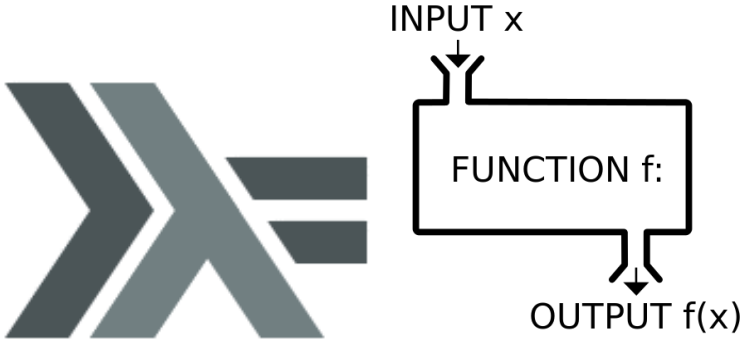
Functional Programming



Introduction

Christoph Denzler / Daniel Kröni
University of Applied Sciences Northwestern Switzerland
Institute for Mobile and Distributed Systems

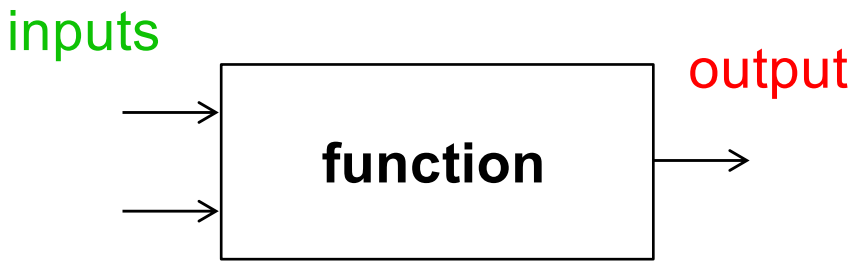
Learning Targets

You

- understand the concept of functions
- know what it means to apply a function to an argument
- have a working Haskell installation on your computer

What is a Function?

- **A Function**
  - can be pictured as a box with some inputs and an output

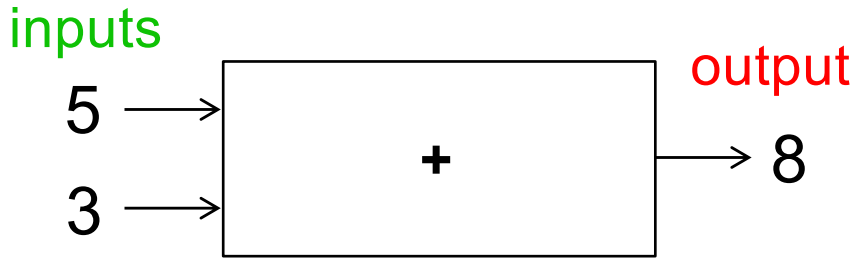


  - gives an output value which depends upon the input value(s)

- **We will often use the term result for the output and the term arguments and parameters for the input**

Function Application

- **Giving inputs to a function is called function application**

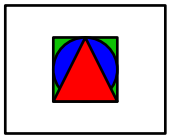


- **The function takes the inputs and computes the output**
- **Rules:**
  - **A function uses only its inputs to compute the output**
  - **It does nothing else! No side effects!**
- **Haskell functions are pure!**
  - For every specific input a function always computes exactly the same output!

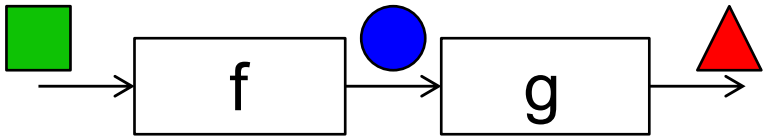
Models of Computation

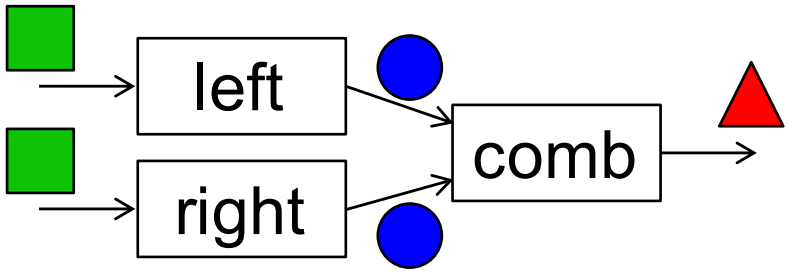
- **Imperative: Step by step instructions**
  - Changing memory cells

```
⇨ f();
⇨ g();
```

result 

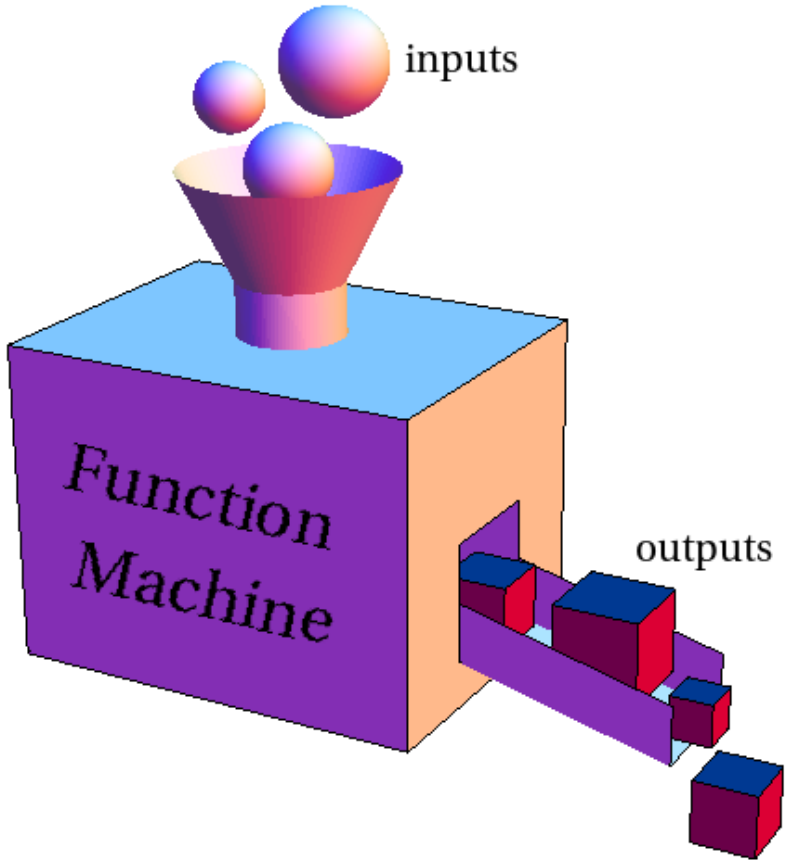
- **Functional: Applying functions to arguments**
  - Transforming data through pipelines of functions





result = g (f ■) **result = comb (left ■) (right ■)**

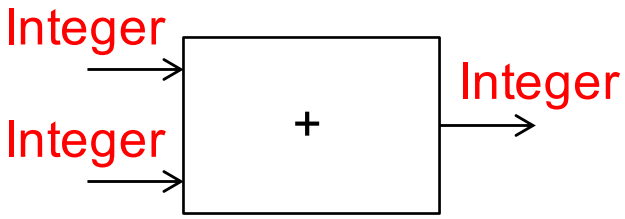
The Function Machine

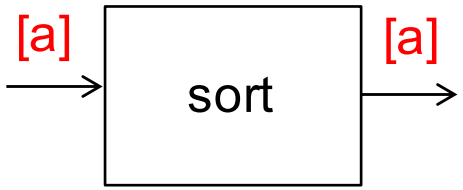


http://mathinsight.org/image/function_machine

Types

- **The input data which a function can accept as well as the output data has to be of a specific type.**
- **Examples:**





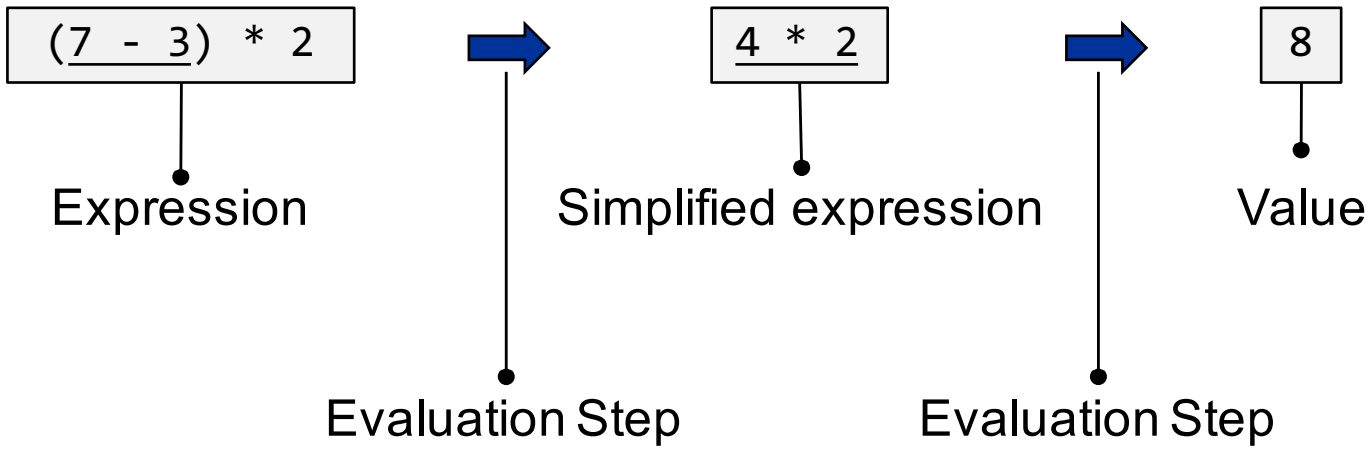
- **Type errors**
  - Applying sort to a Bool (rather than to a list with elements of any type)

```
dk — ghc — 72×8
Prelude Data.List> sort True

<interactive>:5:6:
    Couldn't match expected type ‘[a]’ with actual type ‘Bool’
    Relevant bindings include it :: [a] (bound at <interactive>:5:1)
    In the first argument of ‘sort’, namely ‘True’
    In the expression: sort True
Prelude Data.List>
```

Expressions and Evaluation in Haskell

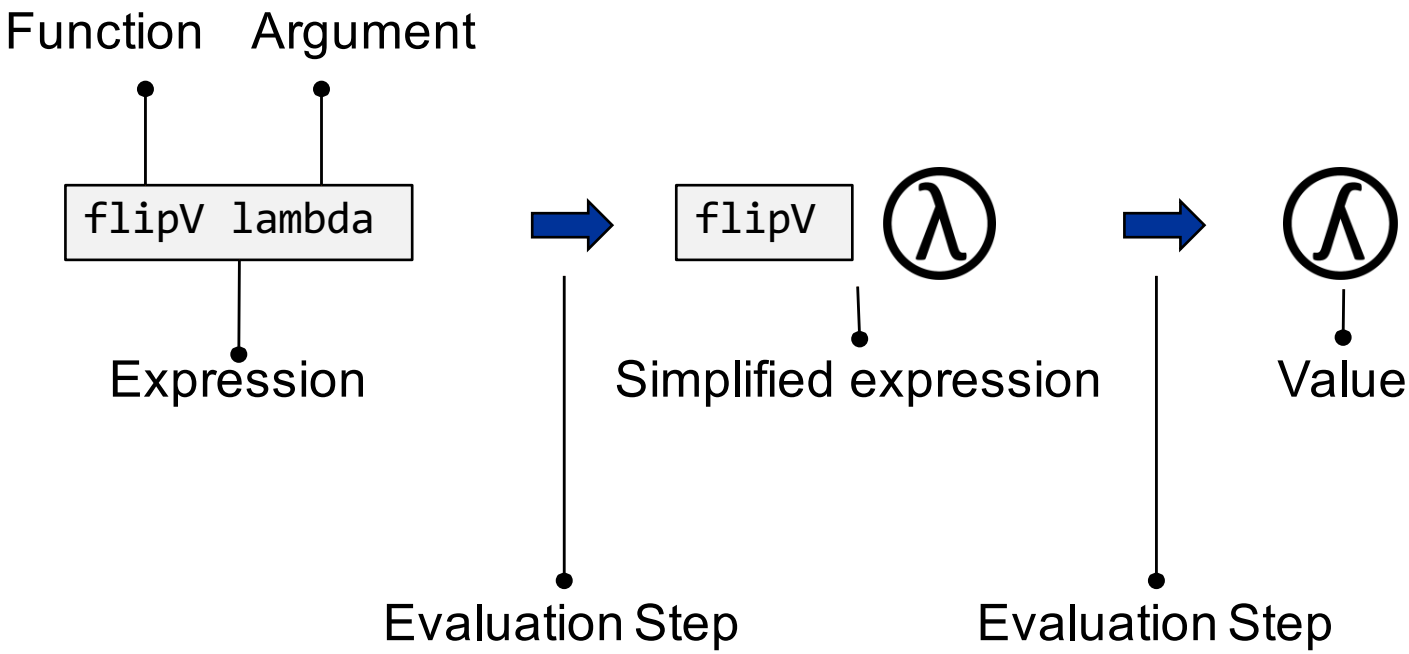
- **Evaluation is the process of calculating the resulting value of an expression.**



Expressions and Evaluation in Haskell

- **If this picture**  **is called** `lambda`
- **Then the expression** `flipV lambda` **means:**

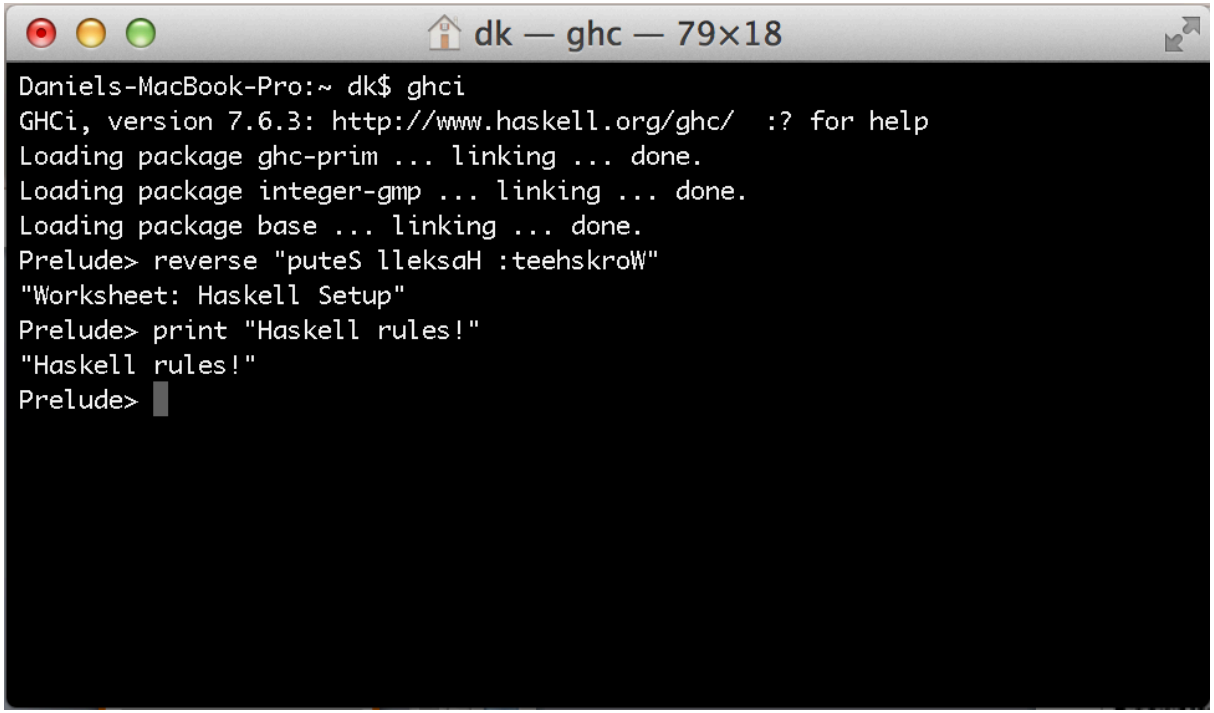
  Apply the function called flipV to the picture called lambda.



Worksheet: Haskell Setup

Key learnings

- **You know how to start GHCi and evaluate simple expressions**



Definitions

- A functional program in Haskell consists of definitions
- A definition associates a name with a value of a particular type

```
name :: type
name =  expression
```

```
type name = expression;

int size =  (7 - 3) * 2;
```

- Examples:

```
size :: Integer
size =  (7 - 3) * 2
```

Associates the name size with the value of the expression, 8, whose type is Integer.

```
fl :: Picture
fl = flipV lambda
```

Associates the name fl with the value of the expression,  , whose type is Picture.

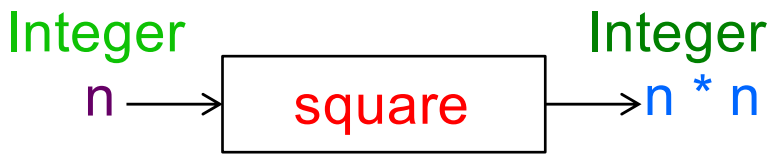
Function Definition Example I

- **Defining the square function in Haskell**

```
square :: Integer -> Integer
square n =  n * n
```

```
int square(int n) {
  return n * n;
}
```

- **Diagrammatically**



- **The first line declares the type `Integer -> Integer`**
  - The arrow '->' signifies that it is a function type
    - Taking an input/argument of type Integer
    - Returns a result/value of type Integer
- **Read as: "square is a function taking an Integer to an Integer."**

Haskell vs. Java Syntax Comparison


```haskell
-- Definitions
size :: Integer
size =  12

square :: Integer -> Integer
square n =  n * n


mul:: Integer -> Integer -> Integer
mul x y =  x * y

-- Application
square 2
mul 1 size
mul (square 2) 3
```

```java
// Definitions
int size = 12;


int square(int n) {
  return n * n;
}

int mul(int x, int y) {
  return x * y;
}
// Application
square(2);
mul(1, size);
mul(square(2), 3);
```

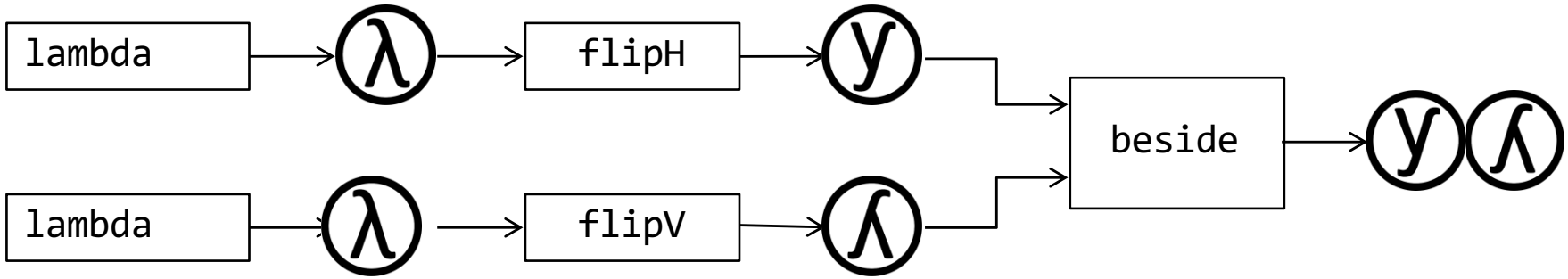
Function Definition Example II

- **Definitions**

```
flipH     :: Picture -> Picture
flipV     :: Picture -> Picture
beside    :: Picture -> Picture -> Picture
lambda    :: Picture
```

- **Expression**

```
            beside (flipH lambda   ) (flipV lambda   )
```



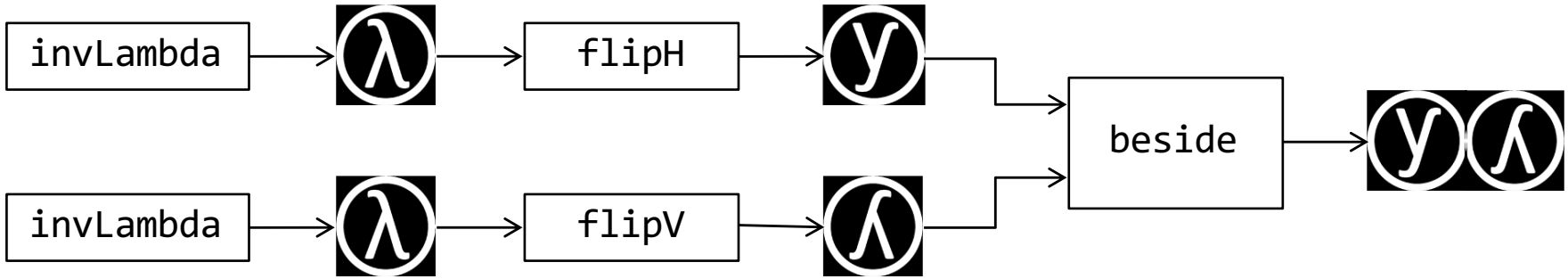
Function Definition Example II

- **Definitions**

```
flipH     :: Picture -> Picture
flipV     :: Picture -> Picture
beside    :: Picture -> Picture -> Picture
invLambda :: Picture
```

- **Expression**

```
beside (flipH invLambda) (flipV invLambda)
```



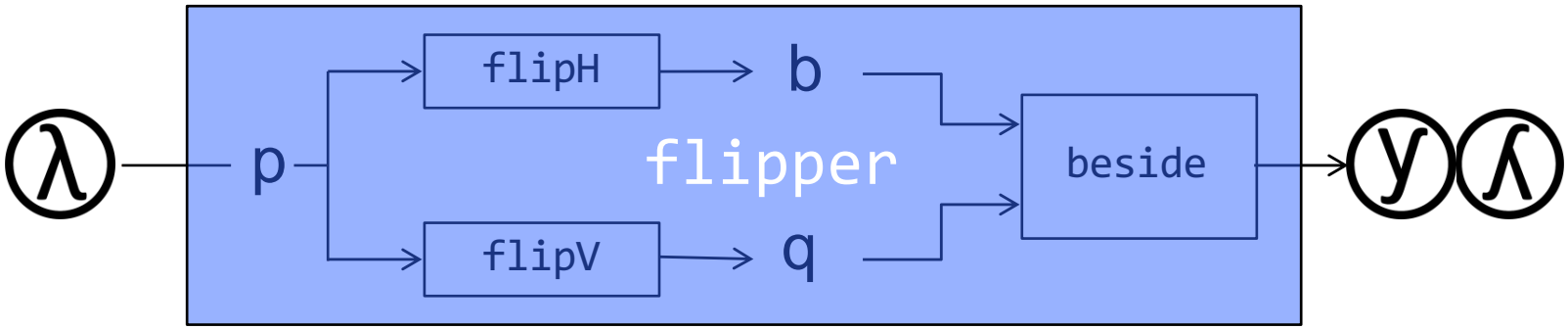
Function Definition Example II

- **Definitions**

```
flipH     :: Picture -> Picture
flipV     :: Picture -> Picture
beside    :: Picture -> Picture -> Picture
lambda    :: Picture
```

- **Defining a new function:**

```
flipper p = beside (flipH p        ) (flipV p        )
```



- **Usage:**

```
flipper lambda
```

Function Definition Example II

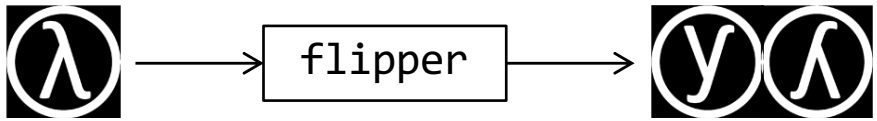
- **Defining a new function**

```
flipper :: Picture -> Picture
flipper p = beside (flipH p) (flipV p)
```

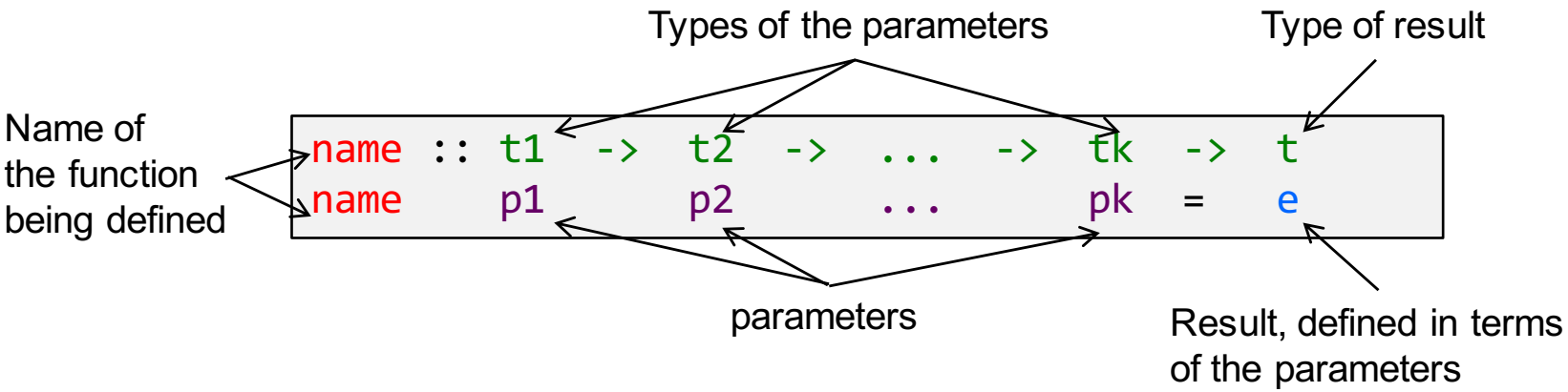
- **Expression**

```
flipper lambda
```

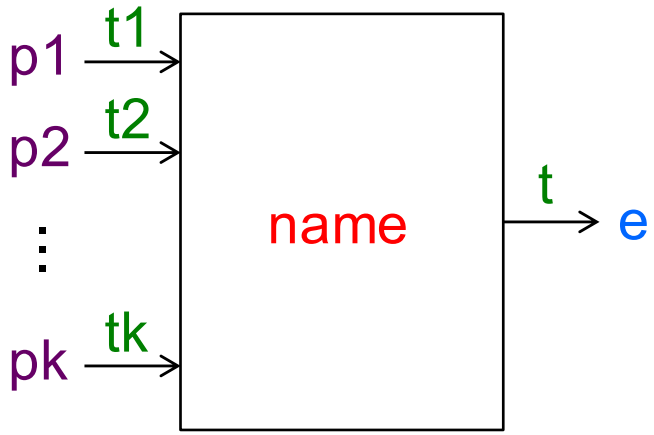

```
flipper invLambda
```



Function Definition in General



- **Diagrammatically**




```
t name(t1 p1, t2 p2, …, tk pk) {
  return e;
}
```

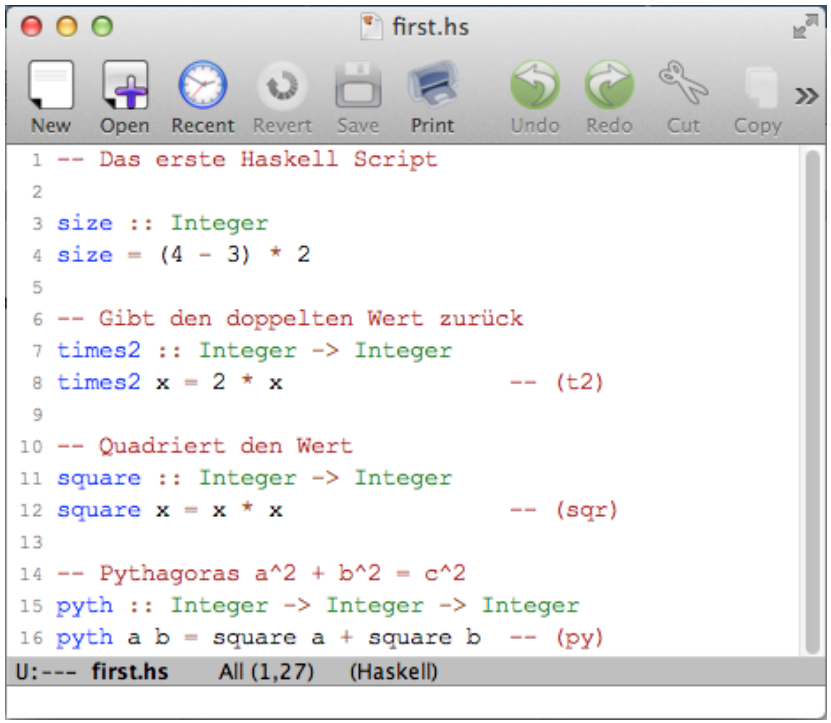
Worksheet: Haskell Script

Key learnings

- **A Haskell program is stored in a file whose name ends with .hs**
- **The development cycle is**
  1. Edit and save source code
  2. Load into GHCi

     :l filename

     :r
  3. Experiment with expressions
  4. Goto 1.



Function Application / Evaluation

- **Function application is evaluated by replacing every occurrence of a parameter with the given argument**
- **Example:**
  - To evaluate:

```
23 - (times2 (3+1))
```

    - We need to use the definition of the function:

```
times2 :: Integer -> Integer
times2 n = 2*n
```

    - We replace the parameter `n` with the argument `(3+1)` giving

```
times2 (3+1) = 2*(3+1)
```

  - By replacing equals by equals we arrive at

```
23 - (2*(3+1))
```

Function Application / Evaluation

- **Example 2:**

```
times2 :: Integer -> Integer
times2 n = 2*n                                  -- (t2)
```

Comment with label

Call by name

```
23 – (times2 (3+1))
~> 23 – (2*(3+1))                               using (t2)
~> 23 – (2*4)                                   arithmetic
~> 23 – 8                                       arithmetic
~> 15                                           arithmetic
```

Explanations (may refer to label)

Call by value

```
23 – (times2 (3+1))
~> 23 – (times2 4)                              arithmetic
~> 23 – (2*4)                                   using (t2)
~> 23 – 8                                       arithmetic
~> 15                                           arithmetic
```

Function Application / Evaluation

- **Example 3:**

```
times2 :: Integer -> Integer
times2 n = 2*n                              -- (t2)

negSum :: Integer -> Integer -> Integer
negSum a b = - (a + b)                      -- (ns)
```

```
negSum (times2 3) (times2 (-4))
~> negSum (2 * 3) (times2 (-4))          using (t2)
~> negSum 6 (times2 (-4))                arithmetic
~> negSum 6 (2 * (-4))                   using (t2)
~> negSum 6 (-8)                         arithmetic
~> - (6 + (-8))                          using (ns)
~> - (-2)                                arithmetic
~> 2                                     arithmetic
```

Why Pure Functions are Great

- **Given an unknown function named xxx**

```
xxx :: Integer -> Integer
```

- **What is the result / value of the following expression?**

```
(xxx 42) – (xxx 42)
```

Always 0! Because pure functions always return the same result when given the same arguments!

Why Objects are Dangerous

- **In comparison take the following unknown Java method**

```
class X { ...
  public int xxx(int i) { ... }
}
```

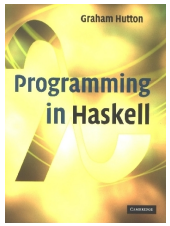
- **What can be said about the result of the following expression?**

```
X x = ...;
x.xxx(42) - x.xxx(42)
```

Nothing! Because methods can behave differently on every invocation!

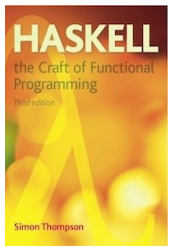
```
class X {
  int cnt = 3;
  public int xxx(int i) {
    if(--cnt == 0) {
      killBambi();
      return i * cnt;
    }
    return i * 3;
  }
}
```

Further Reading



Chapter 1 and Chapter 2

Pages 1 – 16



Chapter 1 and Chapter 2

Pages 1 - 38

Chapter 1

Pages 1 - 7