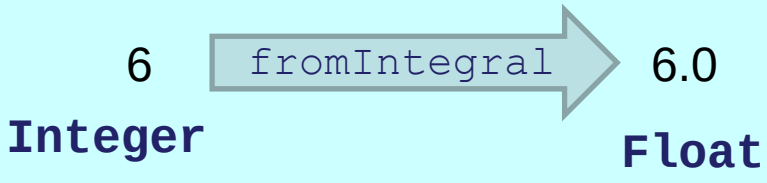
```
--function to compute total sales; uses sales function
totalSales :: Int -> Int
totalSales 0   = sales 0
totalSales n   = totalSales(n-1) + sales n


-------------------------------------------------
--function to compute maximum sale up to week n;
--uses max (from Prelude) and sales functions
maxSales :: Int -> Int
maxSales 0 = sales 0
maxSales n = max (sales n) (maxSales (n-1))
```

Sale application

```
--function to compute the mean of the sales up to week n;
--uses totalSales and fromIntegral (fromPrelude)functions
meanSales :: Int -> Float
meanSales n = (totalSales n)/(n+1))                    WRONG!
```

How to let the computer know we want to perform division with real numbers?

```
meanSales n =fromIntegral (totalSales n) /fromIntegral (n+1)
```



Integer Type

`fromIntegral` is defined for `Integer` values

```
--function to compute the mean of the sales up to week n;
--uses totalSales and fromIntegral functions
meanSales :: Integer -> Float
meanSales n = fromIntegral (totalSales n) / fromIntegral (n+1)
```

`Integer` type is used for very large integer values

- Operations same as for `Int (+,-, div, mod, ^)`
- Relations same as for `Int (==,/=,>=,<=,<,>)`
- Functions same as for `Int (gcd, lcm, odd, even, max,min)`

Float Type

`Float:` single precision
`Double:` double precision

Operations: `+,-,/ * ^`
Functions: `truncate, min, max, abs`

```
val1::Float
val1=12345678912345678945.12345678912345678 9
val2::Double
val2=12345678912345678945.12345678912345678 9
```

ceiling, floor, round

```
*Main> val1
1.2345679e19
*Main> val2
1.234567891234568e19
*Main>
```

Strings

```
{- The heading of the table -}

heading :: String
heading = "\tWeek" ++  "\t\tSales" ++ "\n"
```

What is a String? `[Char]`
What can we do with `String`s? `++`

Sometimes we need to combine strings with numbers:

`Total =  45`

The function **`show`** performs the conversion:

```
"Total =" ++ show 45
```

Sale application

```
-- A recursive function for printing the first
--n rows in the table; uses printWeek function
printUpTo :: Integer -> String
printUpTo 0 = printWeek 0
printUpTo n = printUpTo (n - 1) ++ printWeek n
-----------------------------------------
-- print each week's data;
-- uses show(from Prelude) and sales functions
printWeek :: Integer -> String
printWeek n = "\t" ++ (show n) ++ " \t\t" ++ (show (sales n)) ++ "\n"
-------------------------------------------
--function for printing the total
--uses show (from Prelude) and totalSales functions
printTotal :: Integer -> String
printTotal n = "\tTotal =" ++ "\t" ++ (show (totalSales n)) ++ "\n"
---------------------------------------------
--prints the mean of all sales up to week n
--uses show (Prelude) and meanSales functions
printMean :: Integer -> String
printMean n = "\t"++ "Mean =" ++ "\t"++ (show (meanSales n))++ "\n"
```

Performing output

```
{--- function to display the table of up to n-th week
sales uses putStr (from Prelude), printUpTo,
printMean  and printTotal functions and the value
heading --------}
printTable :: Integer -> IO( )
printTable n|isNotValidWeek n =
            error ("Week number must be from 0 to "
                  ++ show maxWeekVal)
printTable n = putStr (heading ++ printUpTo n ++
                       printTotal n ++ printMean n)
```

Actions

Performing input/output and pure functions.

What are the arguments and the results when
a) performing input from the keyboard?
b) sending output to the screen?

IO action types:

`IO a` an input /output is performed and a value of type `a` is returned

`IO()` an input/output is performed and no meaningful value is returned

```
putStr ::String -> IO ()
putStrLn ::String -> IO ()
```

Types

```
True :: Bool
not :: Bool → Bool
('a',65) :: (Char, Int)
[99.6,89.8,100.00]::[Float]
```

e::T -- evaluating expression `e` will produce a value of type `T`.

```
myNumber ::Integer
myNumber = 1234562149
```

Bool

Operator	Precedence	Description
&&	3	Logical AND
||	2	Logical OR
not	9	Logical NOT

Examples:

```
a) 8 < 6 && 4 > 2
b) not False || True && False
```

Char

Characters:

‘a’ to ‘z’, ‘A’ to ‘Z’, ‘0’ to ‘9’

Also characters:

tab ‘\t’,
newline ‘\n’,
backslash ‘\\’
single quote ‘\’’,
double quote ‘\”’

The **ASCII code** of the characters can be used for representing them.

Char

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(nul)	0	0000	0x00	(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
(soh)	1	0001	0x01	!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
(stx)	2	0002	0x02	"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
(etx)	3	0003	0x03	#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
(eot)	4	0004	0x04	$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
(enq)	5	0005	0x05	%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
(ack)	6	0006	0x06	&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
(bel)	7	0007	0x07	'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(bs)	8	0010	0x08	(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
(ht)	9	0011	0x09	)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
(nl)	10	0012	0x0a	*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
(vt)	11	0013	0x0b	+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
(np)	12	0014	0x0c	,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
(cr)	13	0015	0x0d	-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
(so)	14	0016	0x0e	.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
(si)	15	0017	0x0f	/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
(dle)	16	0020	0x10	0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
(dc1)	17	0021	0x11	1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
(dc2)	18	0022	0x12	2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
(dc3)	19	0023	0x13	3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
(dc4)	20	0024	0x14	4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
(nak)	21	0025	0x15	5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
(syn)	22	0026	0x16	6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
(etb)	23	0027	0x17	7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
(can)	24	0030	0x18	8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
(em)	25	0031	0x19	9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
(sub)	26	0032	0x1a	:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
(esc)	27	0033	0x1b	;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
(fs)	28	0034	0x1c	<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
(gs)	29	0035	0x1d	=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
(rs)	30	0036	0x1e	>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
(us)	31	0037	0x1f	?	63	0077	0x3f	_	95	0137	0x5f	(del)	127	0177	0x7f

Char

The standard functions are in the module `Data.Char`
(`import Data.Char` or `Prelude> :m Data.Char`)

Function Name	Type
ord	`Char →Int`
chr	`Int →Char`
toUpper, toLower	`Char →Char`
isAscii, isDigit, isUpper, isLower	`Char →Bool`

Char example

```
--program 22.2 ,  Notes page 104
import Data.Char
caseDiff ::Int
caseDiff =ord 'A'-ord 'a'

capitalize::Char->Char
capitalize ch =chr(ord ch +caseDiff)

digitChar::Char ->Bool
digitChar ch =('0'<=ch) &&(ch <= '9')
```

String

A **string** is a special list consisting of characters.

```
type String  = [Char]
```

Examples:

```
"Haskell is a fun to learn."
"\72el\108o world"
"Haskell " ++ " programming“
```

++ is the **concatenation** operator.

Functions:

```
length "hello"
reverse "hello"
```

String

```
Prelude>putStr “Massey University”
Massey  University

Prelude> putStr “\99u\116”
cut

Prelude>putStr “oranges\napples\npears”
oranges
apples
pears
```

Tuples

A tuple is a collection of data items considered as a single entity. These data items (components) can be of different types.

("George", 25)
("John", 87)
("Andrew", 55)

(String, Int)

("Sugarl", 25, 0.95)
("Flour", 35, 1.25)
("Gin", 15, 20)
("Bread", 45, 2.6)

(String, Int, Float)

fst and snd

Functions **for pairs**:

```
Prelude> :t fst
fst :: (a,b) -> a

Prelude> :t snd
snd :: (a,b) -> b

Prelude> fst ( "Star", "Craft")
"Star"
Prelude> snd ("Star", "Craft")
"Craft"
```

Relations for tuples:

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

Examples:
(9,4,8) > (9,3,15)
(9,14,8) /= (9,8,14)
```

Type synonyms

We can give names to types

```
type String =[Char] –-standard Haskell
```

```
type  Person = (String, Int)
```

and we can define functions using our new types:

```
age ::Person -> Int
age x = snd x

("George", 25)::Person

type   ShopItem = (String, Int, Float)
price :: ShopItem -> Float
price (_,_, x) = x
```

Practice

a) Define a function which given two integer numbers, return a pair with the smaller number first

Examples: sort 10 20 = (10, 20)

sort 15 5 = (5, 15)

b) Write a function that calculates the area of a sphere, given the sphere's radius (`A(R)= 4πR`^{2}).

c) Define a function to compute Fibonacci numbers.

(see https://en.wikipedia.org/wiki/Fibonacci_number)

d) Define the **Ackermann function**

(see https://en.wikipedia.org/wiki/Ackermann_function)