Haskell Overview I (1A)

Young Won Lim
8/6/16

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

Based on

Haskell Tutorial, Medak & Navratil
ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

Yet Another Haskell Tutorial, Daume
https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf

Function Declaration and Definition

```
increment :: Int -> Int
```

function name arg type return type

Int increment (Int)

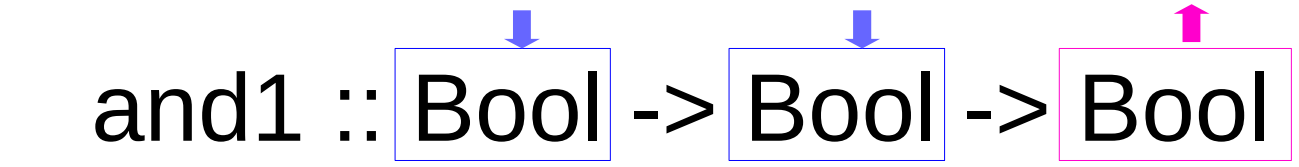
```
increment    x  =  x + 1
```

Int arg Int return

```
increment      2
```

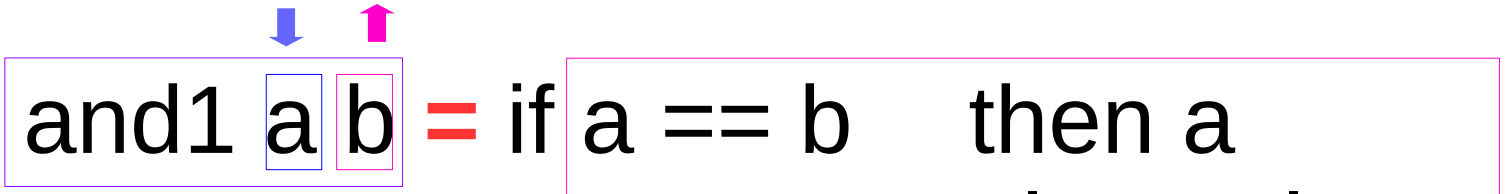
↳ 3 :: Int

Function Examples



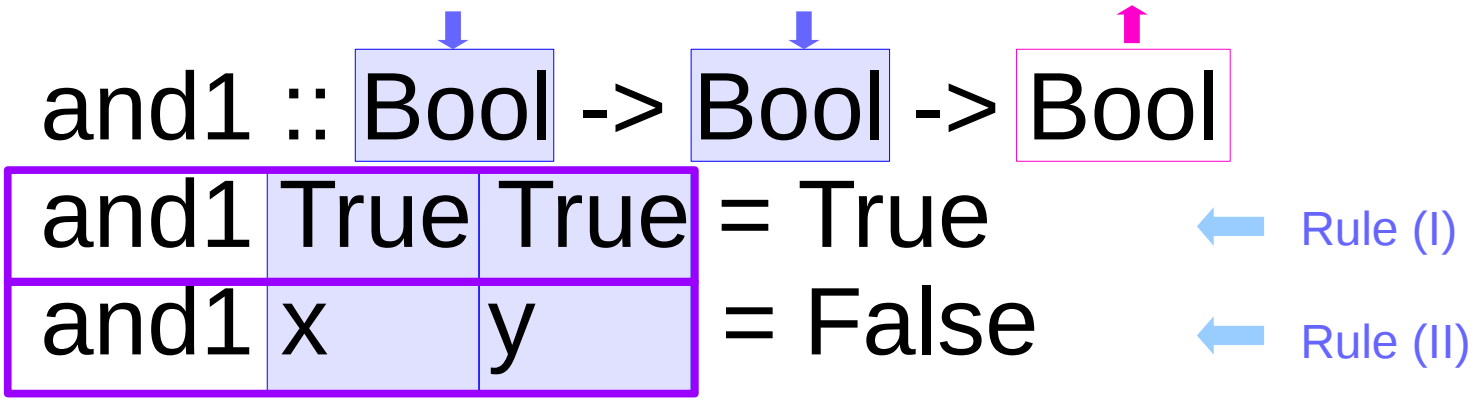
function name arg1 type arg2 type return type

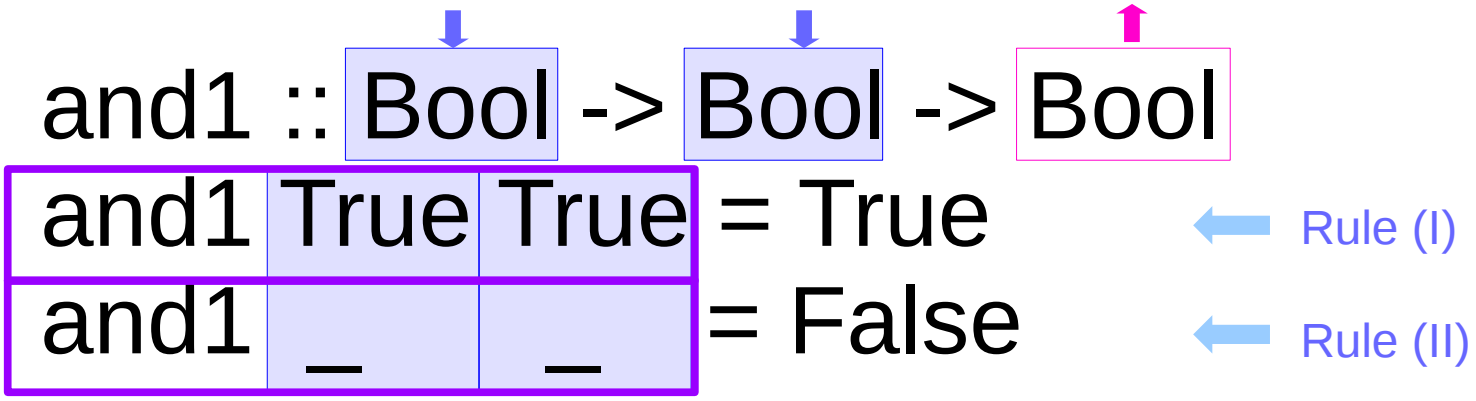
Bool and1 (Bool, Bool)



Bool arg1, arg2 Bool return

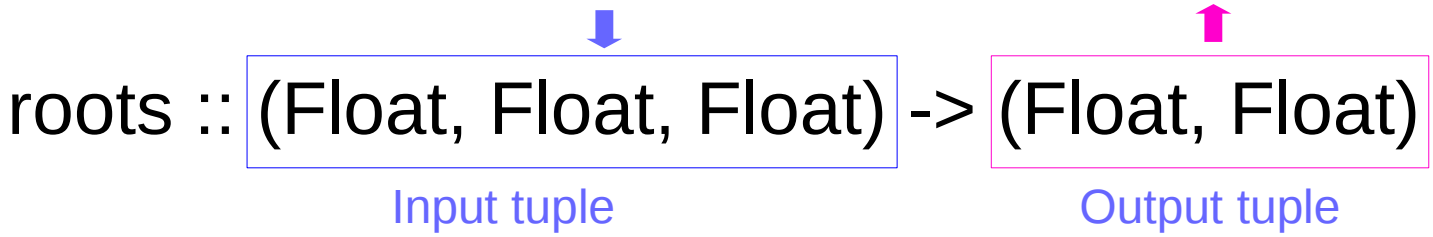
Several Function Definitions





wild character

Where Statement – Local Definition




```
roots   (a, b, c)    = (x1, x2)   where
    x1 = e + sqrt d / (2*a)
    x2 = e – sqrt d / (2*a)
    d = b*b – 4*a*c
    e = – b / (2*a)
```

the same indentation

without using { } as in C

No tab

Function Call Results

```
p1, p2 :: (Float, Float, Float)
p1 = (1.0, 2.0, 1.0)
p2 = (1.0, 1.0, 1.0)
```

Variable declaration

p1 assignment

p2 assignment

```
? roots p1 (-1.0,-1.0) :: (Float, Float)
(94 reductions, 159 cells)

? roots p2
( Program error: {primSqrtFloat (-3.0)}
(61 reductions,183 cells)
```

If-then-else Statement

```
roots :: (Float, Float, Float) -> (Float, Float)
```

Input tuple

Output tuple

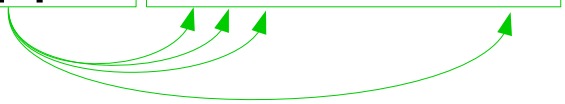
```
roots    (a, b, c)     =    do
    if d < 0
        then error “sorry”
        else (x1, x2)
    where
        x1 = e + sqrt d / (2*a)
        x2 = e – sqrt d / (2*a)
        d = b*b – 4*a*c
        e = – b / (2*a)
```

Indentation

No Tab

Simple List Functions

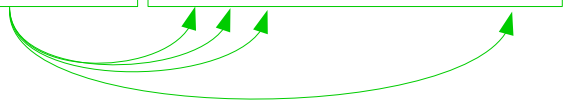
```
map Data.Char.toUpper "Hello World"
```



```
"HELLO WORLD"
```

the same number of list elements

```
filter Data.Char.isLower "Hello World"
```



```
"elloorld"
```

fewer number of list elements

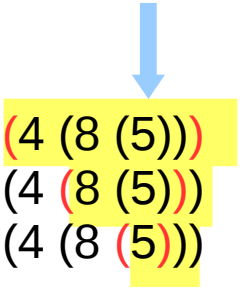
Simple List Functions – foldr function

foldr (-) 1 [4,8,5]
0

fold – Reduction operator
r - right hand side first

(4 (8 (5)))

foldr (-) 1 [4,8,5]
4 - (foldr (-) 1 [8,5])
4 - (8 - foldr (-) 1 [5])
4 - (8 - (5 - foldr (-) 1 []))
4 - (8 - (5 - 1))
4 - (8 - 4)
4 - 4
0



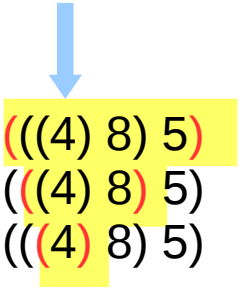
Simple List Functions – foldl function

foldl (-) 1 [4,8,5]
-16

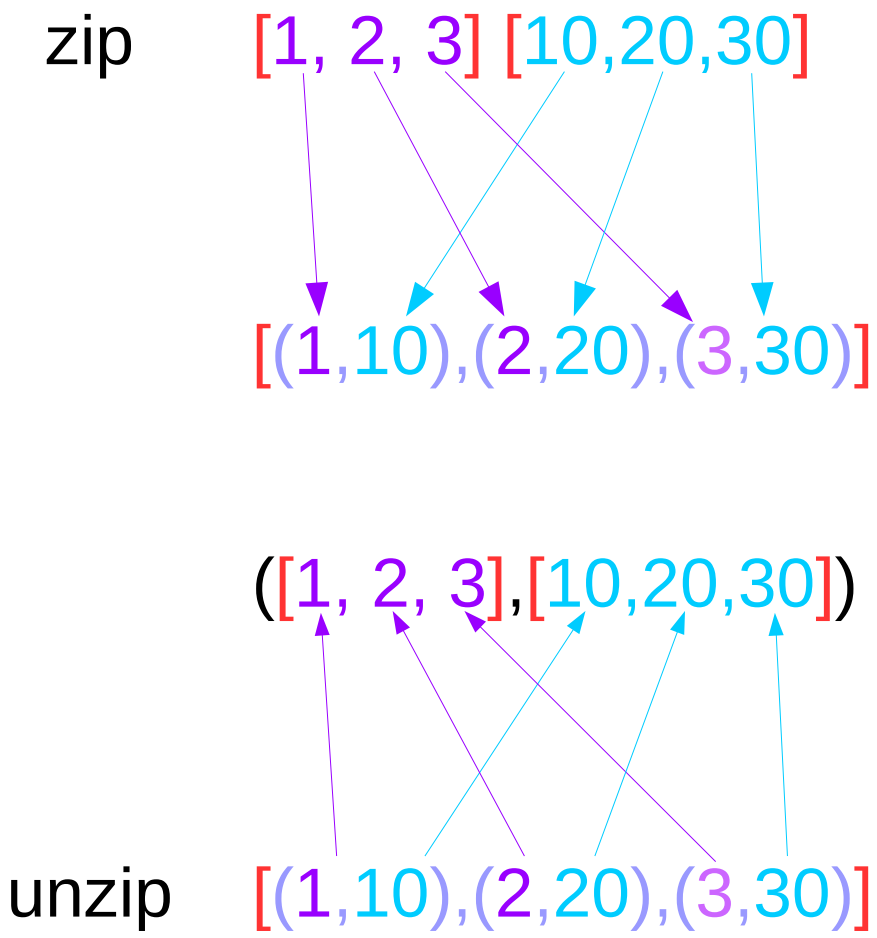
fold – Reduction operator
l - left hand side first

(((4) 8) 5)

foldl (-) 1 [4,8,5]
(foldl (-) 1 [4,8]) - 5
((foldl (-) 1 [4]) - 8) – 5
((1 – 4) – 8) – 5
((-3) – 8) – 5
-11 -5
-16



Simple List Functions – zip and unzip functions



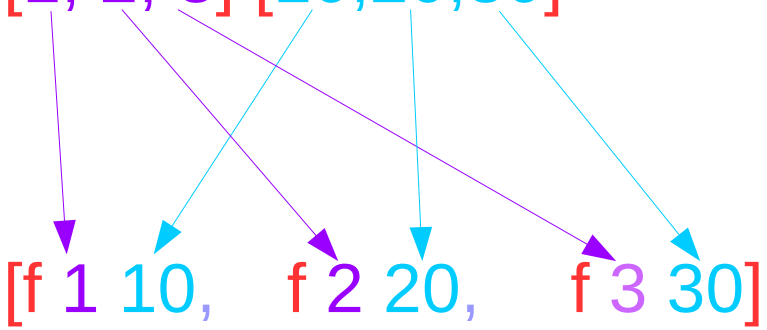
Simple List Functions – zip and unzip functions

```
zip "hello" [1,2,3,4,5]
[('h',1),('e',2),('l',3),('l',4),('o',5)]

unzip [('h',1),('e',2),('l',3),('l',4),('o',5)]
("hello",  [1,2,3,4,5])
```

Simple List Functions – zipWith function

zipWith f [1, 2, 3] [10,20,30]



zipWith (*) [1, 2, 3] [10,20,30]

[(*) 1 10, (*) 2 20, (*) 3 30]
[1 * 10, 2 * 20, 3 * 30]
[10, 40, 90]

Simple List Functions – myZipWith function

```
myZipWith :: (a -> a -> a) -> [a] -> [a] -> [a]

myZipWith    _   []     _      =  []
myZipWith    _   _      []     =  []
myZipWith    f   (a:as) (b:bs) = (f a b) : myZipWith f as bs
```

The n-th element in a list

```
nthListEl L 1 = head L
nthListEl L n = nthListEl (tail L) (n-1)
```

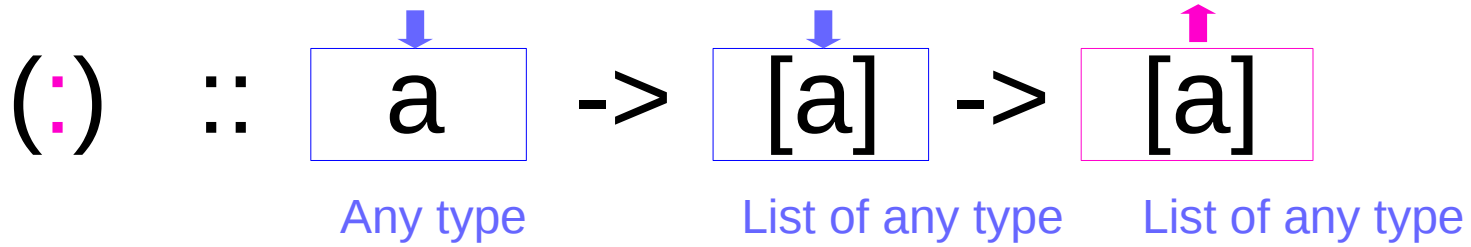
```
*Main> nthListE1 [1, 2, 3] 3
3
*Main> nthListE1 [1, 2, 3] 2
2
*Main> nthListE1 [1, 2, 3] 1
1
*Main> nthListE1 [1, 2, 3, 4, 5, 6] 5
5
*Main> nthListE1 [11, 22, 33, 44, 55, 66] 5
55
```

Recursion - factorial

```
fact 0 = 1
fact n = n * fact (n-1)
```

```
6 * fact(5)
      5 * fact(4)
            4 * fact(3)
                  3 * fact(2)
                        2 * fact(1)
                              1 * fact(0)
                              1 * 1
                        2 * 1
                  3 * 2 * 1
            4 * 3 * 2 * 1
      5 * 4 * 3 * 2 * 1
6 * 5 * 4 * 3 * 2 * 1
```

Construct



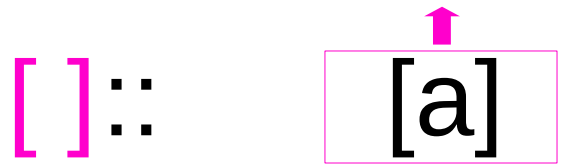
[1, 2, 3]

1 : 2 : 3 : [ ]

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

cons 1 (cons 2 (cons 3 Nil)))

takes an element and combines it with the rest of the list



the empty list takes zero arguments

Functions using the list constructor

```
sumList :: [Int] -> Int
sumList []    = 0
sumList (x:xs) = x + sumList xs
```

```
NthListEl' (L, Ls) 1   = L
NthListEl' (L, Ls) n   = NthListEl' Ls (n-1)
```

Map Function using the list constructor

```
map :: (a -> b) -> [a] -> [b]
```

Function param: a type, return: b type

List of a type

List of b type

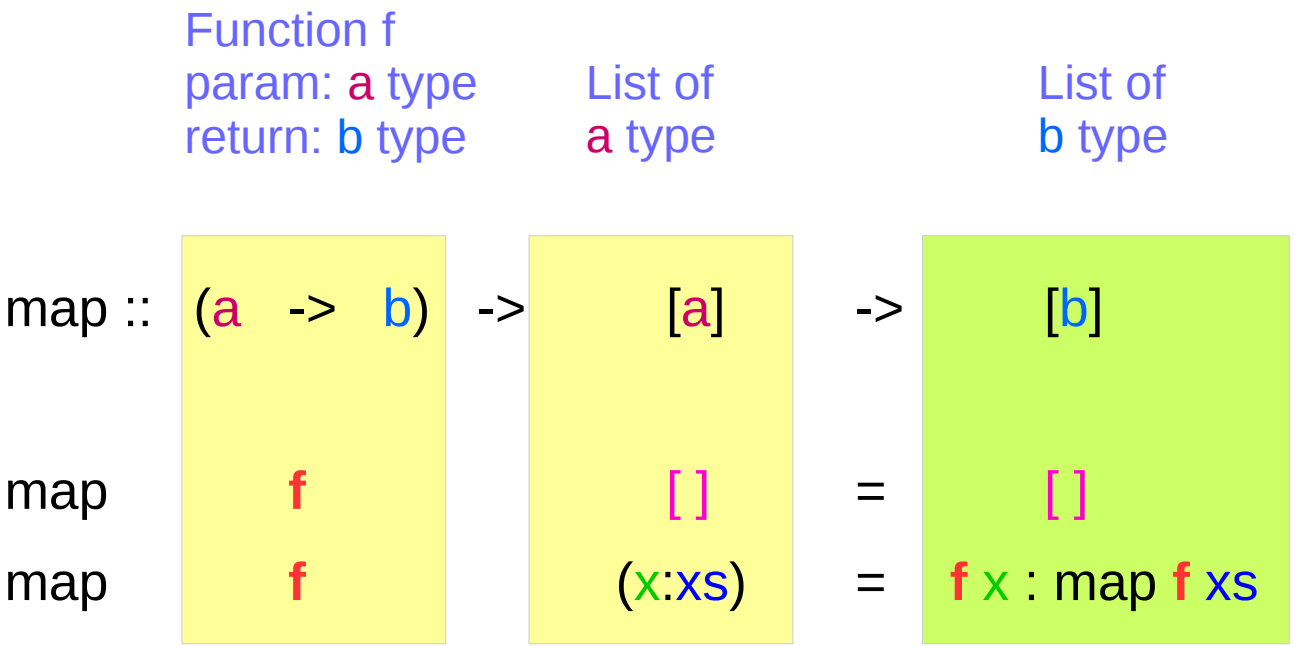
```
map f []     = []
map f (x:xs) = f x : map f xs
```

```
code :: String -> [Int]
code x = map ord x
```

code “Hugs”

[72, 117, 103, 115)

Function as an argument (1)



Function as an argument (2)

```
myfilter :: (a -> Bool)  ->    [a]      ->  [a]
myfilter       f         =    [ ]      =   [ ]
myfilter       f         =    (x:xs)   =   if f  x then  x : myfilter f xs
                                             else myfilter f xs
```

Solving a list of quadratic equations

```
roots :: (Float, Float, Float) -> (Float, Float)
roots   (a,b,c)                  = if d < 0 then error "sorry" else (x1, x2)
     where   x1 = e + sqrt d / (2 * a)
             x2 = e - sqrt d / (2 * a)
             d = b * b - 4 * a * c
             e = - b / (2 * a)

real ::   (Float, Float, Float) -> Bool
real      (a,b,c)                = (b*b - 4*a*c) >= 0

p1 = (1.0,2.0,1.0) :: (Float, Float, Float)
p2 = (1.0,1.0,1.0) :: (Float, Float, Float)
ps = [p1,p2]
newPs       = filter real ps
rootsOfPs   = map roots newPs
```

Tuple Functions

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

Extract the **first** component of a pair.

```
snd :: (a, b) -> b
```

Extract the **second** component of a pair.

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

curry converts an uncurried function to a **curried** function.

```
uncurry :: (a -> b -> c) -> (a, b) -> c
```

uncurry converts a curried function to **a function on pairs**.

```
swap :: (a, b) -> (b, a)
```

Swap the components of a pair.

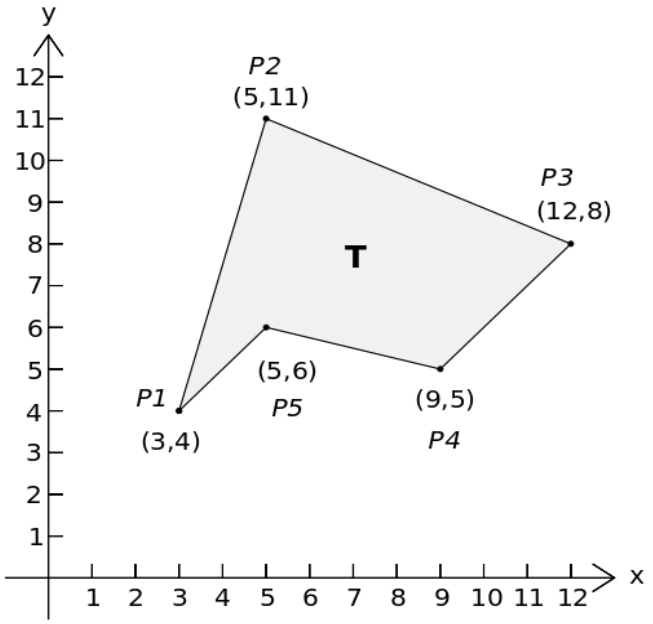
Gauss Area Formula

$$\mathbf{A} = \frac{1}{2}\left|\sum_{i=1}^{n-1} x_i y_{i+1} + x_n y_1 - \sum_{i=1}^{n-1} x_{i+1} y_i - x_1 y_n\right|$$

$$= \frac{1}{2}\left|x_1 y_2 + x_2 y_3 + \cdots + x_{n-1} y_n + x_n y_1 - x_2 y_1 - x_3 y_2 - \cdots - x_n y_{n-1} - x_1 y_n\right|$$

$$\mathbf{A} = \frac{1}{2}|3 \times 11 + 5 \times 8 + 12 \times 5 + 9 \times 6 + 5 \times 4$$
$$- 4 \times 5 - 11 \times 12 - 8 \times 9 - 5 \times 5 - 6 \times 3|$$

$$= \frac{60}{2} = 30$$



https://en.wikipedia.org/wiki/Shoelace_formula

Shoelace Area Formula

$$\mathbf{A} = \frac{1}{2}\left|\sum_{i=1}^{n-1} x_i y_{i+1} + x_n y_1 - \sum_{i=1}^{n-1} x_{i+1} y_i - x_1 y_n\right|$$

$$= \frac{1}{2}\left|x_1 y_2 + x_2 y_3 + \cdots + x_{n-1} y_n + x_n y_1 - x_2 y_1 - x_3 y_2 - \cdots - x_n y_{n-1} - x_1 y_n\right|$$

Shoelace Area Formula

$$\frac{1}{2}\left|\begin{matrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ \vdots & \vdots \\ x_n & y_n \\ x_1 & y_1 \end{matrix} - \begin{matrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ \vdots & \vdots \\ x_n & y_n \\ x_1 & y_1 \end{matrix}\right|$$

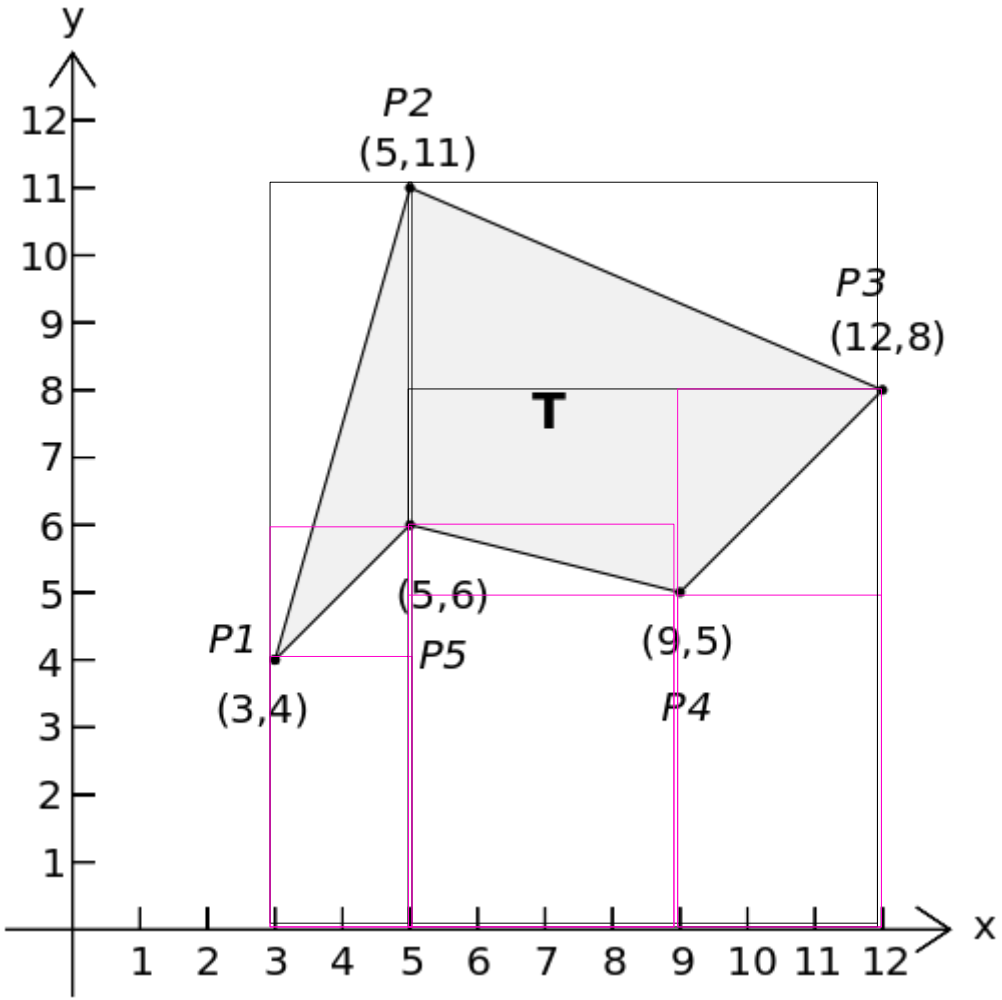
Another Form

$$2A = \sum_{i=1}^{n} (x_i - x_{i+1})(y_i + y_{i+1})$$

$$= \sum_{i=1}^{n} (x_i y_i - x_{i+1} y_{i+1} - x_{i+1} y_i + x_i y_{i+1})$$

https://en.wikipedia.org/wiki/Shoelace_formula

Shoelace Area Formula



$$2A = \sum_{i=1}^{n} (x_i - x_{i+1})(y_i + y_{i+1})$$

$$= \sum_{i=1}^{n} (x_i y_i - x_{i+1} y_{i+1} - x_{i+1} y_i + x_i y_{i+1})$$

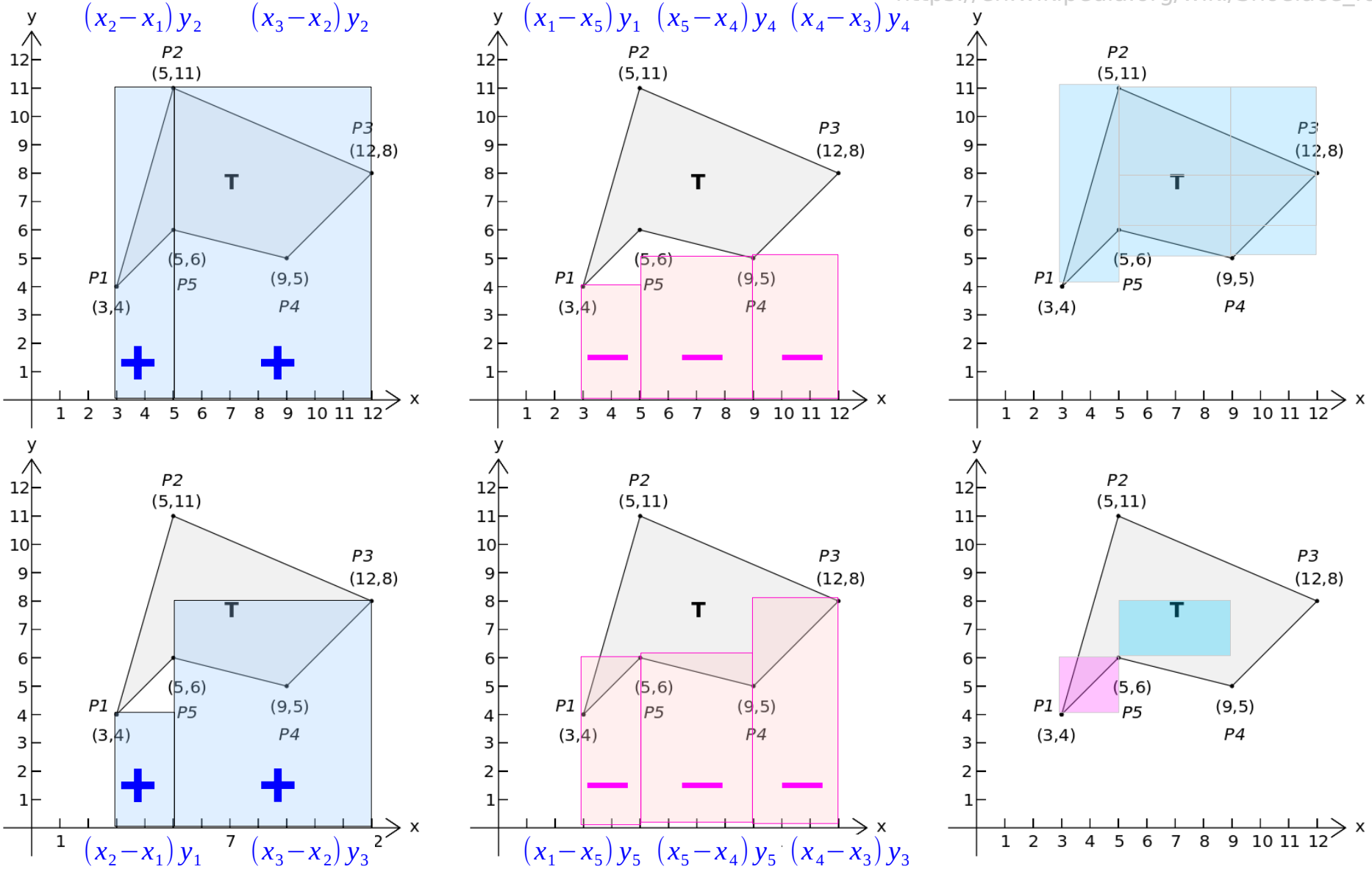
$$= \sum_{i=1}^{n} (x_i y_{i+1} - x_{i+1} y_i)$$

$$(x_1 y_1 + x_2 y_2 + \cdots + x_n y_n)$$
$$- (x_2 y_2 + \cdots + x_n y_n + x_1 y_1)$$

https://en.wikipedia.org/wiki/Shoelace_formula

Simple Verification of Gauss Area Formula

https://en.wikipedia.org/wiki/Shoelace_formula



Making Lists

```
p = [(100.0,100.0),(100.0,200.0),(200.0,200.0),(200.0,100.0)] :: [(Float,Float)]
```

p = [(x1, y1), (x2, y2), (x3, y3), (x4, y4)] :: [(Float, Float)]

list1 → [x1, x2, x3, x4]
list2 → [x2, x3, x4, x1]
list3 → [y1, y2, y3, y4]
list4 → [y2, y3, y4, y1]

```
list1 = map fst p
list2 = tail list1 ++ [head list1]
list3 = map snd p
list4 = tail list3 ++ [head list3]
```

ps

$$A = \frac{1}{2}\left|\sum_{i=1}^{n}(x_i - x_{i+1})(y_i + y_{i+1})\right|$$

Calculating area

list1 → [x1, x2, x3, x4]
list2 → [x2, x3, x4, x1]
list3 → [y1, y2, y3, y4]
list4 → [y2, y3, y4, y1]

zipWith (–) list1 list2 → [x1–x2, x2–x3, x3–x4, x4–x1]

zipWith (+) list3 list4 → [y1+y2, y2+y3, y3+y4, y4+y1]

```
zipWith (*)
      zipWith (–) list1 list2
      zipWith (+) list3 list4
```
→
```
[(x1–x2)(y1+y2),
 (x2–x3)(y2+y3),
 (x3–x4)(y3+y4),
 (x4–x1)(y4+y1)]
```

```
foldl (+) 0
      zipWith (*)
            zipWith (–) list1 list2
            zipWith (+) list3 list4
```
→ $\sum_{i=1}^{n} (x_i - x_{i+1})(y_i + y_{i+1})$

$$\frac{1}{2}\left|\sum_{i=1}^{n} (x_i - x_{i+1})(y_i + y_{i+1})\right|$$

Area function

```
area        :: [(Float,Float)] -> Float

area []     = error "not a polygon"
area [x]    = error "points do not have an area"
area [x,y]  = error "lines do not have an area"

area ps     = abs ((foldl (+) 0.0 parts) / 2) where
    parts   = zipWith (*) (zipWith (-) l1 l2) (zipWith (+) l3 l4)
    l1      = map fst ps
    l2      = tail l1 ++ [head l1]
    l3      = map snd ps
    l4      = tail l3 ++ [head l3]
```

list1 → [x1, x2, x3, x4]
list2 → [x2, x3, x4, x1]
list3 → [y1, y2, y3, y4]
list4 → [y2, y3, y4, y1]

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf