Programming Paradigms

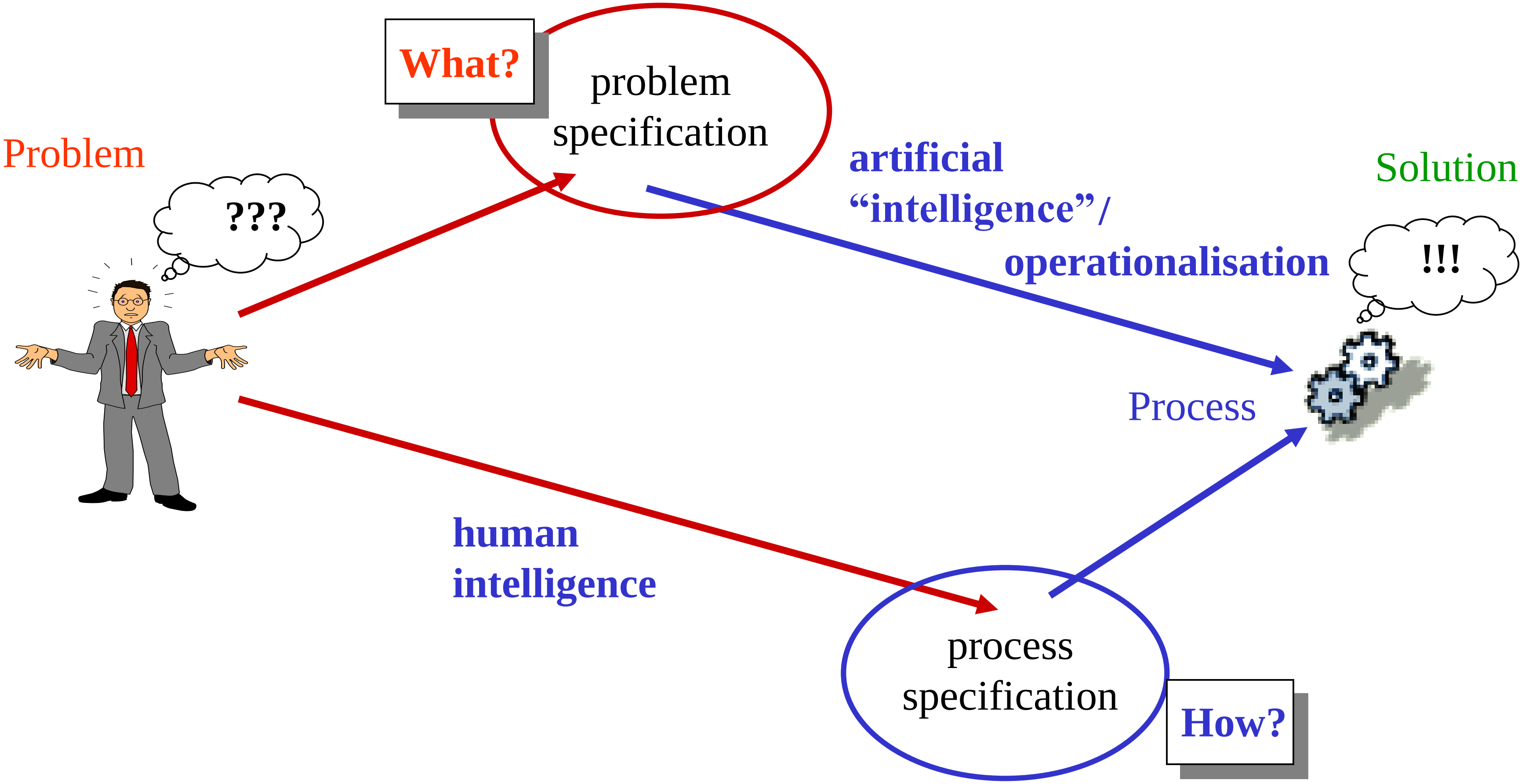
Summer Term 2017

4th Lecture

Prof. Janis Voigtländer
University of Duisburg-Essen

Ideal (and to some extent, history) of declarative programming

Freeing the programmer from the necessity to explicitly plan and specify the computation process that leads to a problem solution: **“What instead of How”**



Characteristics of declarative specifications (vs. imperative programs)

- Declarative programs (specifications) are often:
  - significantly shorter
  - significantly more readable
  - significantly more maintainable (and more reliable)

  than their imperative "counterparts".

- In particular functional programming languages emphasize abstractions that exclude/constrain or (flexibly) put under control side effects like mutation etc. (S. Peyton Jones: "Haskell is the world's finest imperative programming language.")

- Declarative concepts are particularly well suited for realising/embedding domain specific languages (DSLs).

- But:
  - Declarative languages are still less widespread than imperative languages.
  - Development tools like IDEs etc. for working with declarative languages are often lacking (in quantity or quality).
  - Limitations to apply declarative languages are often based on (assumptions about) not sufficiently efficient execution/operationalisation.

Declarative programming "in the real world"

- Commercial users:
  - in banking sector (trading, quantitative analysis), e.g., Barclays Capital, Jane Street Capital, Standard Chartered Bank, McGraw Hill Financial, …
  - in communication/web services, e.g., Ericsson, Facebook, Google
  - hardware design/verification, e.g., Intel, Bluespec, Antiope
  - system-level development, e.g., Microsoft
  - high assurance software, e.g., Galois

http://cufp.org/

http://groups.google.co.uk/group/cu-lp

- "Non-academic" languages:
  - for special applications, e.g., Erlang (Ericsson), reFLect (Intel)
  - for general purposes, e.g., F# (Microsoft)
  - influence on mainstream languages, e.g., Java, C#, and "even" Visual Basic (generally: LINQ framework)

Important functional languages in historic overview

LISP — 1959, McCarthy (MIT/USA)

1960

1970

Scheme — 1975, Sussman/Steele (MIT/USA)

1980

ML — 1980, Milner (Edinburgh/GB)

Miranda — 1985, Turner (Canterbury/GB)

Haskell — 1987, Hudak/Wadler (Yale/USA)

1990

Haskell 98

http://haskell.org

2000

"A History of Haskell: Being Lazy with Class"
(P. Hudak, J. Hughes, S. Peyton Jones, P. Wadler)

What does the name "Haskell" stand for?

- Programming languages are often named via acronyms (e.g., COBOL, FORTRAN, BASIC, …)
- But the name "Haskell" is derived from a person:

Haskell Brooks Curry
(1900 – 1982)

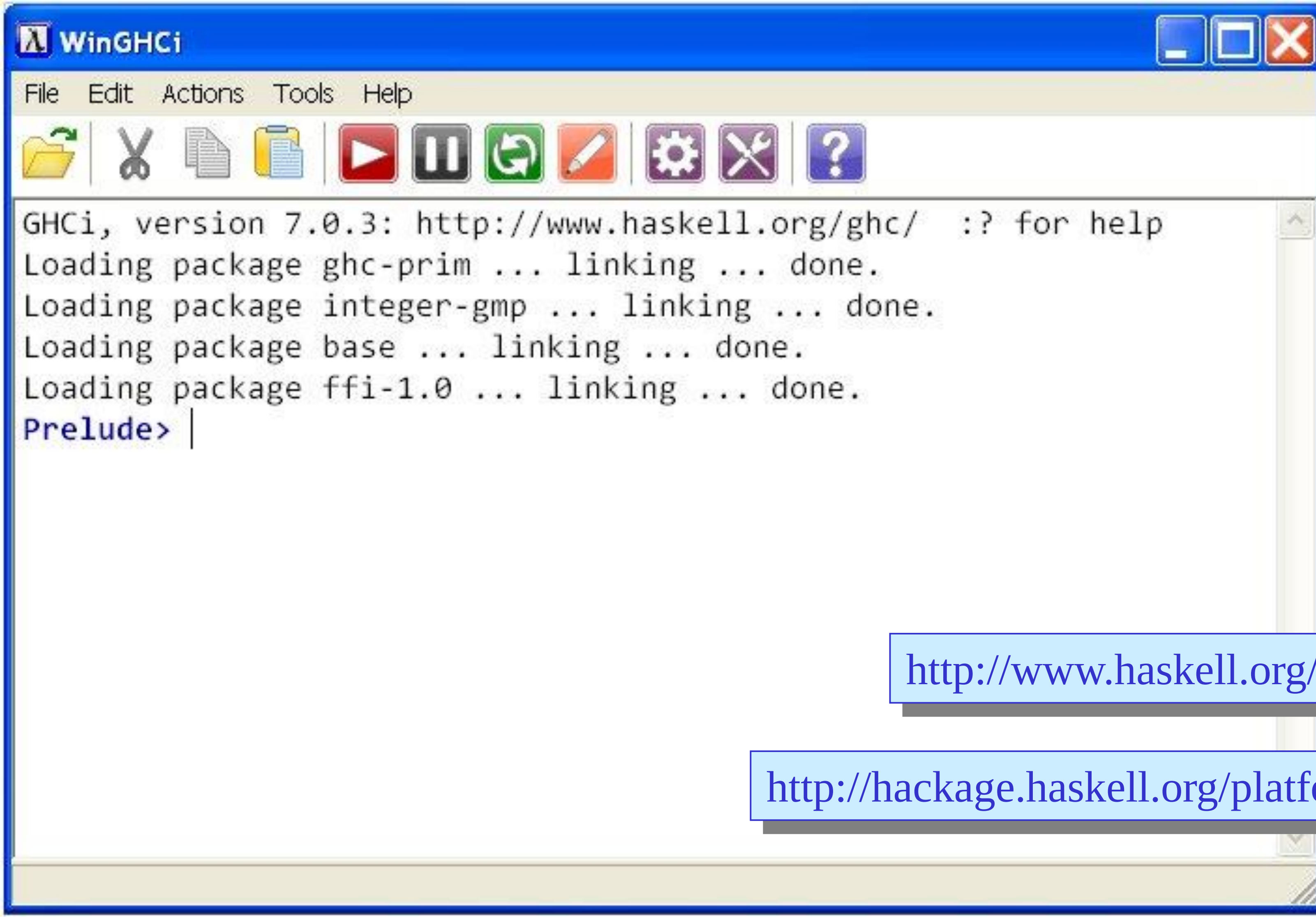
American logician

Literature on Haskell

- Text books (for example):
  - R. Bird:
    Introduction to Functional Programming using Haskell
    Prentice Hall, 1998
  - M. Block, A. Neumann:
    Haskell-Intensivkurs
    Springer-Verlag, 2011
  - P. Hudak:
    The Haskell School of Expression
    Cambridge University Press, 2000
  - G. Hutton:
    Programming in Haskell
    Cambridge University Press, 2007
  - S. Thompson:
    Haskell – The Craft of Functional Programming
    Addison Wesley, 2011

- Introductory article:
  P. Hudak, J. Peterson, J. Fasel: A Gentle Introduction to Haskell
  (haskell.org, 1999)

The implementation we are going to use: GHC(i)



http://www.haskell.org/ghc/

http://hackage.haskell.org/platform/

Programming Paradigms

Examples of DSLs embedded in Haskell

Parsing and dealing with arithmetic expressions

- Suppose we want to compile arithmetic expressions into "machine code", for example thus:

"2+3*5"	$\mapsto$	"LIT 2; LIT 3; LIT 5; MUL; ADD; "
"2*3+5"	$\mapsto$	"LIT 2; LIT 3; MUL; LIT 5; ADD; "

- First we need to describe the structure of (valid) expressions.

- For example by means of a formal grammar (say, a BNF):

```
⟨Expr⟩   ::= ⟨Term⟩ '+' ⟨Expr⟩ | ⟨Term⟩
⟨Term⟩   ::= ⟨Factor⟩ '*' ⟨Term⟩ | ⟨Factor⟩
⟨Factor⟩ ::= ⟨Nat⟩ | '(' ⟨Expr⟩ ')'
```

- … and now we could (in a "conventional" programming language) develop/implement an algorithm for parsing according to this grammar (or any grammar).

Parsing and dealing with arithmetic expressions

- It would be more attractive to use the available specification

```
⟨Expr⟩   ::= ⟨Term⟩ ‘+’ ⟨Expr⟩ | ⟨Term⟩
⟨Term⟩   ::= ⟨Factor⟩ ‘*’ ⟨Term⟩ | ⟨Factor⟩
⟨Factor⟩ ::= ⟨Nat⟩ | ‘(’ ⟨Expr⟩ ‘)’
```

and consider it, as directly as possible, as a “program” itself.

- Actually quite close:

```
expr   =  ( ADD  <$>  term  <*  char '+'  <*>  expr )  |||  term

term   =  ( MUL  <$>  factor  <*  char '*'  <*>  term )  |||  factor

factor  =  ( LIT  <$>  nat )  |||  ( char '('  *>  expr  <*  char ')' )
```

- Trying out:

```
>  parse expr "2*3+5"
ADD (MUL (LIT 2) (LIT 3)) (LIT 5)
```

Parsing and dealing with arithmetic expressions

- To get the actually desired output:

```
data Expr  =  LIT Int  |  ADD Expr Expr  |  MUL Expr Expr

instance Show Expr where
  show (LIT n)          =  "LIT "  ++  show n  ++  "; "
  show (ADD e1 e2)  =  show e1  ++  show e2  ++  "ADD; "
  show (MUL e1 e2)  =  show e1  ++  show e2  ++  "MUL; "
```

- Then indeed:

```
>  parse expr "2*3+5"
LIT 2; LIT 3; MUL; LIT 5; ADD;
```

- Alternatively, also possible to, e.g., directly compute the result:

```
eval  (LIT n)          =  n
eval  (ADD e1 e2)  =  eval e1  +  eval e2
eval  (MUL e1 e2)  =  eval e1  *  eval e2
```

Parsing and dealing with arithmetic expressions

- Alternatively, also possible to, e.g., directly compute the result:

```
eval  (LIT n)          =  n
eval  (ADD e1 e2)  =  eval e1  +  eval e2
eval  (MUL e1 e2)  =  eval e1  *  eval e2
```

- Then, for example:

```
>  eval  (parse expr "2*3+5")
11
```

- Or even evaluation directly while parsing:

```
expr   =  ( (+)  <$>  term  <*  char '+'  <*>  expr )  | | |  term

term    =  ( (*)  <$>  factor  <*  char '*'  <*>  term )  | | |  factor

factor  =  nat  | | |  ( char '('  *>  expr  <*  char ')' )
```

- Since then:

```
>  parse expr "2*3+5"
11
```

Another domain: describing graphics with "gloss"

- A simple library (install instructions will be given with exercises).

- Basic concepts:

```
Float, String, Path, Color, Picture

text      ::  String → Picture
line      ::  Path → Picture
polygon   ::  Path → Picture
arc       ::  Float → Float → Float → Picture
circle    ::  Float → Picture
…

color     ::  Color → Picture → Picture
translate ::  Float → Float → Picture → Picture
rotate    ::  Float → Picture → Picture
scale     ::  Float → Float → Picture → Picture

pictures  ::  [ Picture ] → Picture
```

Another domain: describing graphics with “gloss”

- Use in a concrete “program”:

```
module Main (main) where

import Graphics.Gloss

main = display (InWindow "Example" (100, 100) (0, 0)) white scene

scene = pictures
        [
          circleSolid 20
        , translate 25 0 (color red (polygon [(0, 0), (10, –5), (10, 5)]))
        ]
```

- Let’s play a bit. …

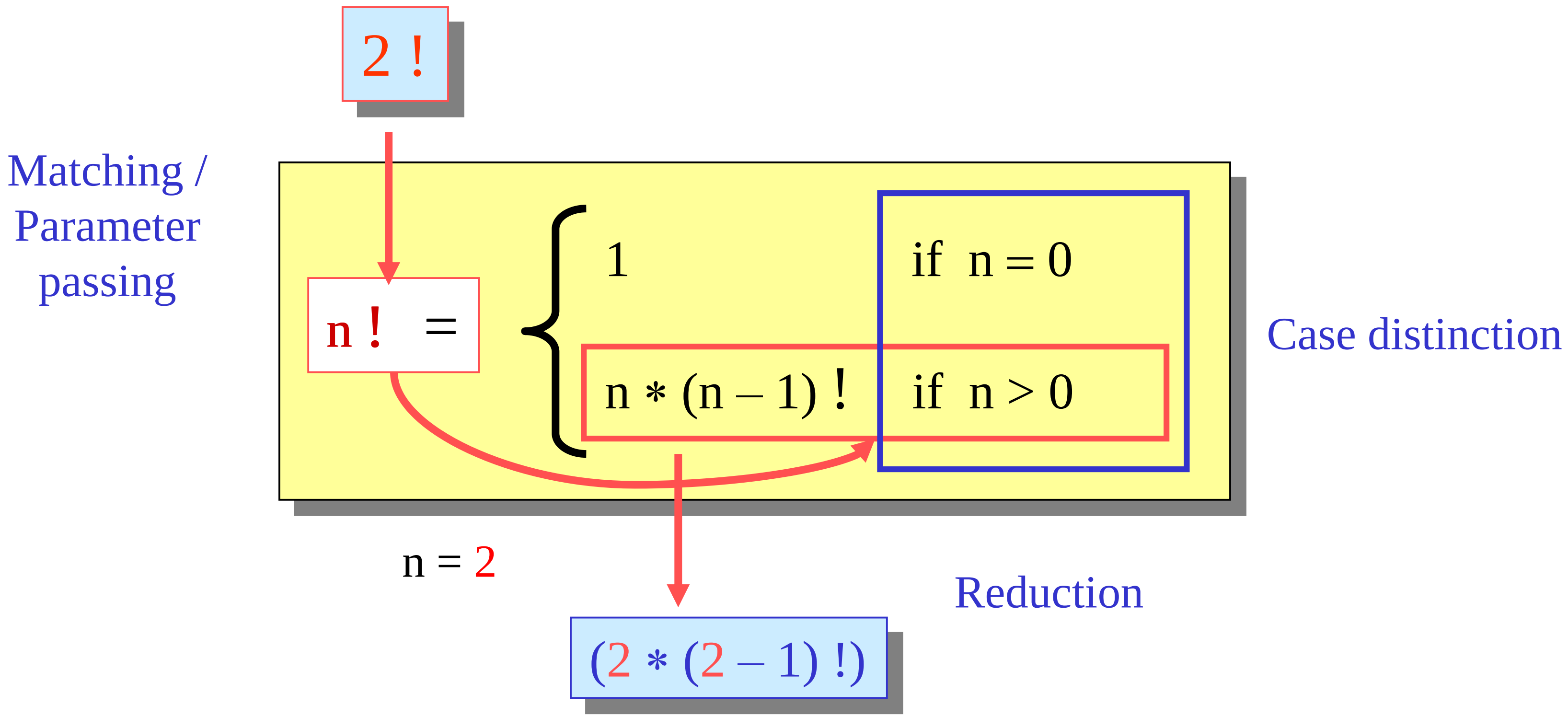
Programming Paradigms

Haskell Basics/Syntax

The principle of functional programming

Specifications: Function definitions

Operationalisation: Evaluation of expressions (syntactic reduction)



The principle of functional programming

> *"Let the symbols do the work."*
> Leibniz/Dijkstra

Specification ("program") ≡
Function definition(s)

$$n\,! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)\,! & \text{if } n > 0 \end{cases}$$

predefined operators

$2\,!$
$\Rightarrow (2 * (2 - 1)\,!)$
$\Rightarrow (2 * 1!\,)$
$\Rightarrow (2 * (1 * (1 - 1)\,!))$
$\Rightarrow (2 * (1 * 0\,!))$
$\Rightarrow (2 * (1 * 1))$
$\Rightarrow (2 * 1)$
$\Rightarrow 2$

Input: term/expression to be evaluated

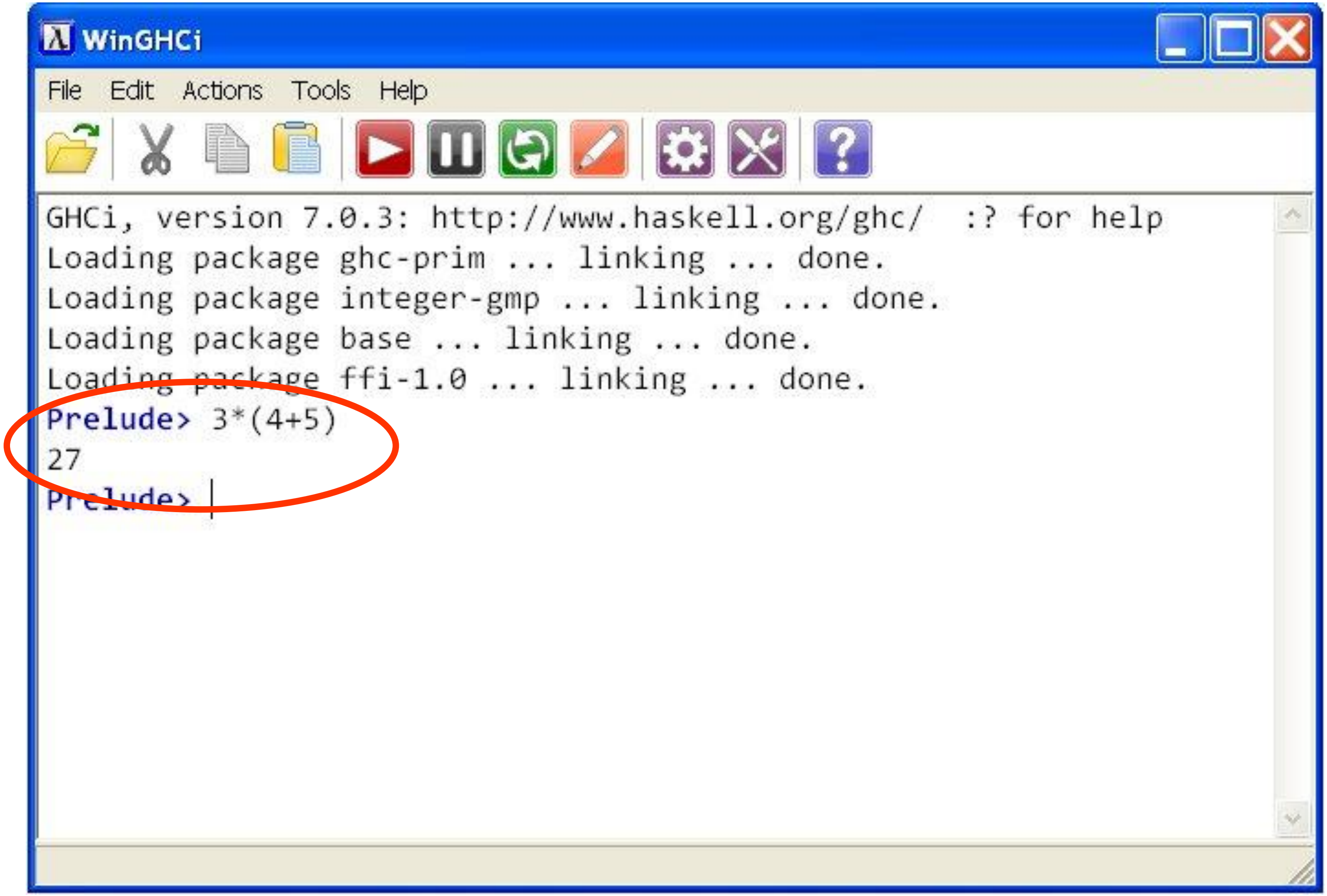
⬇

(repeated) function application

⬇

Output: resulting value

GHCi as simple calculator



Basic types, operators and functions

- Int, Integer:
  - the integer numbers (–12, 0, 42, ...)
  - operators: +, –, *, ^
  - functions: div, mod, min, max, ...
  - comparisons: ==, /=, <, <=, >, >=
- Float, Double:
  - the floating point numbers (–3.7, pi, ...)
  - operators: +, –, *, /
  - functions: sqrt, log, sin, min, max, ...
  - comparisons: …
- Bool:
  - the Boolean values (True, False)
  - operators: &&, ||
  - functions: not; comparisons: …
- Char:
  - individual characters ('a', 'b', '\n', ...)
  - functions: succ, pred; comparisons: …

Evaluating simple expressions

```
> 5+7
12
```

```
> div 17 3
5
```

not: div(17,3) instead:

```
> 17 `div` 3
5
```

```
> pi/1.5
2.0943951023932
```

```
> min (sqrt 4.5) (1.5^3)
2.1213203435596
```

not: min(sqrt(4.5),1.5^3)

```
> 'a' <= 'c'
True
```

```
> if 12 < 3 || 17.5 /= sqrt 5 then 17 – 3 else 6
14
```

never without else-branch!

More complex types, expressions and values

- Lists:
  - [Int] for [] or [–12, 0, 42]
  - [Bool] for [] or [False, True, False]
  - [[Int]] for [[3, 4], [], [6, –2]]
  - …
  - operators: :, ++, !!
  - functions: head, tail, last, null, …
- Character sequences:
  - String = [Char]
  - special notation: "" for [] and "abcd" for ['a', 'b', 'c', 'd']
- Tuples:
  - (Int, Int) for (3, 5) and (0, –4)
  - (Int, String, Bool) for (3, "abc", False)
  - ((Int, Int), Bool, [Int]) for ((0, –4), True, [1, 2, 3])
  - [(Bool, Int)] for [(False, 3), (True, –4), (True, 42)]
  - …
  - functions: fst and snd on pairs

```
> 3 : [–12, 0, 42]
[3, –12, 0, 42]
```

```
> [1.5, 3.7] ++ [4.5, 2.3]
[1.5, 3.7, 4.5, 2.3]
```

```
> [False, True, False] !! 1
True
```

```
> (3 – 4, snd (head [('a', 17), ('c', 3)]))
(–1, 17)
```

Declaration of values

- In a file:

```
x = 7
y = 2 * x
z = (mod y (x + 2), tail [1 .. y])

a = b – c
b = fst z
c = head (snd z)

d = (a, e)
e = [fst d, f]
f = head e
```

All these are declarations, not value-changing assignments!

```
x = x + 1
```

makes no sense!

- After loading:

```
> z
(5, [2,3,4,5,6,7,8,9,10,11,12,13,14])
```

```
> a
3
```

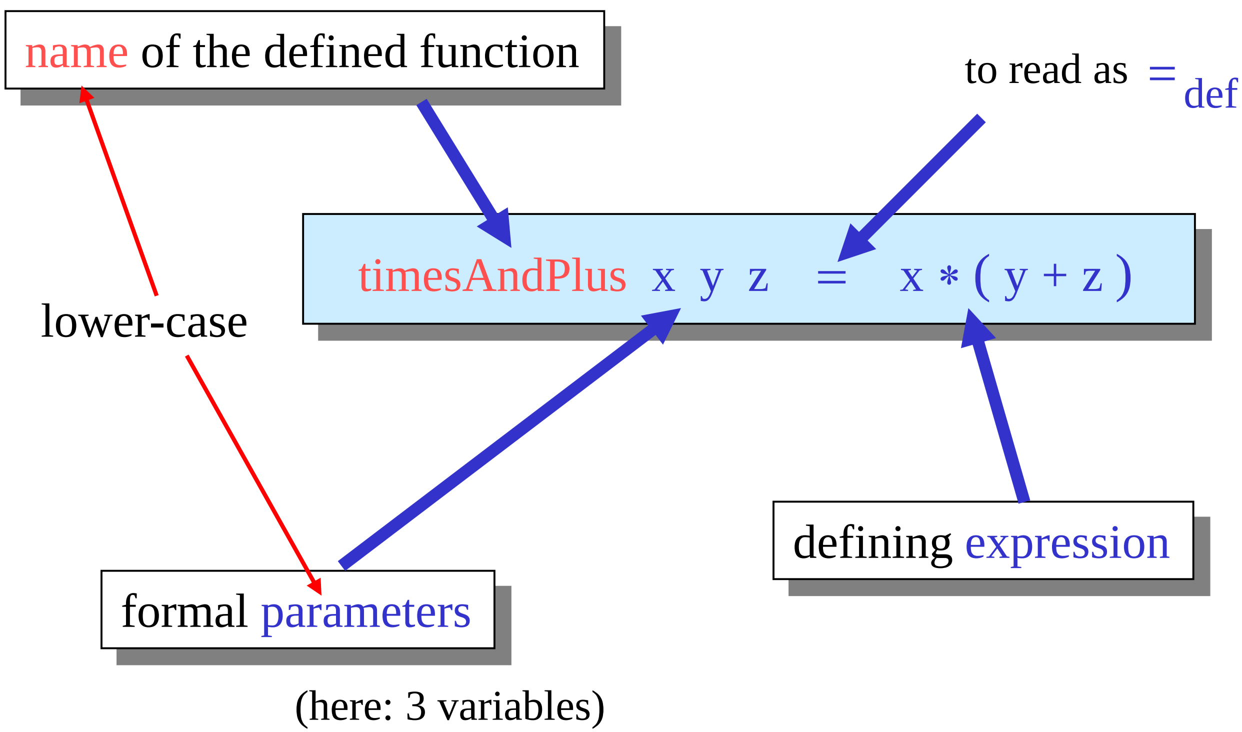
```
> d
(3, [3, 3])
```

Optional type annotations

```
x , y :: Int
x = 7
y = 2 * x

z :: (Int, [Int])
z = (mod y (x + 2), tail [1 .. y])

a , b , c :: Int
a = b – c
b = fst z
c = head (snd z)

d :: (Int, [Int])
d = (a, e)


...
```

Function definitions in Haskell

General form of a (very simple) function definition:



Declaration of functions (with type annotations)

Recall: "if-then" in Haskell always with explicit "else"!

```
min3 :: Int → Int → Int → Int
min3 x y z  =  if x<y then (if x<z then x else z)
                      else (if y<z then y else z)
```

```
> min3 5 4 6
4
```

```
min3' :: (Int, Int, Int) → Int
min3' (x, y, z)  =  if x<y then (if x<z then x else z)
                           else (if y<z then y else z)
```

```
> min3' (5, 4, 6)
4
```

```
min3" :: Int → Int → Int → Int
min3" x y z  =  min (min x y) z
```

```
> min3" 5 4 6
4
```

```
isEven :: Int → Bool
isEven n  =  (n `mod` 2) == 0
```

```
> isEven 12
True
```

equality test!

Examples: syntax for function application

Math-like	Haskell-like
f(x)	f x
f(x,y)	f x y
f(g(x))	f (g x)
f(x,g(y))	f x (g y)
f(x) + g(y)	f x + g y
f(a+b)	f (a + b)
f(a) + b	f a + b

More on syntax of function definitions

- On the left side of a defining equation in Haskell,
  - no expressions still to be evaluated, but …
  - only variables and constants (and patterns, see later …)

  may occur:

```
f x (2 * y) = x * y
```

not allowed!

```
f x 1 = x * 2
```

okay

- On the right side of a defining equation,
  - arbitrary expressions, also ones still to be evaluated, but …
  - only variables from the left side (so no “fresh” variables)

  may occur:

```
f x = x * y
```

not allowed!

```
f x 1 = x * 2
```

okay

More on syntax of function definitions

- In the list of formal parameters of a function definition, every variable must appear only exactly once:

```
f n 0 n = n^2
```

not allowed!

instead:

```
f n 0 m | n == m = n^2
```

Function definitions: distinguishing cases (1)

More complex function definitions are build from several alternatives.
Each alternative defines one case of the function:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{if } n > 0 \end{cases}$$

In Haskell, possible like so:

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

But the “mathematical” style can also be imitated in Haskell, though the conditions are placed before the equation sign:

```
fac n      | n == 0    =   1
           | n  >  0   =   n * fac (n – 1)
```

Function definitions: distinguishing cases (2)

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{if } n > 0 \end{cases}$$

```
fac n   | n == 0   = 1
        | n >  0   = n * fac (n – 1)
```

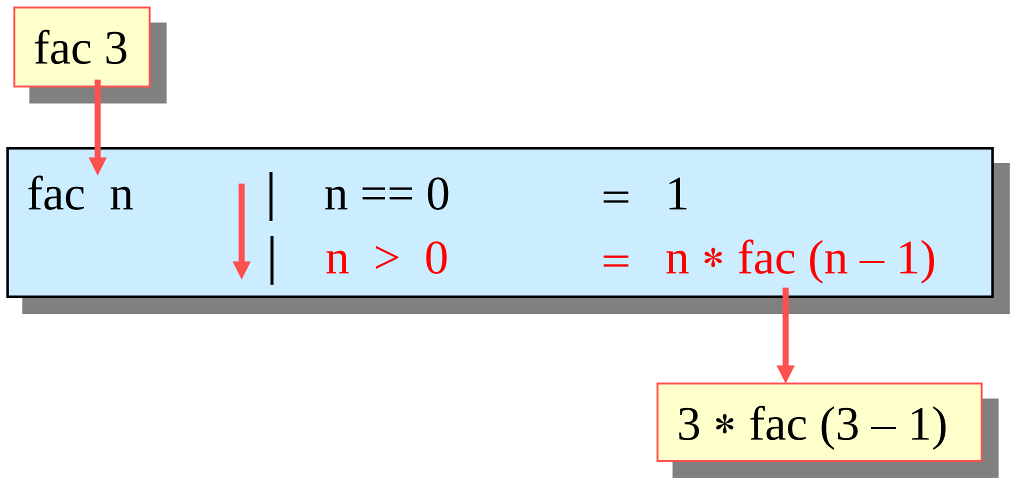
"Guards"

Boolean expressions

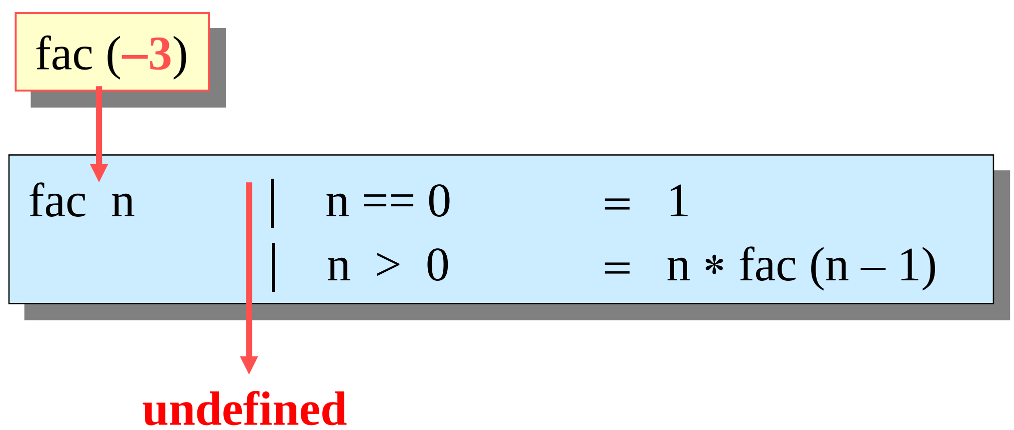
As in the mathematical notation, the guards are checked from top to bottom, until the first time a condition is satisfied.

That case is then used for reduction/continuing evaluation.

Function definitions: distinguishing cases (3)



The factorial function is only partially defined: for negative input parameters, no “matching” case is found, so the result is undefined.



Function definitions: distinguishing cases (4)

Changing into a totally defined function by adding a “catch all” case using the pseudo condition otherwise:

fac (–3)

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

0

Sometimes also helpful for abbreviation:

```
between  x y z      | (x <= y)  &&  (y <= z)      =  True
                    | (y < x) || (y > z)          =  False
```

```
between  x y z      | (x <= y)  &&  (y <= z)      =  True
                    | otherwise                   =  False
```

Function definitions: distinguishing cases (5)

Variations:

```
fac n    | n == 0   =  1
         | n > 0    =  n * fac (n – 1)
```

is essentially only an abbreviation for:

```
fac n    | n == 0   =  1
fac n    | n > 0    =  n * fac (n – 1)
```

Yet another notation variant, in which the first condition is expressed through a constant parameter:

```
fac 0               =  1
fac n    | n > 0    =  n * fac (n – 1)
```

Function definitions: distinguishing cases (6)

- Apparently an important basic technique:
  selection of a "matching" definition case for a function application
  to be evaluated

- Two selection criteria (in this order!):
  - "pattern matching" (to be considered in a bit more detail next)
  - evaluation of guard conditions

```
(1)   ack  0   n    | n >= 0              =  n + 1
(2)   ack  m   0    | m > 0               =  ack  (m – 1)  1
(3)   ack  m   n    | n  > 0  &&  m > 0   =  ack  (m – 1)  (ack  m  (n – 1))
```

Ackermann function

```
ack  0  0      matches (1)
ack  2  0      matches (2)
ack  2  1      matches (3)
```

Order in going through cases in function definitions

- When evaluating the application ack 0 0 all three left sides would match!

```
ack 0 n  | n >= 0          = n + 1
ack m 0  | m > 0           = ack (m – 1) 1
ack m n  | n > 0 && m > 0  = ack (m – 1) (ack m (n – 1))
```

- The actually defining case is the first matching one (going from top to bottom), whose guard is satisfied.

- In this way it is ensured that there is always a unique function result. (… if there is one at all!)

- For the above Ackermann function, every order of the three equations gives the same behaviour. But that is not always so! fac 0 behaves differently here:

```
fac 0 = 1
fac n = n * fac (n – 1)
```

} 1

```
fac n = n * fac (n – 1)
fac 0 = 1
```

} undefined

Pattern matching: principle

concrete application `> ack 0 (ack 2 1)`

pattern matching

left side of a definition `ack 0 n | ... = ...`

Rules of pattern matching:

- prerequisite: identical function name
- constants match
  - themselves (e.g., 1 ↔ 1)
  - every variable (e.g., 1 ↔ n)
- complex expressions match
  - every variable (e.g., (fib 3) ↔ x)
  - the specific constant that denotes their function result
    (e.g., (fib 4) ↔ 5)

    enforces evaluation!
- tuples match
  - every variable, and tuples of same length if components match as well
    (e.g., (1, False, fib 4) ↔ (1, x, 5))
- …