Functional Programming and Compiler Design

Praktikum WS 2001/2

Tobias Nipkow & Norbert Schirmer

(Slides largely by Simon Thompson)

Why functional programming?

Simple but powerful model of computing ...
... where you can understand the fundamentals of programming

Complements Java and OO programming ...
... helps you to understand them.

New .... challenging ... fun

2

How is it taught?

Introduction to functional programming in Haskell
Weekly lectures in the first half of term

Compiler Design
4 lectures + projects in the second half of term:
•Lexer and parser
•Static analysis
•Abstract machine
•Code generation

3

The Web page

Backup materials: exercises, solutions, booklist, background etc.

Hugs error messages

The Hugs98 distribution and documentation.

...

4

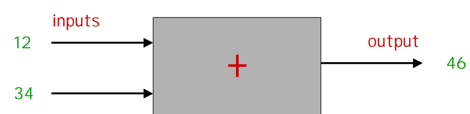
Today:

Intro to functional programming in Haskell

5

What is a function?

A function gives an output value which depends on some input values:



6

Functions over pictures

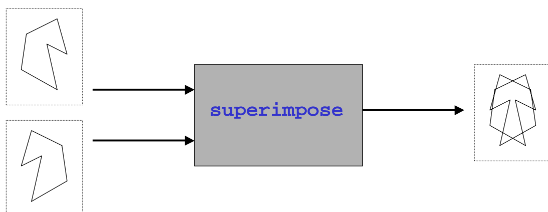
A function to flip a picture in a vertical mirror:

Functions over pictures

A function to invert the colours in a picture:

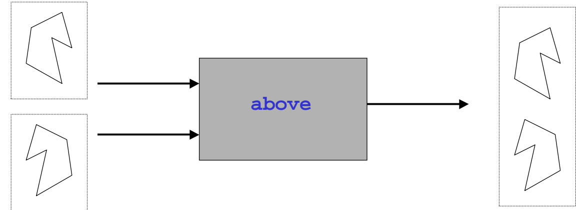
Functions over pictures

A function to superimpose two pictures:



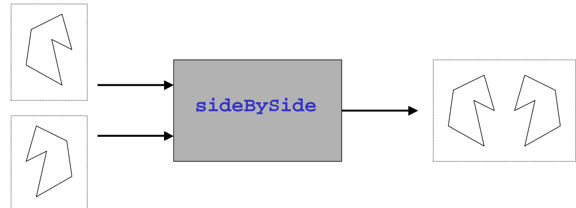
Functions over pictures

A function to put one picture above another:



Functions over pictures

A function to put two pictures side by side:



Types

A type is a collection of values, like numbers or pictures.

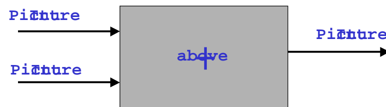
Grouped together because we can do the same things to them ...

... we can add two numbers, but we can't add a picture to a number, and indeed we can't add two pictures.

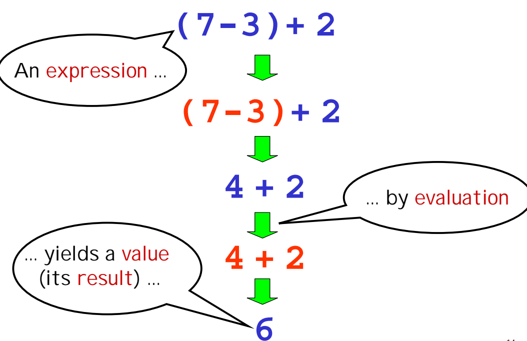
Types and functions

Functions have types.

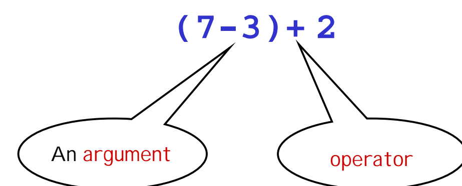
Given inputs of particular types, they produce outputs of a certain type.



Expressions and evaluation



Expressions and evaluation



An expression is made by applying a function or operator to arguments, which are also expressions.

Can evaluate these expressions in a calculator.

Haskell and Hugs

Haskell is a standard functional programming language, named after Haskell B. Curry.

Haskell98 is a recent `standard' version of the language.

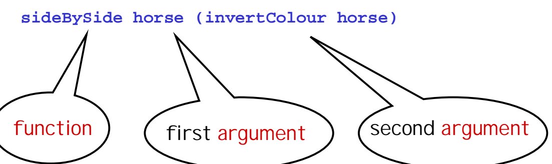
There are various implementations, including Hugs, which we use.

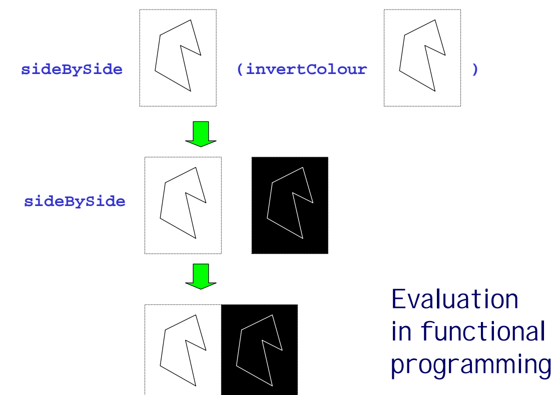
Hugs works on Windows, Mac and Unix.

Another standard functional language: ML

Expressions in functional programming

Applying a function ...





Evaluation in functional programming

Definitions in Haskell

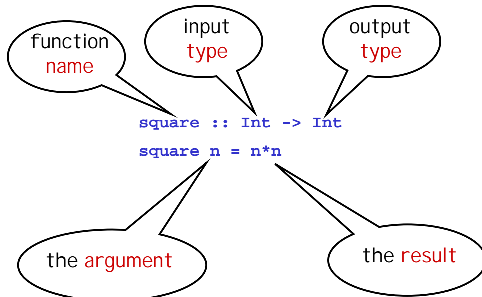
```
name :: Type
name = expression
```

```
size :: Int
size = 12+13

blackHorse :: Picture
blackHorse = invertColour horse
```

19

Function definitions in Haskell



20

Function definitions in Haskell

Other examples

```
double :: Int -> Int
double n = 2*n

rotate :: Picture -> Picture
rotate pic = flipH (flipV pic)
```

21

Evaluation in Haskell

Definitions

```
square  6 = 6*6
double  3 = 2*3
```

Evaluation

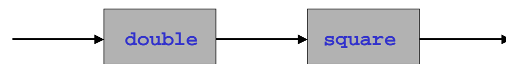
```
square (double 3)
➡ square (2*3)
➡ square 6
➡ 6*6
➡ 36
```

22

Square and double

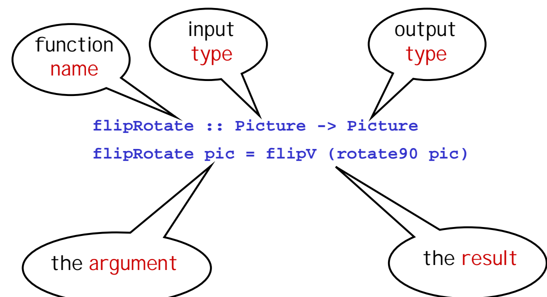
Double then square ...

```
doubleThenSquare :: Int -> Int
doubleThenSquare n = square (double n)
```



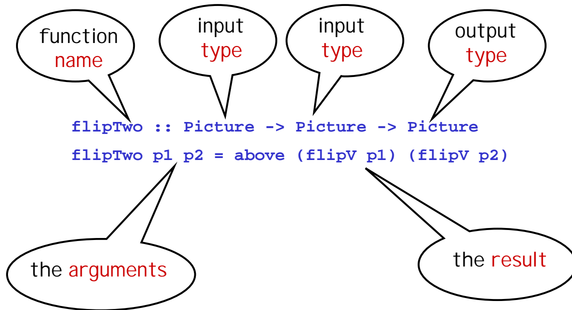
23

Function definitions in Haskell



24

Function definitions in Haskell (2)



25

Notes on syntax

Function application binds more strongly than everything else

Example: `f n-1` means `(f n)-1` not `f(n-1)`

Function application associates to the left

Example: `f g x` means `(f g) x` not `f(g x)`

26

The basic types

- Bool
- Int, Integer
- Float, Double
- Char, String

27

Booleans

Everything is `True` or `False`

28

Some example tests on numbers

```
3>47
3/=45
3==45
4<=6
```

29

More complex tests

Is someone a child?

```
0 <= age && age <= 16
```

Is someone a child or an older person?

```
(0 <= age && age < 18) || age >= 60
```

Is someone not an adult?

```
not (18 <= age && age < 59) && 0 <= age
```

30

Boolean operators

The Boolean operators in Haskell are

`&&`	and
`||`	or
`not`	not

The `||` is inclusive: `a||b` is `True` if

- `a` is `True`
- `b` is `True`
- both are `True`

31

Restaurant menu or

Choose pizza or spaghetti … but not both.

```
exOr :: Bool -> Bool -> Bool

exOr x y = (x || y) && not (x && y)

exOr x y = (x && not y) || (y && not x)

exOr x y = not ((x && y) || (not x && not y))

exOr x y = not (x==y)

exOr x y = x/=y
```

32

The truth table definition

```
exOr :: Bool -> Bool -> Bool

exOr True  True   = False
exOr True  False  = True
exOr False True   = True
exOr False False  = False
```

We'll come back to this sort of pattern matching definition later on.

33

Booleans and guards

We want to be able to do different things depending on whether a certain condition holds or not …

```
min :: Int -> Int -> Int

min n m
  | n < m       = n
  | otherwise   = m
```

If this is `True` the result is `n` …

… if not the result is `m`

34

Booleans and guards (2)

We evaluate the guards in sequence; we only look at the second if the first is `False` and so on …

```
howManyEqual :: Int -> Int -> Int -> Int

howManyEqual n m k
  | n==m && m==k           = 3
  | n==m || m==k || k==n   = 2
  | otherwise              = 1
```

… and if we reach here they're all different.

If we reach here they're not all equal …

35

The general form of functions

A function definition can have any number of clauses: in any particular case we use the first one with a `True` guard.

```
fun :: t1 -> t2 -> … -> tn -> t

fun x1 x2 … xn
  | guard1          = result1
  | guard2          = result2
  | otherwise       = resultm
```

36

Numbers in Haskell

Haskell has whole numbers

```
Int        Integer
```

and fractional (or `real') numbers

```
Float      Double
```

They differ in accuracy … `Int` are stored 16-bit boxes, whilst `Integer` are of arbitrary size.

What can we do with integers ? …

```
+    *    -    ^         2^3 = 8

`div`                    11 `div` 4 = 2

`mod`                    11 `mod` 4 = 3

<=   <    ==   /=   >    >=

show
```


… and floating point numbers?

```
+    *    -    /         11/4 = 2.75

<=   <    ==   /=   >    >=

sin  cos  …              fromInt

round                    round 2.75   = 3
floor                    floor 2.75   = 2
ceiling                  ceiling 2.75 = 3
```


Characters

Letters, digits and special characters.

Literal characters are:

```
'a'  'b'  'c'  …  'A'  'B'  …  '0'  '1'  '2'  …
```

Special characters are:

`'\n'`	`'\t'`	`'\\'`	`'\''`	`'\"'`
newline	tab	backslash	single quote	double quote

Coding characters

To play around with characters, we can convert them to and from their ASCII numerical codes:

```
chr :: Int -> Char
ord :: Char -> Int

chr 100 = 'd'
ord 'A' = 65
```


Functions over characters

Using the numerical codes to manipulate characters:

```
toUpper :: Char -> Char
toUpper ch = chr (ord ch + offset)

offset :: Int
offset = ord 'A' - ord 'a'

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

Overloading

The same name is used to mean different (but related) things.

```
(==) :: Int -> Int -> Bool
(==) :: Char -> Char -> Bool

(+) :: Int -> Int -> Int
(+) :: Float -> Float -> Float
```

More on Haskell's type system later ...

43

Pitfalls

Functions and other values begin with small letters ...
... types begin with capital letters.

Function `min` is already defined in the standard prelude of commonly used functions.

If we want to redefine it we need to say:

```
import Prelude hiding (min)
```

44

Recursion

Recursion means that a function is used in its own definition

In Haskell all iterative computations are expressed by recursion - there is no need for loops (`while` or `for`)

45

Example: factorial

Definition

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

and calculation by rewriting:

```
fac 3
~>  fac 2 * 3
~>  (fac 1 * 2) * 3
~>  ((fac 0 * 1) * 2) * 3
~>  ((1 * 1) * 2) * 3
~>  6
```

46

Undefined or error values

The problem of `fac (-2)` ... two solutions:

```
fac n
  | n==0       = 1
  | n>0        = fac (n-1) * n
  | otherwise  = 0

fac n
  | n==0       = 1
  | n>0        = fac (n-1) * n
  | otherwise  = error "negative fac"
```

47

Primitive recursion

What if I were given `fun (n-1)` ; how would I define `fun n`?

A template or pattern:

```
fun n
  | n==0       = ....
  | n>0        = .... fun (n-1) ....
```

Example: powers of two.

```
power2 :: Int -> Int
```

48

Powers and their sums

```
power2 :: Int -> Int

power2 n
  | n==0        = 1
  | n>0         = 2 * power2 (n-1)
```

How can you add the powers of two?

```
sumPowers :: Int -> Int

sumPowers n = power2 0 + … + power2 n
```

49

Powers and their sums

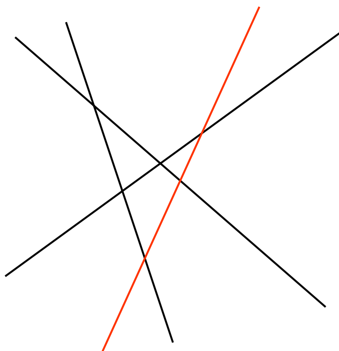
How can you add the powers of two?

```
sumPowers :: Int -> Int

sumPowers n
  | n==0      = power2 0
  | n>0       = sumPowers (n-1) + power2 n
  | otherwise = 0
```

50

How many pieces with `n` cuts?



51

How many pieces with `n` cuts?

No cuts: 1 piece.

With the `n`th cut, you get `n` more pieces:

```
cuts :: Int -> Int
cuts n
  | n==0        = 1
  | n>0         = cuts (n-1) + n
  | otherwise   = 0
```

52

Not everything is primitive recursive

```
gcd :: Int -> Int -> Int
gcd m n
  | n==0        = m
  | otherwise   = gcd n (m `mod` n)
```

53

`if then else`

If `b::Bool` and `e1::T` and `e2::T` then
`if b then e1 else e2 :: T`

Example:

```
fac n = if n==0 then 1 else n*fac(n-1)
```

54

Local definitions: `let` and `where`

```
let x = 42 in x+x
~> 42+42 ~> 84
```

Alternative: `x+x where x=42`

Works also for functions:

```
f 42 where f 0 = 0
           f n = f(n-1)
```


Layout

The way that a program is laid out in Haskell counts.

Follow the style of the book ...

... otherwise can get bizarre error messages ...

... which should lead you to the errors page.

What is a script?

It defines a collection of fixed or constant things.

```
size :: Int
size = 12+13

flipRotate :: Picture -> Picture
flipRotate pic = flipV (rotate90 pic)
```

`flipRotate` always has the same effect, and the value of `size` is always 25 ... not like variables in Java.

Regular and literate scripts

In a regular script there are definitions and comments:

```
-- FirstScript.hs
-- 5 October 2000

-- Double an integer.

double :: Int -> Int
double n = 2*n
```

Everything is program, except comments beginning --.

In a literate script there are comments and definitions:

```
FirstLit.lhs
5 October 2000

Double an integer.

>  double :: Int -> Int
>  double n = 2*n
```

Everything is comment, except program beginning > .