



# IO and Instructions

# Apple Pie

## Mumsig äppelpaj

Värm upp ugnen till 225 grader, blanda ingredienserna nedan och se till att fatet är både ugnsäkert och insmört med margarin. Lägg på äpplena som du tärnar först och sen kanel och socker ovanpå. Håll på resten av smulpajen och låt stå i ugnen i ca 25 minuter. Servera med massor av vaniljsås!

2.5 dl mjöl

100 gram margarin

5-6 äpplen, gärna riktigt stora

1 dl socker

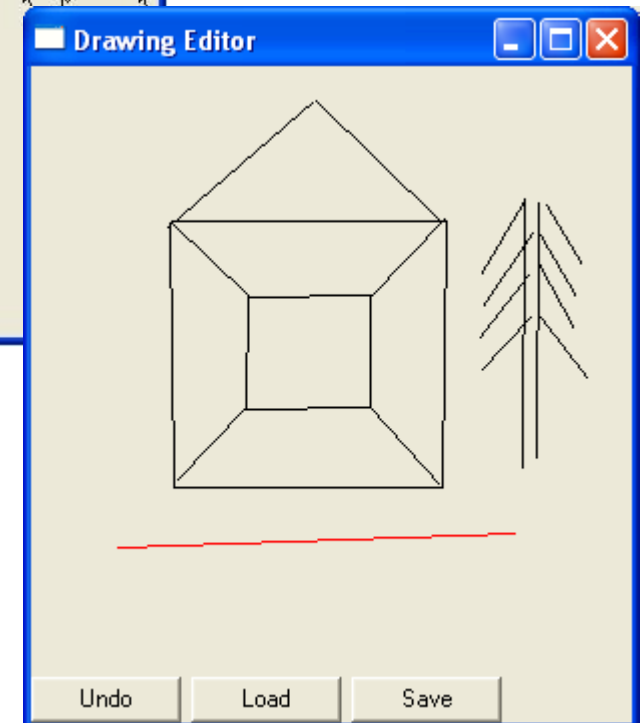
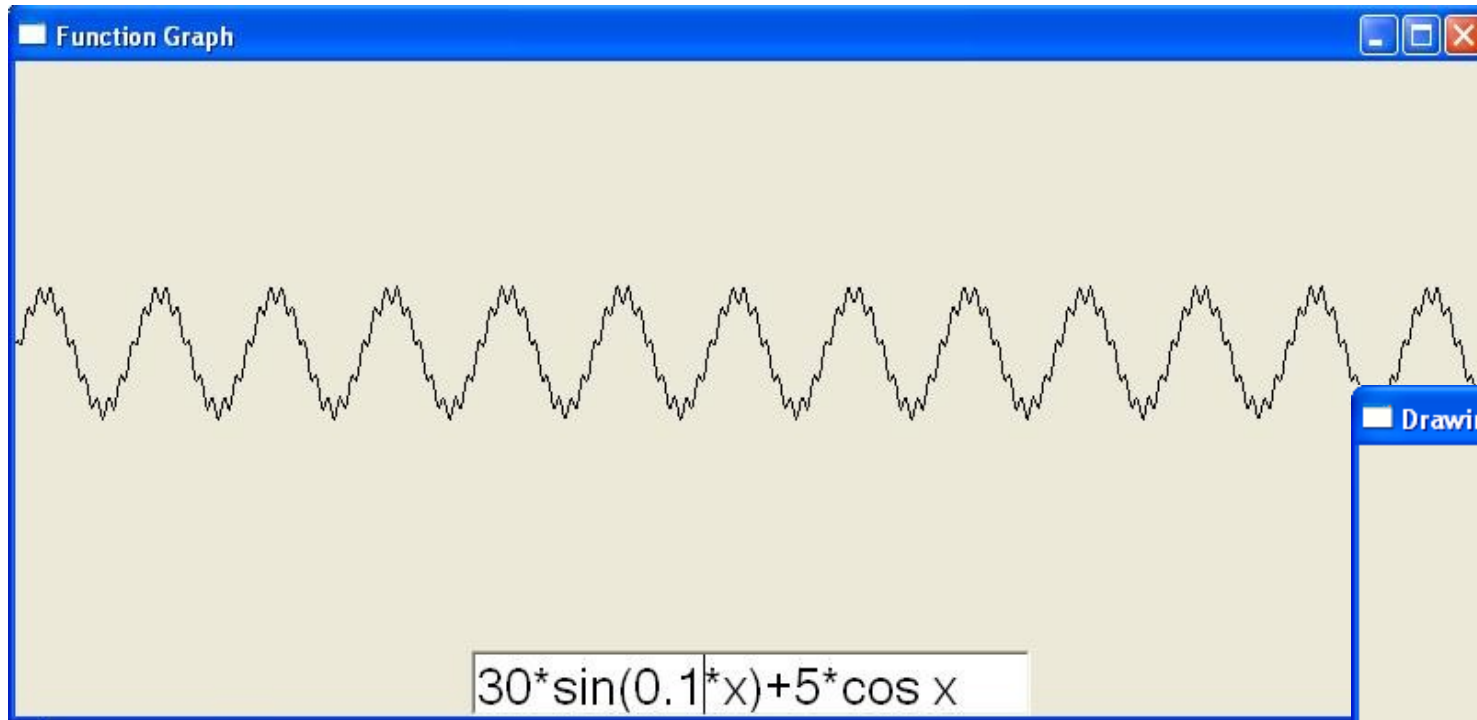
1 msk kanel

Mycket vaniljsås, gärna Marzan



Difference?

# Running a Program



How do you write  
this as a function?

What is the type  
of the result?

# A Simple Example

```
Prelude> writeFile "myfile.txt" "Anna+Kalle=sant"  
Prelude>
```

- Writes the text “Anna+Kalle=sant” to the file called “myfile.txt”
- No result displayed – why not?

# What is the Type of writeFile?

```
Prelude> :i writeFile  
writeFile :: FilePath -> String -> IO ()
```

Just a String

**INSTRUCTIONS** to the  
operating system to write  
the file

- When you give GHCi an expression of type IO, it *obeys the instructions* (instead of printing the result)
- Note: The function writeFile does *not* write the file
- It only computes the instruction to write

# The type `()`

- The type `()` is called the *unit type*
- It only has one value, namely `()`
- We can see `()` as the “empty tuple”
- It means that there is no interesting result

# The type FilePath

- Is a *type synonym*...
- ...which is a way to give an additional name to a type that already exists

```
type FilePath = String
```

- for convenience and/or documentation
- Remember: **data** creates a *new type*, which is different

```
data Shape = Circle Float | ...
```



# Instructions with a result value

```
Prelude> :i readFile  
readFile :: FilePath -> IO String
```

**INSTRUCTIONS** for  
computing a String

# Instructions vs. values – an analogy

- Instructions:

1. Take this card
2. Put the card into the ATM
3. Enter the code “1437”
4. Select “500kr”
5. Take the money

- Value:



Which would you rather have?

# Instructions vs. values – an analogy

## Mumsig äppelpaj

Värm upp ugnen till 225 grader, blanda ingredienserna nedan och se till att fatet är både ugnsäkert och insmört med margarin. Lägg på äpplena som du tärnar först och sen kanel och socker ovanpå. Håll på resten av smulpajen och låt stå i ugnen i ca 25 minuter. Servera med massor av vaniljsås!

2.5 dl mjöl

100 gram margarin

5-6 äpplen, gärna riktigt stora

1 dl socker

1 msk kanel

Mycket vaniljsås, gärna Marzan



Which would  
you rather have?

# Instructions with a result value

```
Prelude> :i readFile  
readFile :: FilePath -> IO String
```

**INSTRUCTIONS** for  
computing a String

We cannot extract 500kr  
from the list of instructions  
either...

- readFile “myfile.txt” is not a String
- no String can be extracted from it...
- ...but we can combine it with other instructions that use the result

# Putting Instructions Together

```
writeTwoFiles :: FilePath -> String -> IO ()
writeTwoFiles file s =
  do writeFile (file ++ "1") s
     writeFile (file ++ "2") s
```

Use **do** to combine  
instructions into larger  
ones

```
copyFile :: FilePath -> FilePath -> IO ()
copyFile file1 file2 =
  do s <- readFile file1
     writeFile file2 s
```

# Putting Instructions Together

```
catFiles :: FilePath -> FilePath -> IO String
catFiles file1 file2 =
  do s1 <- readFile file1
      s2 <- readFile file2
      return (s1++s2)
```

Use **do** to combine instructions into larger ones

Use **return** to create an instruction with just a result

```
return :: a -> IO a
```

# Instructions vs. Functions

- **Functions** always give the same result for the same arguments
- **Instructions** can behave differently on different occasions
- Confusing them is a major source of bugs
  - Most programming languages do so...
  - ...understanding the difference is important!

# The IO type

```
data IO a  -- a built-in type

putStr     :: String -> IO ()
putStrLn   :: String -> IO ()
readFile   :: FilePath -> IO String
writeFile  :: FilePath -> String -> IO ()
...
```

Look in the standard modules:  
System.IO, System.\*



# Some Examples

- `doTwice :: IO a -> IO (a,a)`
- `dont :: IO a -> IO ()`
- `second :: [IO a] -> IO a`
  
- (see file `ExampleIO.hs`)

# Evaluating & Executing

- IO actions of result type ()
  - are just executed in GHCi

```
Prelude> writeFile "emails.txt" "anna@gmail.com"
```

- IO actions of other result types
  - are executed, and then the result is printed

```
Prelude> readFile "emails.txt"  
"anna@gmail.com"
```

# Quiz

- Define the following function:

```
sortFile :: FilePath -> FilePath -> IO ()
```

- “sortFile file1 file2” reads the lines of file1, sorts them, and writes the result to file2
- You may use the following standard functions:

```
sort      :: Ord a => [a] -> [a]  
lines    :: String -> [String]  
unlines  :: [String] -> String
```

# Answer

```
sortFile :: FilePath -> FilePath -> IO ()
sortFile file1 file2 =
  do s <- readFile file1
     writeFile file2 (unlines (sort (lines s)))
```

General guideline:  
Do as much as possible using pure functions.  
Only use IO when you *have to*.

# Recursive instructions

- Let's define the following function:

```
getLine :: IO String
```

```
Prelude> getLine  
apa  
"apa"
```

- We may use the following standard function:

```
getChar :: IO Char
```

# Two useful functions

```
sequence_ :: [IO ()] -> IO ()  
sequence  :: [IO a]  -> IO [a]
```

Can be used to *combine* lists of instructions into one instruction

# Analogy for sequence

sequence :: [IO a] -> IO [a]

Book of recipes for  
cookies

Instruction to bake all  
cookies in the book

Cookie jar

# An Example

- Let's define the following function:

```
writeFiles :: FilePath -> [String] -> IO ()
```

```
Prelude> writeFiles "file" ["apa", "bepa", "cepa"]
```

```
Prelude> readFile "file1"  
"apa"
```

```
Prelude> readFile "file3"  
"cepa"
```

- We may use the following standard functions:

```
show      :: Show a => a -> String  
zip      :: [a] -> [b] -> [(a,b)]
```



# A possible definition

```
writeFiles :: FilePath -> [String] -> IO ()
writeFiles file xs =
  sequence_ [ writeFile (file++show i) x
             | (x,i) <- zip xs [1..length xs]
             ]
```

We create complex instructions by combining simple instructions

# Definitions?

```
sequence_ :: [IO ()] -> IO ()
```

```
sequence :: [IO a] -> IO [a]
```

# Functions vs. Instructions

- **Functions** always produce the same results for the same arguments
- **Instructions** can have varying results for each time they are executed
- Are these functions?

```
putStrLn :: String -> IO ()  
readFile :: FilePath -> IO String  
sequence :: [IO a] -> IO [a]
```

YES! They deliver  
the same instructions for the  
same arguments

(but executing these  
instructions can have  
different results)

# What is the Type of doTwice?

```
Prelude> :i doTwice  
doTwice :: Monad m => m a -> m (a, a)
```

Monad = Instructions

There are several different kinds of instructions!

- We will see other kinds of instructions (than IO) in the next lecture

# Reading

Chapter 9 of Learn You a Haskell:

<http://learnyouahaskell.com/input-and-output>  
 (“Instructions” are called “actions”)

# Do's and Don'ts

```
isBig :: Integer → Bool
isBig n | n > 9999 = True
        | otherwise = False
```

guards and  
boolean results

```
isBig :: Integer → Bool
isBig n = n > 9999
```

# Do's and Don'ts

```
resultIsSmall :: Integer → Bool  
resultIsSmall n = isSmall (f n) == True
```

comparison with a  
boolean constant

```
resultIsSmall :: Integer → Bool  
resultIsSmall n = isSmall (f n)
```

# Do's and Don'ts

```
resultIsBig :: Integer → Bool  
resultIsBig n = isSmall (f n) == False
```

comparison with a  
boolean constant

```
resultIsBig :: Integer → Bool  
resultIsBig n = not (isSmall (f n))
```



# and Don'ts

Do not make unnecessary case distinctions

```
fun1 :: [Integer] → Bool
fun1 []      = False
fun1 (x:xs) = length (x:xs) == 10
```

necessary case distinction?

repeated code

```
fun1 :: [Integer] → Bool
fun1 xs = length xs == 10
```

# Do's and Don'ts

Make the base case  
as simple as possible

```
fun2 :: [Integer] → Integer  
fun2 [x] = calc x  
fun2 (x:xs) = calc x + fun2 xs
```

right base case ?

repeated code

```
fun2 :: [Integer] → Integer  
fun2 [] = 0  
fun2 (x:xs) = calc x + fun2 xs
```