CS 320: Concepts of Programming Languages

Wayne Snyder
Computer Science Department
Boston University

Lecture 06: Useful Haskell Syntax, HO Programming Continued

- Goodbye to Bare Bones Haskell: Built-in syntax for lists & tuples
- Lambda expressions and Beta-Reduction
- Let and Case Expressions

Reading: Hutton Ch. 4 & 7

You should be starting to look through the Standard Prelude in Appendix B, particularly the list processing functions!

Useful Haskell Syntax: Built-In Types

We have used Bare Bones Haskell notation for Lists, Pairs, and Triples in order to emphasize the importance of pattern-matching in defining functions. However, enough is enough! Here is a more convenient syntax which is built into the basic Haskell syntax (and not just implemented as functions in the Prelude):

BB Haskell

```
data Bool = False | True

(&&) :: Bool -> Bool -> Bool
False && _ = False
True && b = b


data Nat = Zero | Succ Nat

add :: Nat -> Nat -> Nat
add Zero x = x
add (Succ x) y = Succ (add x y)
```

Flesh and Blood Haskell

Useful Haskell Syntax: Built-In Types

We have used Bare Bones Haskell notation for Lists, Pairs, and Triples in order to emphasize the importance of pattern-matching in defining functions. However, enough is enough! Here is a more convenient syntax which is built into the basic Haskell syntax (and not just implemented as functions in the Prelude):

BB Haskell

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

~~`(&&) :: Bool -> Bool -> Bool`~~
~~`False && _ = False`~~
~~`True && b = b`~~

~~`data Nat = Zero | Succ Nat`~~

~~`add :: Nat -> Nat -> Nat`~~
~~`add Zero x = x`~~
~~`add (Succ x) y = Succ (add x y)`~~

Flesh and Blood Haskell

Built in to the Prelude exactly as we presented it:

`Bool, True, False, &&, ||, not`

Built in types Integer, Double, ...

```
Main> 5 + 2
7

Main> 2039482039848029348 * 2828383838
5768438039397438184032877624
```

Useful Haskell Syntax: Built-In Tuples

BB Haskell

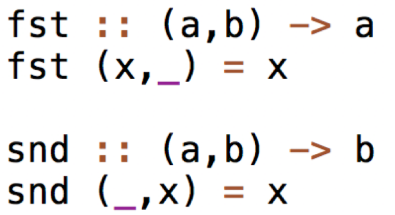
```
data Pair a b = P a b
data Triple a b c = T a b c

fst :: Pair a b -> a
fst (P x _) = x

snd :: Pair a b -> b
snd (P _ x) = x

toLeft :: (Pair a (Pair b c))
       -> (Pair (Pair a b) c)
toLeft (P x (P y z)) = (P (P x y) z)

p2T :: (Pair a (Pair b c))
    -> (Triple a b c)
p2T (P x (P y z)) = (T x y z)
```

```
Main> P 3 True
P 3 True

Main> (P 4 (P True (-9)))
P 4 (P True (-9))

Main> (T 3 5 9)
T 3 5 9

Main> (T 9 False 2)
T 9 False 2

Main> fst (P 3 True)
3

Main> snd (P 3 (P True 2))
P True 2

Main> toLeft (P 4 (P True (-9)))
P (P 4 True) (-9)

Main> p2T (P 4 (P True (-9)))
T 4 True (-9)
```

Useful Haskell Syntax: Built-In Tuples

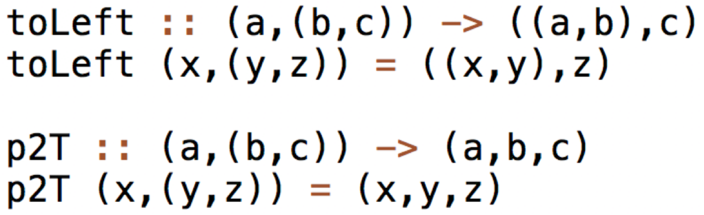
BB Haskell

```
Main> P 3 True
P 3 True
Main> (P 4 (P True (-9)))
P 4 (P True (-9))
Main> (T 3 5 9)
T 3 5 9
Main> (T 9 False 2)
T 9 False 2
```



```
fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd (_,x) = x
```

Provided in Prelude



```
toLeft :: (a,(b,c)) -> ((a,b),c)
toLeft (x,(y,z)) = ((x,y),z)

p2T :: (a,(b,c)) -> (a,b,c)
p2T (x,(y,z)) = (x,y,z)
```

Flesh and Blood Haskell

```
Main> (3,True)
(3,True)
Main> (4,(True,(-9)))
4 (True,(-9))
Main> (3,5,9)
(3,5,9)
Main> (9,False,2)
(9,False,2)
Main> fst (3,True)
3
Main> snd (3,(True,2))
(True,2)
Main> toLeft (4,(True,(-9)))
((4,True),-9)
Main> p2T (4,(True,(-9)))
(4,True,-9)
Main> (2,3,True,5,’a’,7,4,”hi”,5)
(2,3,True,5,’a’,7,4,”hi”,5)
```

Tuples can be any length, but **fst** and **snd** only work on **pairs**.

Useful Haskell Syntax: Built-In Lists

BB Haskell

```haskell
data List a = Nil
            | Cons a (List a)

head :: List a -> a
head (Cons x _) = x

tail :: List a -> List a
tail (Cons _ xs) = xs

length :: List a -> Integer
length Nil = 0
length (Cons _ xs) = 1 + (length xs)
```

```
Main>(Cons 3 (Cons 9 Nil))
Cons 3 (Cons 9 Nil)

Main> head (Cons 3 (Cons 9 Nil))
3

Main> tail (Cons 3 (Cons 9 Nil))
Cons 9 Nil

Main> length (Cons 3 (Cons 9 Nil))
2
```

Flesh and Bones Haskell

Built in as part of syntax!

Provided in Prelude

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

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

length :: [a] -> Integer
length [] = 0
length (_:xs) = 1 + (length xs)
```

```
Main> []
[]

Main> 3:9:[]
[3,9]

Main> 3:[9]
[3,9]

Main> [3,9]
[3,9]
```

```
Main> head [3,9]
3

Main> tail [3,9]
[9]

Main> length [3,9]
2
```

Useful Haskell Syntax: Built-In Lists

Start to become familiar with the list-processing functions in the Prelude, there are many useful functions already defined! See Hutton pp.285 – 287.

```
Main> [0,1,2] ++ [3,4]
[0,1,2,3,4]
Main> last [0,1,2,3,4]
4
Main> init [0,1,2,3,4]
[0,1,2,3]
Main> take 3 [0,1,2,3,4]
[0,1,2]
Main> drop 3 [0,1,2,3,4]
[3,4]
Main> takeWhile (<3) [0,1,2,3,4]
[0,1,2]
Main> dropWhile (<3) [0,1,2,3,4]
[3,4]
```

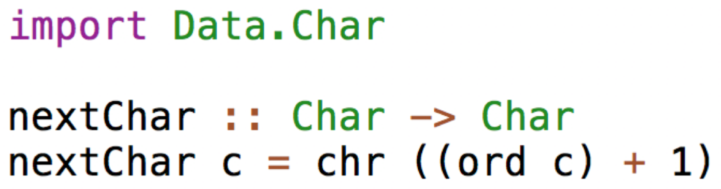
```
Main> splitAt 3 [0,1,2,3,4]
([0,1,2],[3,4])
Main> replicate 5 1
[1,1,1,1,1]
Main> [0,1,2] ++ [3,4]
[0,1,2,3,4]
Main> reverse [0,1,2,3,4]
[4,3,2,1,0]
Main> map (^2) [0,1,2,3,4]
[0,1,4,9,16]
Main> filter even [0,1,2,3,4]
[0,2,4]
Main> concat [[0],[1,2],[3,4]]
[0,1,2,3,4]
```

Many more advanced functions can be found in `Data.List`.

Useful Haskell Syntax: Characters and Strings

Characters (Hutton p.282)

```
Main> 'a'
'a'
Main> ['h','i','!']
"hi!”
```



```haskell
import Data.Char

nextChar :: Char -> Char
nextChar c = chr ((ord c) + 1)
```

```
Main Data.Char> isLower 'a'
True
Main Data.Char> isUpper 'a'
False
Main Data.Char> isAlpha 'a'
True
```

```
Main Data.Char> isDigit 'a'
False
Main Data.Char> ord 'a'
97
Main Data.Char> chr 97
'a'
Main Data.Char> digitToInt '9'
9
Main Data.Char> intToDigit 4
'4'
Main Data.Char> toUpper 'a'
'A'
Main Data.Char> toLower 'A'
'a'
Main Data.Char> nextChar 'a'
'b'
```

Useful Haskell Syntax: Characters and Strings

Strings are simply lists of Characters (Hutton p.282)

```
Main> ['h','i','!']
"hi!”

Main> "hi " ++ "there" ++ "!"
"hi there!"

Main> "hi there" !! 3
't'

Main> take 5 "hi there!"
"hi th”

Main> words "hi there!"
["hi","there!"]

Main> import Data.Char

Main Data.Char> map toUpper "hi there!"
"HI THERE!"
```

Any list function can be used on Strings. Check out `Data.List`!

This nifty function is provided in the Prelude

Case Expressions

A very useful kind of conditional expression is the case expression:

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```

In other languages, the case statement is an alternative to a long nested if-then-else, but in Haskell (of course!) it is more powerful, as it does pattern matching:

```
describe :: [a] -> String
describe []  = "empty"
describe [x] = "singleton"
describe _   = "big!"
```

```
describe :: [a] -> String
describe xs =
   case xs of []  -> "empty"
              [x] -> "singleton"
              _   -> ”big!"
```

```
*Main> describe [4]
"singleton"
```

Case Expressions

This solves the problem that lambda expressions can pattern match, but not do multiple patterns:

```
describe :: [a] -> String
describe = \xs -> case xs of
                     []   -> "empty"
                     [x]  -> "singleton"
                     _    -> ”big!"
```

Beta Reduction and Let Expressions

Recall: a lambda expression represents an anonymous function:

```
makePair :: a -> b -> (a,b)
makePair x y = (x,y)

makePair x = \y -> (x,y)

makePair = \x -> \y -> (x,y)

Main> makePair 3 True
(3,True)
```

By referential transparency, we can simply use the lambda expression and apply it directly to arguments:

```
Main> (\x -> \y -> (x,y)) 3 True
(3,True)
```

Beta Reduction and Let Expressions

We will study this much more in a few weeks, when we start to think about how to implement functional languages, but for now, we just define the concept of Beta-Reduction, which is simply substituting an argument for its parameter:

```
  ((\x -> <expression>) <argument>)
```

`=>  <expression>` with `x` replaced by `<argument>`

Examples:

```
Main> (\x -> (x,x))  4
(4,4)
Main>(\x -> [3,x,9]) 4
[3,4,9]
Main>(\x -> Just x) "hi"
Just "hi"
Main>(\x -> 5) 6
5
```

```
Main> (\x -> (\y -> (x,y))) 5 True
(5,True)
Main>(\x y -> [3,x,y]) 4 9
[3,4,9]
Main>(\x y -> \z -> [x,y,z]) 2 4 9
[2,4,9]
Main> (\x -> (\x -> (x,x))) 5 True
??
```

Beta Reduction and Let Expressions

We will study this much more in a few weeks, when we start to think about how to implement functional languages, but for now, we just define the concept of Beta-Reduction, which is simply substituting an argument for its parameter:

```
   ((\x -> <expression>) <argument>)
```

`=>  <expression>` with `x` replaced by `<argument>`

Examples:

```
Main> (\x -> (x,x))  4
(4,4)
Main>(\x -> [3,x,9]) 4
[3,4,9]
Main>(\x -> Just x) "hi"
Just "hi"
Main>(\x -> 5) 6
5
```

```
Main> (\x -> (\y -> (x,y))) 5 True
(5,True)
Main>(\x y -> [3,x,y]) 4 9
[3,4,9]
Main>(\x y -> \z -> [x,y,z]) 2 4 9
[2,4,9]
Main> (\x -> (\x -> (x,x))) 5 True
(True,True)
```

Why??

Scope in Haskell

The scope of a variable (e.g., local variable, parameter) is the region of the program where it is legal to refer to that variable.

```
Main> x

<interactive>:14:1: error: Variable not in scope: x
Main>
Main> x = 4
Main> x
4
```

In Java there are several kinds of scoping rules.....

Digression: Scope in Java

The scope of a variable (e.g., local variable, parameter) is the region of the program where it is legal to refer to that variable.

Local Variable Names: Can be referenced from point of definition to end of {…}

```
static void silly(int m) {          m
    int i = 4;                      m         i
                                    m         i
    for(int j=0; j<10; j++) {       m         i         j
        int k = 2;                  m         i         j         k
        k = k + i + j;              m         i         j         k
    }                               m         i
                                    m         i
    for(int j=0; j<20; j++) {       m         i         j
        int k = 9;                  m         i         j         k
        k = k + i - j;              m         i         j         k
    }                               m         i
                                    m         i
}
```

Digression: Scope in Java

The scope of a variable (e.g., local variable, parameter) is the region of the program where it is legal to refer to that variable.

Member names: Can be referenced ANYWHERE in the class and from outside if public

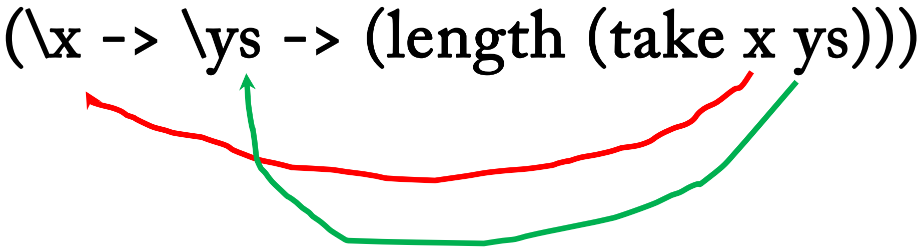
```
public class TestDefault {
       int n;                            n      m      k      p
       int m = 4;                        n      m      k      p
                                         n      m      k      p
       int sillyMethod(int q) {          n      m      k      p      q
          return q + n + m + k;          n      m      k      p      q
       }                                 n      m      k      p
                                         n      m      k      p
       int k = n + m;                    n      m      k      p
       int p = m + 1;                    n      m      k      p
  }
```

Scope in Let Expressions

The **scope of a lambda parameter** is the expression to the right of the **->**

$$(\backslash x\ \texttt{->}\ \underbrace{\texttt{<expression>}}_{\text{Scope of } x})$$

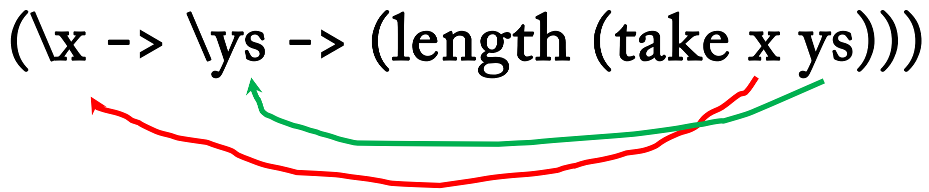
To find the parameter associated with an instance of a variable in the expression, look for the **closest enclosing binding of the variable**:



```
(\x -> \ys -> (length (take x ys)))
```

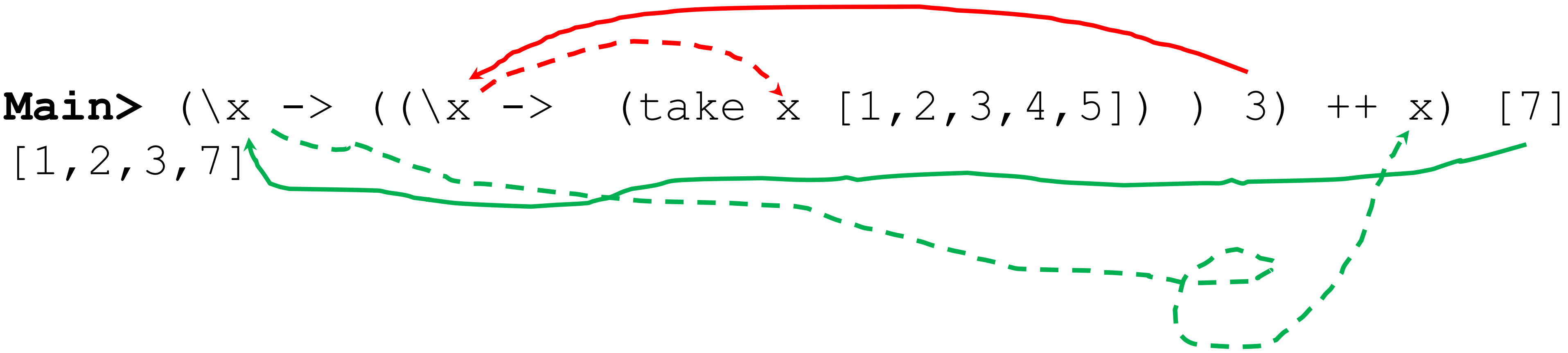
Scope in Let Expressions: Hole in Scope

To find the parameter associated with an instance of a variable in the expression, look for the **closest enclosing binding of the variable**:

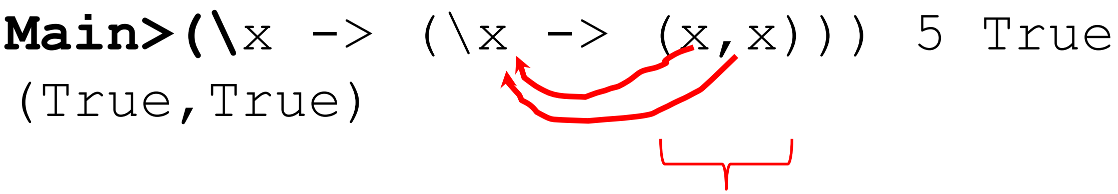


```
(\x -> \ys -> (length (take x ys)))
```

Some weird things can happen when there is more than one occurrence of the same variable:



```
Main> (\x -> ((\x ->  (take x [1,2,3,4,5]) ) 3) ++ x) [7]
[1,2,3,7]
```

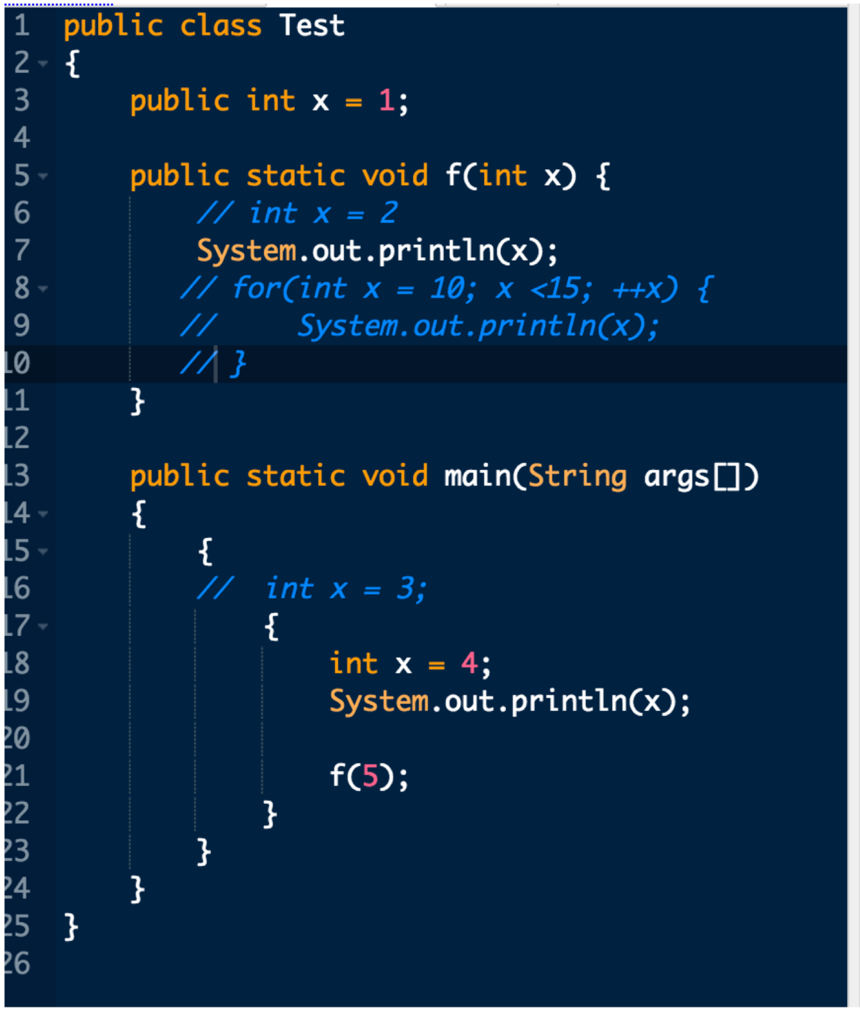


```
Main>(\x -> (\x -> (x,x))) 5 True
(True,True)
```

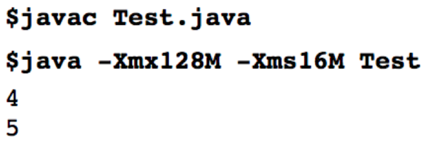
Hole in scope of outer x

Digression: Scope in Java

Java allows multiple declarations of the same variable if one is a field and one is a local variable (either a parameter or a local variable), creating a hole in the scope of the field declaration:



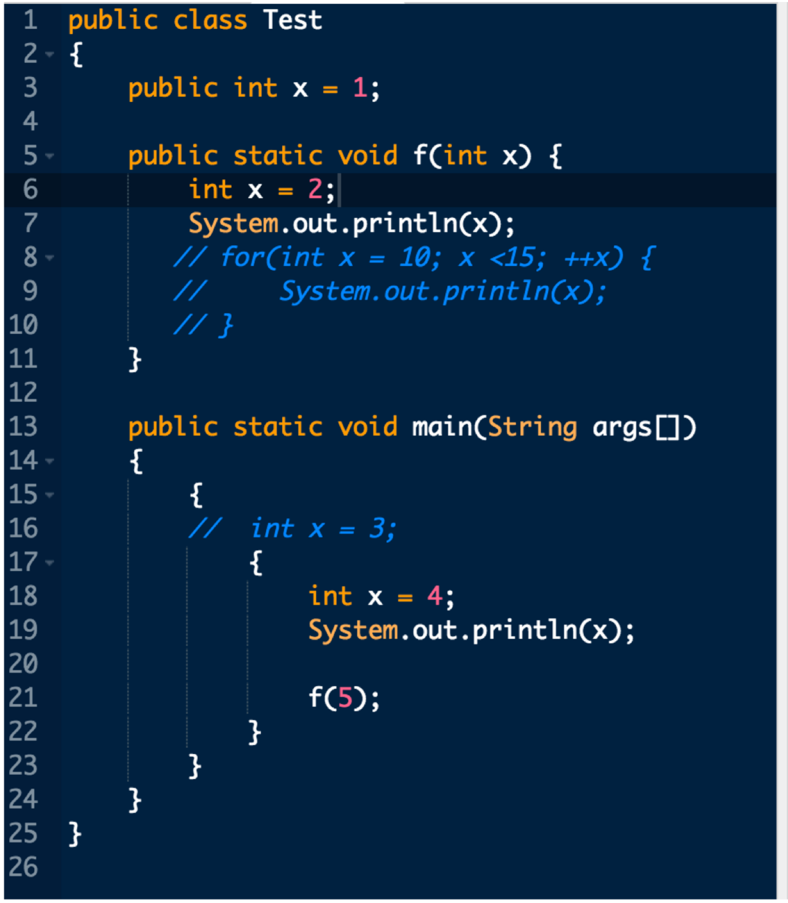
```
public class Test
{
    public int x = 1;

    public static void f(int x) {
        // int x = 2
        System.out.println(x);
       // for(int x = 10; x <15; ++x) {
       //      System.out.println(x);
       // }
    }

    public static void main(String args[])
    {
        {
        //  int x = 3;
            {
                int x = 4;
                System.out.println(x);

                f(5);
            }
        }
    }
}
```



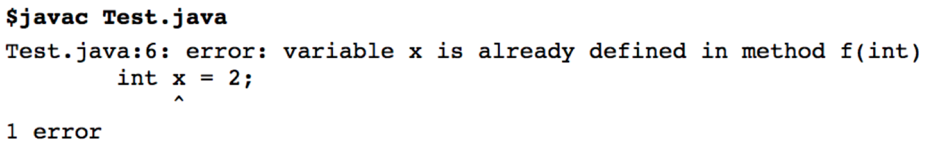
```
$javac Test.java
$java -Xmx128M -Xms16M Test
4
5
```

Digression: Scope in Java

But Java does NOT allow multiple declarations (and hence avoids the hole in scope issue) for two local variables:



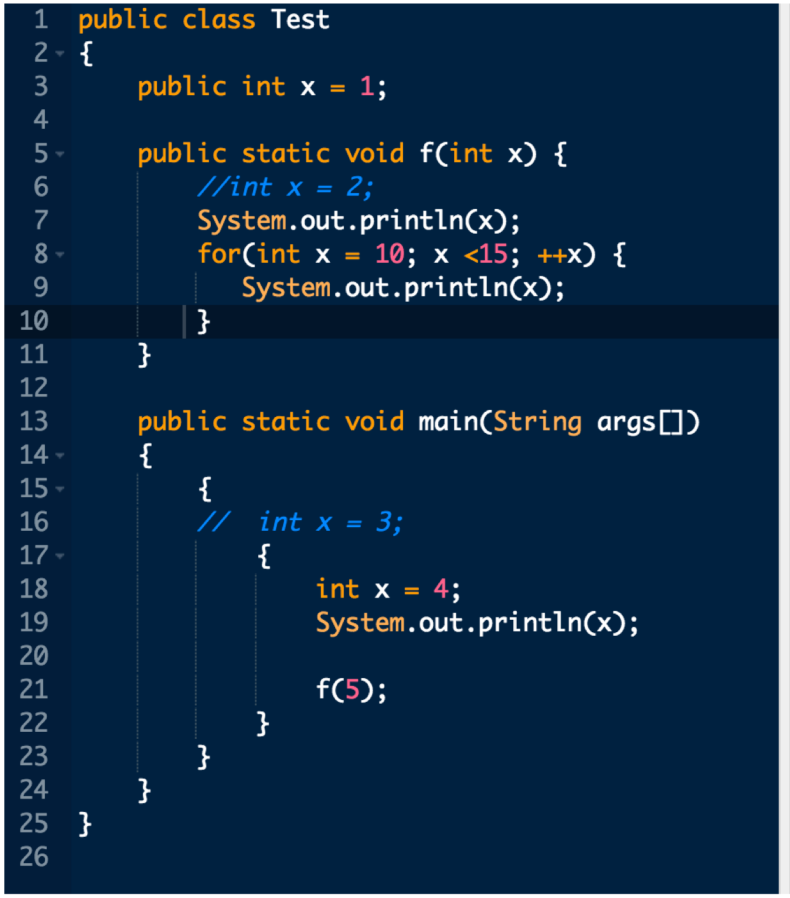
```
1  public class Test
2  {
3      public int x = 1;
4  
5      public static void f(int x) {
6          int x = 2;
7          System.out.println(x);
8         // for(int x = 10; x <15; ++x) {
9         //     System.out.println(x);
10        // }
11     }
12 
13     public static void main(String args[])
14     {
15         {
16         //  int x = 3;
17             {
18                 int x = 4;
19                 System.out.println(x);
20 
21                 f(5);
22             }
23         }
24     }
25 }
26 
```



```
$javac Test.java
Test.java:6: error: variable x is already defined in method f(int)
        int x = 2;
            ^
1 error
```

Digression: Scope in Java

But Java does NOT allow multiple declarations (and hence avoids the hole in scope issue) for two local variables:

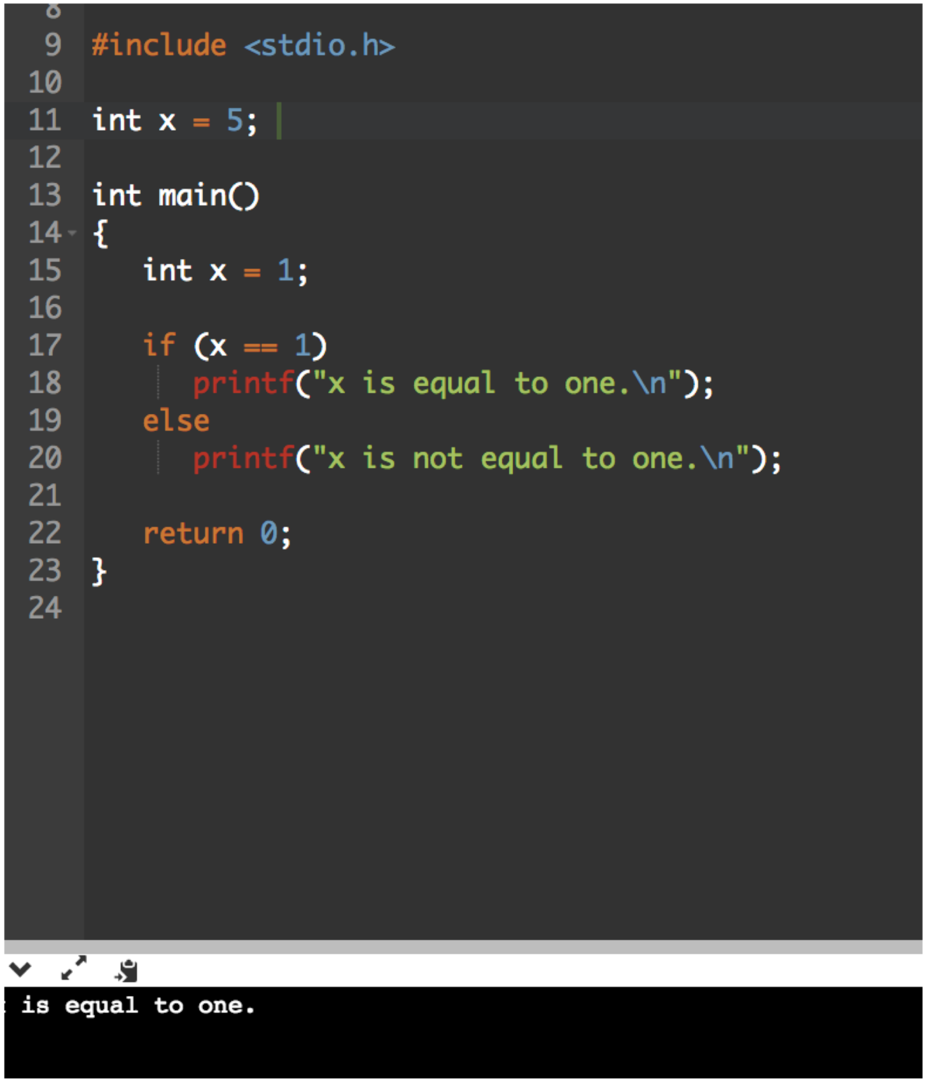


```java
public class Test
{
    public int x = 1;

    public static void f(int x) {
        //int x = 2;
        System.out.println(x);
        for(int x = 10; x <15; ++x) {
           System.out.println(x);
        }
    }

    public static void main(String args[])
    {
        {
        //  int x = 3;
            {
                int x = 4;
                System.out.println(x);

                f(5);
            }
        }
    }
}

```

```
$javac Test.java
Test.java:8: error: variable x is already defined in method f(int)
        for(int x = 10; x <15; ++x) {
                ^
1 error
```

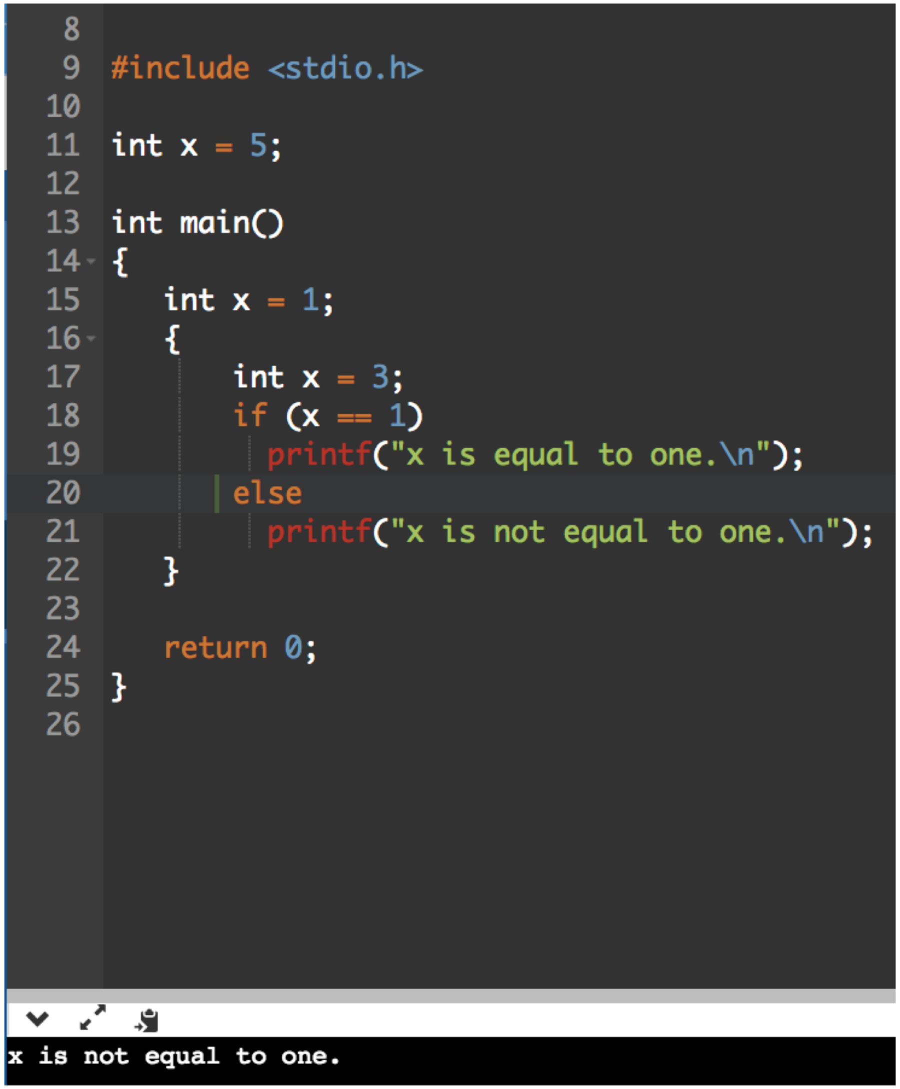
Digression: Scope in C

C allows multiple declarations without many restrictions:



```c
#include <stdio.h>

int x = 5;

int main()
{
    int x = 1;

    if (x == 1)
        printf("x is equal to one.\n");
    else
        printf("x is not equal to one.\n");

    return 0;
}
```

```
 is equal to one.
```



```c
#include <stdio.h>

int x = 5;

int main()
{
    int x = 1;
    {
        int x = 3;
        if (x == 1)
            printf("x is equal to one.\n");
        else
            printf("x is not equal to one.\n");
    }

    return 0;
}
```

```
x is not equal to one.
```

Let Expressions in Haskell

In Haskell we create local variables using let:

```
(let x = <expr1> in <expr2>)
```

```
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in  sideArea + 2 * topArea
```

Scope of local variables

Equivalent to a lambda application:

```
((\x -> <expr2>) <expr1>)
```

Except that you can have multiple bindings in the same let.

```
   let sq x = x * x in (sq 5, sq 3, sq 2)

=> (25,9,4)

   let x = 5
   in let y = 2 * x
      in let z = x + y
         in (\w -> x * y + z) 10

=> 65
```

Let Expressions in Haskell

Haskell let’s you define local variables any time you want with let (and where), and therefore hole in scope issues become relevant.

Notice the enormous flexibility of Haskell and the referential transparency principle: You can use these kinds of expressions nearly anywhere!

```
   (let sq = (\x -> x*x) in \x -> (x,sq x) ) 5

=> (5,25)


    (\x -> case x of
              1 -> \x -> x + 1
              2 -> \x -> x * 2
              _ -> \x -> x     ) 2 6

 =>  12
```