UNIVERSITÄT
DUISBURG
ESSEN

Open-Minded

Programming Paradigms

all slides (Haskell and Prolog) ■ version: 13.07.2021, 11:42:26 +00:00

UNIVERSITÄT
DUISBURG
ESSEN

Open-Minded

Programming Paradigms – Haskell part

Summer Term 2021

UNIVERSITÄT
DUISBURG
ESSEN

Open-Minded

Organisation

About us

- **Lecturer:**
  Prof. Janis Voigtländer, Room LF 233
  Area: Formal Methods, Programming Languages
- **Teaching Assistants:**
  Marcellus Siegburg, Room LF 232
  Oliver Westphal, Room LF 232
- **Student Assistants:**
  Ashraf Hashash
  Leon Koth
  Daniel Laybourn
  Patrick Ritzenfeld

About you

To my knowledge, mainly:

1. **Bachelor Students "Angewandte Informatik"**
2. **Bachelor Students "Computer Engineering (ISE)"**

Some (relevant) lectures you have presumably attended:

- **Grundlegende Programmiertechniken (or ISE equival.)**
- **Fortgeschrittene Programmiertechniken (?)**
- **Logik (?)**
- **Softwaretechnik (?)**

This lecture

- **Weekly slot:**
  Wednesday, 08:30 – 10:00, in LB 131
  up to 15 times this term
- **Slides to be updated, potentially for/after each lecture.**
- **There will be other material as well.**
- **We use the standard UDE Moodle system.**

The exercises

Groups:

- **Mon, 12:00 – 14:00, LF 035**
- **Mon, 16:00 – 18:00, LE 105**
- **Tue, 10:00 – 12:00, LE 120**
- **Thu, 10:00 – 12:00, LC 137**
- **Thu, 12:00 – 14:00, LK 051**
- **Fri, 08:00 – 10:00, LE 120**

Starting the day after the second lecture.

Role of the exercises

- **You can earn bonus points for the exam taking place at the end of this semester (only).**
- **But mainly, doing the exercises (on your own) is important to be successful in the course at all.**

- **In particular, there will be a focus on programming concepts and tasks. Such material cannot be learned by heart. It needs practice!**
- **Not all tasks each week will be mandatory/contributing to earning the exam bonus. But you are advised to work on all tasks, and to go to the exercise sessions.**

About our use of Moodle

Moodle for:

- **Lecture slides and material**
- **Access to Notabene**
- **Exercise tasks (though mostly in Autotool)**
- **Announcements**
- **Question forum**
- **Submission of (only a few) exercise solutions**

Additionally: CodeWorld and Autotool

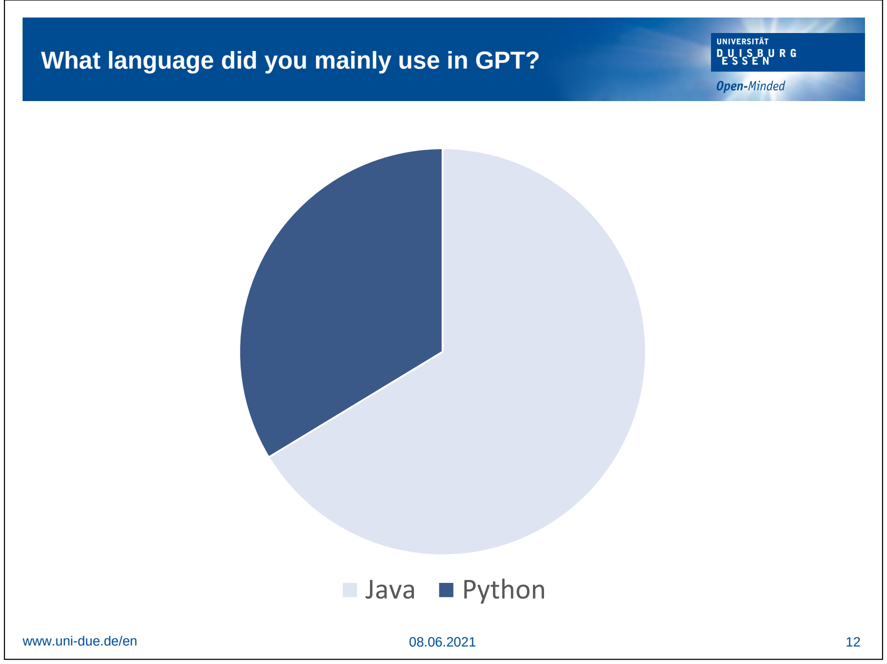
Concerning communication

- In most cases, asking a question in the Moodle forum or in Notabene makes more sense than sending an individual email to me or to the assistants/tutors.
- If emails are sent at all, they should come from your uni-due.de accounts (or may simply be ignored).
- On the other hand, you should check your uni-due.de an least once per (work) day.
- If something was already answered in the slides or in the Moodle forum, there is no promise of a second, third, … answer.
- Generally: announced deadlines are strict.

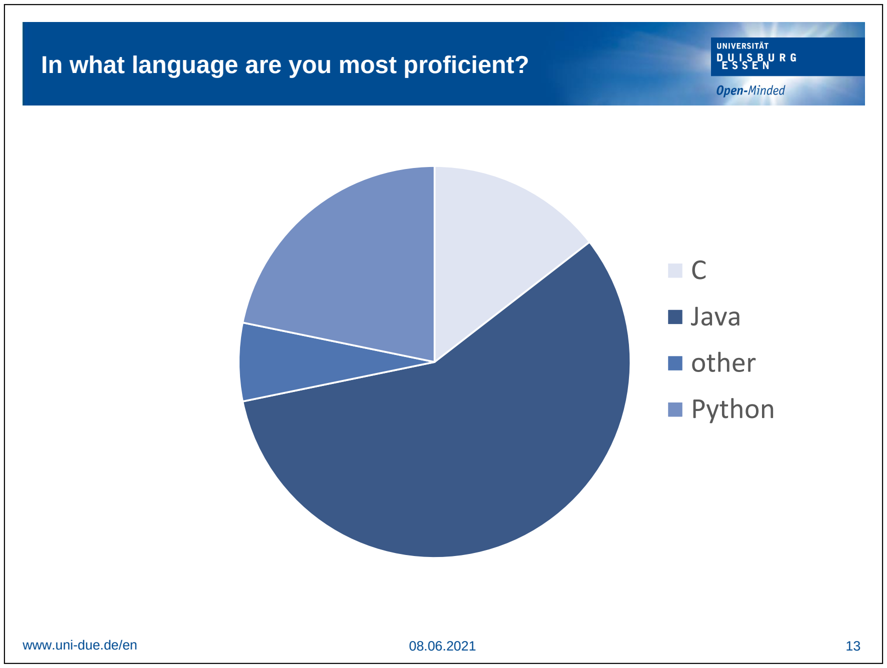
Eventually, the exam

- There will be an exam, possibly written, possibly online. Timing is not yet clear.
- Registration for the exam will be via the exam office (in the usual way you know from other courses, presumably online).
- The exam will be about the course as taught this term!
- The exam tasks will be given (only) in English.

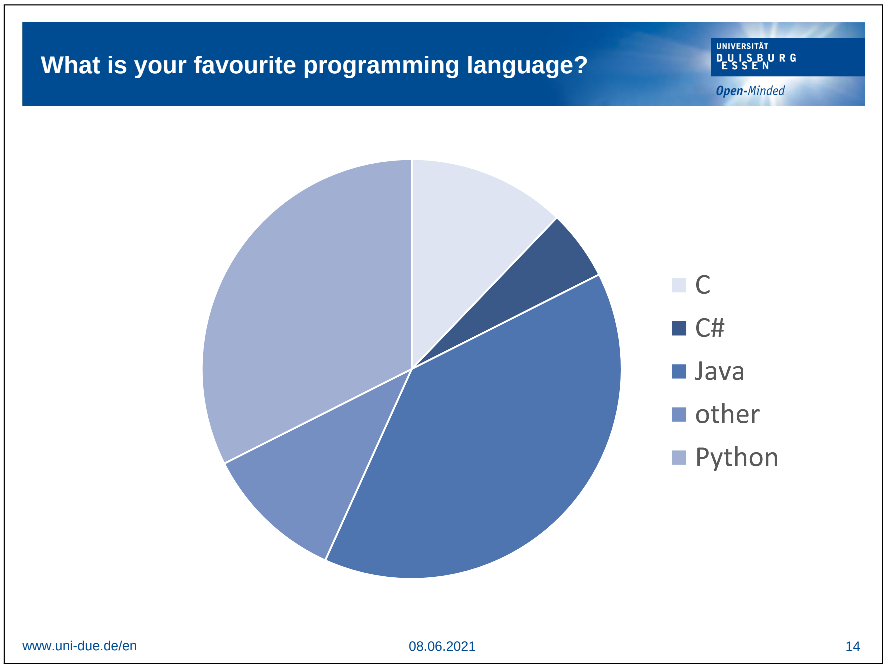
From a Moodle survey in 2019



What language did you mainly use in GPT?



In what language are you most proficient?

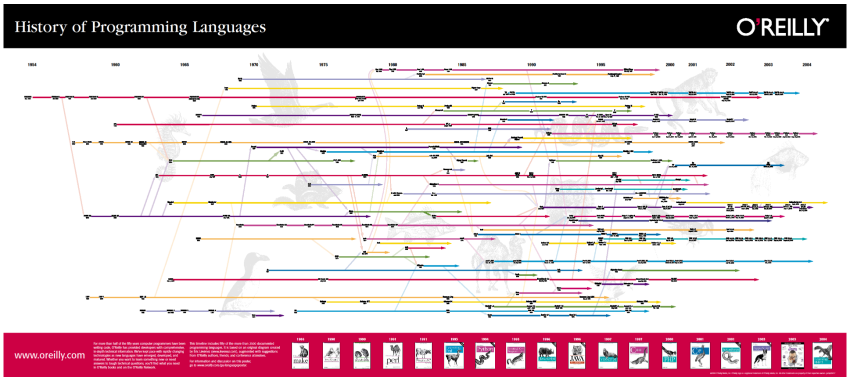


What is your favourite programming language?

Introduction / Motivation

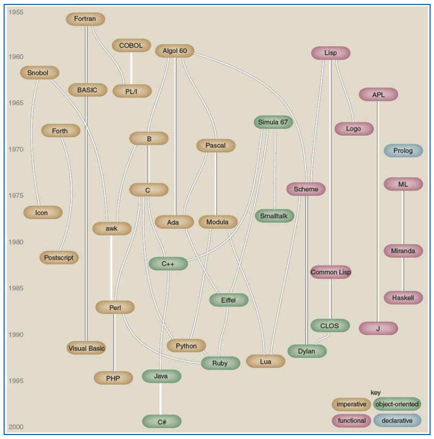
“To know another language is to have a second soul.”
Charlemagne, 747/748 – 814

Many high-level programming languages in existence



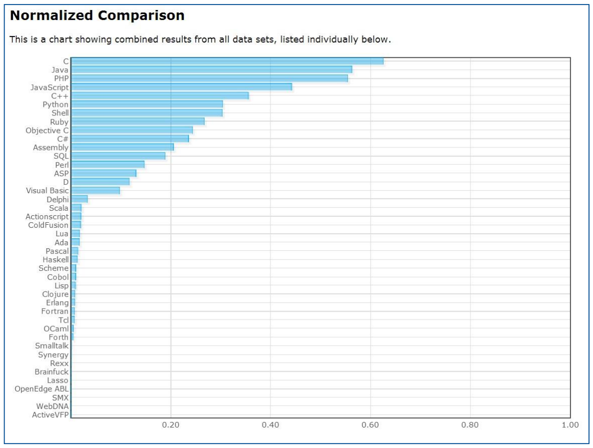
© 2004 O'Reilly Verlag GmbH & Co. KG

Another perspective



From “American Scientist”: The Semicolon Wars, © 2006 Brian Hayes

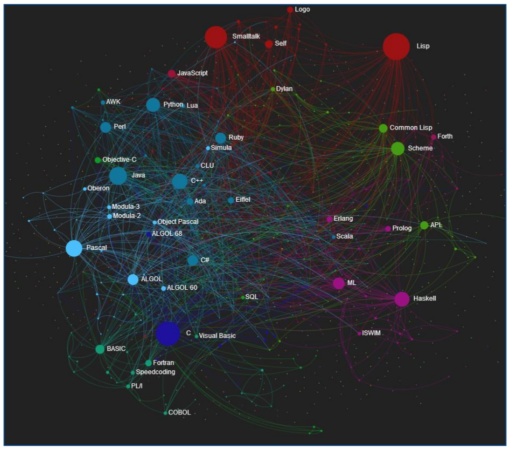
Also, popularity contests, …


Normalized Comparison

This is a chart showing combined results from all data sets, listed individually below.

http://preview.tinyurl.com/popular-languages

And yet another visualization



http://preview.tinyurl.com/language-influences

So, why such diversity?

- **Can one (or each) language do “more” than others?**
- **Are there problems that one cannot solve in certain languages?**
- **Is there a “best” language? At least for a certain purpose or application area?**
- **What does actually separate different programming languages from each other?**

So, why such diversity?

Some relevant distinctions:

- syntactically rich vs. syntactically scarce (e.g., APL vs. Lisp)
- verbosity vs. succinctness (e.g., COBOL vs. Haskell)
- compiled vs. interpreted (e.g., C vs. Perl)
- domain-specific vs. general purpose (e.g., SQL vs. Java)
- sequential vs. concurrent/parallel (e.g., JavaScript vs. Erlang)
- typed vs. untyped (e.g., Haskell vs. Prolog)
- dynamic vs. static (e.g., Ruby vs. ML)
- declarative vs. imperative (e.g., Prolog vs. C)
- object-oriented vs. ???
- …

And, yet, there are common principles

Approaches to the specification of programming languages

- … describing syntax,
- … describing semantics,

as well as implementation strategies.

Language concepts:

- variables and bindings
- type constructs
- control structures and abstraction features

And, of course, paradigms that span a whole class of languages.

A rough plan of the lecture

- We will focus on two paradigms: functional and logic programming.
- For each, we pick a specific language: Haskell, Prolog.
- We consider actual programming concepts, and also aspects related to semantics (evaluation, resolution).
- With Haskell, we explore typing concepts like inference, genericity, polymorphism.
- We discuss and compare concepts like variables, expressions vs. commands, etc., in different languages.

Declarative programming

- Functional and logic programming are often called “declarative” or “descriptive” programming.
- The idea is that programmers can think more in terms of “What?” instead of “How?”, in other words, more in terms of specification than planning a certain computation process.
- Of course, there is still a need for algorithmic thinking etc., as there is no magic.
- But it is true that declarative programming has a more high-level, sometimes mathematical, feel to it.
- Also, the “What-instead-of-How” aspect will become concrete with observations like the roles of expressions vs. commands in different languages/paradigms.
- A side benefit in declarative languages is often reduced syntax.

Other reasons for studying “new” paradigms

- Learning different languages now makes it easier to pick up new languages later on.
- Concepts from once “exotic” languages make their way into “mainstream” ones.
- In some application domains, there is an increased demand for very disciplined, conceptually expressive, mathematics-based languages.
- Generally, knowing more paradigms increases capacity to express ideas.

Literature

Books on Haskell

- **Programming in Haskell, 2nd edition; Graham Hutton**
- **Haskell – The Craft of Functional Programming, 3rd edition; Simon Thompson**
- **Thinking Functionally with Haskell; Richard Bird**
- **Haskell-Intensivkurs; Marco Block, Adrian Neumann**
- **Einführung in die Programmierung mit Haskell; Manuel Chakravarty, Gabriele Keller**

Books on Prolog

- **Learn Prolog Now!; Patrick Blackburn, Johan Bos, Kristina Striegnitz**
- **Programmieren in Prolog; William Clocksin, Christopher Mellish**
- **Prolog – Verstehen und Anwenden; Armin Ertl**

A first glimpse of FP with CodeWorld

A first complete animation program

```
import CodeWorld

main = animationOf scene

scene t =
   circle 8
 & colored green (solidRectangle 4 4)
 & rotated (pi/2)
   (translated 8 0 (colored red (polygon [(0,0),(1,-0.5),(1,0.5)])))
 & pictures
   [ rotated ((a+t)*pi/20)
     (rectangle (4+a) (4+a)) | a <- [ 0, 0.5 .. 9 ] ]
```

Expressions vs. commands

Expression-based programming

- **Proposition:**
  Functional programming is about expressions, whereas imperative programming is about commands.
- **Some kinds of expressions you (probably) know:**

$$\mathbf{2+3\cdot(x+1)^2}$$

$$\boldsymbol{p\wedge\neg(q\vee r)}$$

SUMIF(A1:A8,"<0")

- **Generally: terms in any algebra, built from constants and functions/operators, possibly containing variables**

Properties of (pure) expressions

Expressions

- … are compositional, built completely from subexpressions,
- … often have a meaningful type,
- … have a value, which does not depend on “hidden influences”, and does not change on re-evaluation or based on the order of evaluating subexpressions.

The compositionality is not just syntactical (expressions are built from subexpressions), but extends to typing and semantics/evaluation.

Properties of (pure) expressions

Example $2 + 3 \cdot (x + 1)^2$:

The constants are 1, 2, 3 of type $\mathbb{Z}$.

The operators are $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, $\cdot : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, $()^2 : \mathbb{Z} \to \mathbb{Z}$.

The value of $2 + 3 \cdot (x + 1)^2$ depends only on the value of 2 and the value of $3 \cdot (x + 1)^2$, the latter only depends on the value of 3 and the value of $(x + 1)^2$, …

Properties of (pure) expressions

- Thanks to these properties, we can easily use notation known from mathematics, for example reformulating “$2 + 3 \cdot (x + 1)^2$” as follows:
  “$2 + 3 \cdot y^2$ where $y = x + 1$”.
- Also, we can apply simplifications, for example replacing exponentiation by multiplication:
  “$2 + 3 \cdot y \cdot y$ where $y = x + 1$”.

- And while this example was about arithmetic expressions, the concepts apply much more generally.
- But only if we have pure expressions!

The situation in imperative programming languages

- So what is different in imperative programming?
- Don’t we also have expressions there?
  For example in:

```
b = 100000;
if (z > 0) {
  z = 100 + z;
  j = 0;
  while (b < 200000) {
    b = b * z / 100;
    j = j + 1; }
} else j = -1;
```

- Yes, there are expressions, but they are not the dominating syntactical construct. Commands are!

The situation in imperative programming languages

- Why is this difference relevant? What properties do commands, as opposed to expressions, not have?
- Well, for example, they are not even syntactically compositional: Not every well-formed smaller part of a command is itself a command.

```
while (b < 200000) {
  b = b * z / 100;
  j = j + 1;
}
```

- Instead, expressions occur, also keywords, …
- Moreover, commands do not always have a meaningful type.
- Or even just a value. (Try giving a value for the above block.)

The situation in imperative programming languages

- As a consequence, we cannot name arbitrary well-formed smaller parts (as opposed to what we saw for expressions and their subexpressions).
- For example, we cannot simply write:

```
body = {
  b = b * z / 100;
  j = j + 1;
}
while (b < 200000) body;
```

- Even workarounds involving functions/procedures/methods are not as flexible and useful as the kind of mathematical notation for expressions: “$2 + 3 \cdot y^2$ where $y = x + 1$”.

The situation in imperative programming languages

- Okay, so what about the sublanguage of expressions in an imperative language? Can they, at least, be treated as we saw before?
- Not in general! For example, we saw that mathematically we should be able to rewrite something like “$\boldsymbol{exp_1 + exp_2 \cdot (exp_3)^2}$” as any of:

$$\begin{array}{ll} exp_1 + exp_2 \cdot var^2 & \text{where } var = exp_3 \\ exp_1 + exp_2 \cdot var \cdot var & \text{where } var = exp_3 \\ exp_1 + exp_2 \cdot exp_3 \cdot exp_3 & \end{array}$$

- But code snippets like “$\texttt{result = exp}_1\texttt{ + exp}_2\texttt{ * (exp}_3\texttt{)^2;}$” do not always take well to being replaced by:

```
var = exp3; result = exp1 + exp2 * var^2;
```

- … or by code snippets corresponding to the other expression alternatives above.

The situation in imperative programming languages

- Indeed, consider these four code snippets:

```
result = exp1 + exp2 * (exp3)^2;
var = exp3; result = exp1 + exp2 * var^2;
var = exp3; result = exp1 + exp2 * var * var;
result = exp1 + exp2 * exp3 * exp3;
```

- And imagine instantiations with $\texttt{exp}_3$ being the “expression” `i++` or some invocation `f()` for a procedure/method `f`.
- The problem is that expressions in an imperative language are typically not pure expressions. Instead, they have side-effects!
- (For same reason, re-evaluation of an expression can change the value. And order of evaluating subexpressions becomes relevant.)

So what?

- So, how “bad” is all that?
- Do these artificial examples “prove” anything?
- Well, I haven’t (yet?) really argued that the pure expression-based style is better in some sense.
- But what should have become clear is that it is different!
- In any case, let us (again) “do” something with CodeWorld. (… also in your first exercise tasks)

Another look at FP with CodeWorld

Describing a picture via an expression

A rather simple example:

```
main :: IO ()
main = drawingOf scene

scene :: Picture
scene = circle 0.1 & translated 3 0 (colored red triangle)

triangle :: Picture
triangle = polygon [(0,0),(1,-0.5),(1,0.5)]
```

Let us discuss this from the “expression” perspective …

Brief recap from last week

- Expressions: syntactic structures one could imagine after the “=” in an assignment “`var =` …” in C or Java.
- Values: results of evaluating expressions, obtained by combining values of subexpressions.
- Commands: syntactic structures that are characterized not so much by what (if anything at all) they evaluate to, but rather by what effect they have (change of storage cells, looping, etc.).
- In a pure setting without commands, any two expressions that have the same value can be replaced for each other, without changing the behaviour of the program.

Describing a picture via an expression

Observations:

- Compositionality on level of syntax, types, and values.
- Pictures are expressions/values here, can be named etc.
- Functions/operators used:

```
circle     : ℝ → Picture
polygon    : [ℝ×ℝ] → Picture
colored    : Color × Picture → Picture
translated : ℝ × ℝ × Picture → Picture
&          : Picture × Picture → Picture
```

- Properties like:

```
  translated a b (colored c d)
≡ colored c (translated a b d)
```

Describing an animation via a function

A slight variation of example from last week:

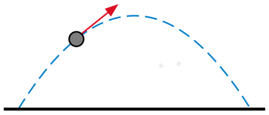
```
main :: IO ()
main = animationOf scene


scene :: Double -> Picture
scene t = translated t 0 (colored red triangle)
```

- Dependence on time expressed via parameter `t`.
- That parameter is never set by us ourselves for the animation.
- No `for`-loop or other explicit control.
- Instead, the `animationOf` construct takes care “somehow” (this involves evaluating `scene` for different `t`).

Another example

- Mathematically describing dynamic behaviour as a function of time should not be much of a surprise.
- A well-known physics example:

$$x(t) = v_{0x} \cdot t$$
$$y(t) = v_{0y} \cdot t - \frac{g}{2} \cdot t^2$$



- As a program:

```
scene :: Double -> Picture
scene t = cliff & translated x y (circle 0.1)
  where x = 3 * t
        y = 6 * t - 9.81 / 2 * t^2
        cliff = polyline [(-5,0),(0,0),(0,-2)]
```

Rich expressions

A desire for additional expressivity

- In the examples today, we have already expressed continuous distribution, throughout time, via functions.
- What if we also, or alternatively, want a discrete distribution, "throughout space"?
- So, instead of one triangle moving in time, we want several static triangles at different places.
- But we do not really want to replicate these "by hand".
- Maybe now is the time for a `for`-loop?
- No, we don't have that.
- But what do we have instead?

One kind of richer expressions: list comprehensions

Using a list comprehension:

```
main :: IO ()
main = drawingOf (pictures [ scene d | d <- [0..5] ])

scene :: Double -> Picture
scene d = translated d 0 (colored red triangle)
```

- With `pictures :: [ Picture ] -> Picture`.
- And a list comprehension `[ scene d | d <- [0..5] ]`.
- This is not exactly like a `for`-loop, for several reasons.
- Instead, it is like a mathematical set comprehension $\{ 2 \cdot n \mid n \in \mathbb{N} \}$.

More mundane examples of list comprehensions

```
> [1,3..10]
[1,3,5,7,9]

> [ x^2 | x <- [1..10], even x ]
[4,16,36,64,100]

> [ y | x <- [1..10], let y = x^2, mod y 4 == 0 ]
[4,16,36,64,100]

> [ x * y | x <- [1,2,3], y <- [1,2,3] ]
[1,2,3,2,4,6,3,6,9]
```

More mundane examples of list comprehensions

```
> [ (x,y) | x <- [1,2,3], y <- [4,5] ]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]

> [ (x,y) | y <- [4,5], x <- [1,2,3] ]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]

> [ (x,y) | x <- [1,2,3], y <- [1..x] ]
[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]

> [ x ++ y | (x,y) <- [("a","b"),("c","d")] ]
["ab","cd"]
```

So where are we, expressivity-wise?

Some takeaways from examples we have seen:

- **Non-constant behaviour expressed as functions, in the mathematical sense.** $f(x) = \cdots$
- **Such a description defines the behaviour “as a whole”, not in a “piecemeal” fashion.**
- **For example, there is no “first run this piece of animation, then that piece, and then something else”.**
- **Actually, there is not even a concept of “this piece of animation stops at some point”.**

Of course, we should be able to also express possibly non-continuous behaviours. But we are not resorting to sequential commands, with imperative keywords or semicolons etc.

List comprehensions are also not the answer, because they do not define functions, just (list) values. Instead, …

Case distinctions

- **Switching by conditional expressions:**

```
scene :: Double -> Picture
scene t = if t < 3
          then translated t t (circle 1)
          else blank
```

- **This is very much in line with case distinctions in mathematical functions:**

$$f(x) = \begin{cases} -x, & if\ x < 0 \\ x, & else \end{cases}$$

Comparison to the situation in imperative setting

- **In C/Java we have two forms of `if` on commands:**

```
if (...) { ... }
if (...) { ... } else { ... }
```

- **In an expression language, the form without `else` does not make sense, so in Haskell we always have:**

```
if ... then ... else ...
```

- **This corresponds to C/Java's conditional operator:**

```
... ? ... : ...
```

Some usage hints on case distinctions in Haskell

- **Pragmatically, an `if-then-else` expression "without an `else`" would be realized by having some "neutral value" in the `else`-branch. Remember:**

```
scene :: Double -> Picture
scene t = if t < 3
          then translated t t (circle 1)
          else blank
```

- **Similarly, in a list context:** `if condition then list else []`

- **Also, do not hesitate to use `if-then-else` as subexpressions freely:**

$$\texttt{f x y (if } \texttt{exp}_1 \texttt{ then } \texttt{exp}_2 \texttt{ else } \texttt{exp}_3\texttt{)}$$
$$\equiv\ \texttt{if } \texttt{exp}_1 \texttt{ then f x y } \texttt{exp}_2 \texttt{ else f x y } \texttt{exp}_3$$

Some remarks on syntax and types

“Oddities” of syntax at the type level

Instead of:

```
circle     : ℝ → Picture
polygon    : [ℝ×ℝ] → Picture
colored    : Color × Picture → Picture
translated : ℝ × ℝ × Picture → Picture
&          : Picture × Picture → Picture
```

type signatures actually look like this:

```
circle     :: Double -> Picture
polygon    :: [ (Double, Double) ] -> Picture
colored    :: Color -> Picture -> Picture
translated :: Double -> Double -> Picture -> Picture
(&)        :: Picture -> Picture -> Picture
```

“Oddities” of syntax at the expression/function level

- Instead of `f(x)` and `g(x,y,z)`, we write `f x` and `g x y z`.
- As an example for nested function application, instead of `g(x,f(y),z)`, we write `g x (f y) z`.
- The same syntax is used at function definition sites, so something like

  ```
  float f(int a, char b)
  { ... }
  ```

  in C or Java would correspond to

  ```
  f :: Int -> Char -> Float
  f a b = ...
  ```

  in Haskell.

Layout-sensitivity

In Haskell, this:

```
let y = a * b
    f x = (x + y) / y
in f c + f d
```

is equivalent to:

```
let { y = a * b; f x = (x + y) / y }
in f c + f d
```

But these are not accepted:

```
let y = a * b
      f x = (x + y) / y
in f c + f d
```

```
let y = a * b
   f x = (x + y) / y
in f c + f d
```

Other syntax remarks

- Haskell beginners tend to use unnecessarily many brackets. For example, no need to write `f (g (x))` or `(f x) + (g y)`, since `f (g x)` and `f x + g y` suffice.
- Further brackets can sometimes be saved by using the `$` operator, for example writing `f $ g x $ h y` instead of `f (g x (h y))`. I don't like it in beginners' code.
- We let Autotool give warnings about redundant brackets, as well as about overuse of `$`.
  Sometimes we enforce adherence to those warnings.

A specific observation based on exercise submissions

If you have repeated occurrences of a common subexpression, share them! For example, instead of something like this:

```
scene t =
  if 8 * sin t > 0
  then translated (8 * cos t) (8 * sin t) ...
  else ...
```

rather write this:

```
scene t =
  let x = 8 * cos t
      y = 8 * sin t
  in if y > 0 then translated x y ... else ...
```

Specifics about number types

- Haskell has various number types: `Int`, `Integer`, `Float`, `Double`, `Rational`, …
- Number literals can have a different concrete type depending on context, e.g., `3 :: Int`, `3 :: Float`, `3.5 :: Float`, `3.5 :: Double`
- For general expressions there are overloaded conversion functions, for example `fromIntegral` with, among others, any of the types `Int -> Integer`, `Integer -> Int`, `Int -> Rational`, …, and `truncate`, `round`, `ceiling`, `floor`, each with any of the types `Float -> Int`, `Double -> Integer`, …

… and arithmetic operators

- Operators are also overloaded, and often no conversion is necessary, for example in `3 + 4.5` or also in:

```
f x = 2 * x + 3.5
g y = f 4 / y
```

- In other cases, conversion is necessary, for example in this:

```
f :: Int -> Float
f x = 2 * fromIntegral x + 3.5
```

or:

```
f x = 2 * x + 3.5
g y = f (fromIntegral (length "abcd")) / y
```

… and arithmetic operators

- Some operators are available only at certain types, e.g., no division symbol “`/`” on integer types.
- Instead, the function `div :: Int -> Int -> Int` (also on `Integer`).
- Binary functions (not just arithmetic ones) can be used like operators, for example writing ``17 `div` 3`` instead of `div 17 3`.
- Useful mathematical constants and functions exist, e.g., `pi`, `sin`, `sqrt`, `min`, `max`, …

Some observations based on past years' exercises

- In case of doubt concerning number conversions, it usually does not hurt to add some `fromIntegral`-calls, which in the worst case will be no-ops (since, among others, `fromIntegral :: Int -> Int`).
- It is always a good idea to write down type signatures for (at least) top-level functions. Among other benefits, it saves you from having to deal with (errors involving) types like:

```
    fun :: (Floating a, Ord a) => a -> a
```

Types beside number types

Other pre-existing types:

- Type `Bool`, with values `True` and `False` and operators `&&`, `||`, and `not`.
- Type `Char`, with values `'a'`, `'b'`, …, `'\n'` etc., and functions `succ`, `pred`, as well as comparison operators.
- List types: `[Int]`, `[Bool]`, `[[Int]]`, …, with various pre-defined functions and operators.
- Character sequences: `type String = [Char]`, with special notation `"abc"` instead of `['a','b','c']`.
- Tuple types: `(Int,Int)`, `(Int,String,Bool)`, `((Int,Int),Bool,[Int])`, also `[(Bool,Int)]` etc.

Programming by case distinction

(more ways of doing it)

Expressing conditional behaviour

Remember:

- Switching by conditional expressions:

```
scene :: Double -> Picture
scene t = if t < 3
          then translated t t (circle 1)
          else blank
```

- This is very much in line with case distinctions in mathematical functions:

$$f(x) = \begin{cases} -x, & if\ x < 0 \\ x, & else \end{cases}$$

Expressing conditional behaviour

- Likely not yet seen, function definition using guards:

```
scene t
  | t <= pi               = ...
  | pi < t && t <= 2 * pi = ...
  | 2 * pi < t            = ...
```

- This is again similar to mathematical notation:

$$f(x) = \begin{cases} 0, & if\ x \leq 0 \\ x, & if\ 0 < x \leq 1 \\ 1, & if\ x > 1 \end{cases}$$

Function definition using guards

- Let us discuss some details based on this example:

```
factorial :: Integer -> Integer
factorial n
  | n == 0 = 1
  | n > 0  = n * factorial (n - 1)
```

- First of all, what about the order of clauses?
- Well, in this example, the following variant is equivalent:

```
factorial :: Integer -> Integer
factorial n
  | n > 0  = n * factorial (n - 1)
  | n == 0 = 1
```

Function definition using guards

- What if the guard conditions overlap?
- Then this is okay:

```
factorial :: Integer -> Integer
factorial n
  | n == 0 = 1
  | n >= 0 = n * factorial (n - 1)
```

but this is problematic:

```
factorial :: Integer -> Integer
factorial n
  | n >= 0 = n * factorial (n - 1)
  | n == 0 = 1
```

- Always the first matching clause is used!

Function definition using guards

- Even with the “correct” order:

```
factorial :: Integer -> Integer
factorial n
  | n == 0 = 1
  | n >= 0 = n * factorial (n - 1)
```

we can have problems with some inputs.

- If no clause matches, we get a runtime error!

Function definition using guards

- In fact, if called with appropriate settings, the compiler warns us of a potential runtime error ahead of time.
- We can avoid both the warning and the actual non-exhaustiveness error at runtime by having a “catch-all” clause:

```
factorial :: Integer -> Integer
factorial n
  | n == 0    = 1
  | otherwise = n * factorial (n - 1)
```

Function definition using guards

- In this specific case, negative inputs would still be a problem.
- Which we could remedy as follows:

```
factorial :: Integer -> Integer
factorial n
  | n <= 0    = 1
  | otherwise = n * factorial (n - 1)
```

- Some lessons: order matters (and can be exploited), exhaustiveness matters. Also, some further aspects…

Function definition using guards

- The compiler’s checks ahead of time are nice, but necessarily not perfect.
- For example, it cannot in general detect infinite recursion ahead of time. (The Halting Problem!)
- Even the “simpler” static exhaustiveness checks are not as powerful as one might sometimes hope.
- For example, one might hope that something like this:

```
f x y
  | x == y = ...
  | x /= y = ...
```

is statically determined safe. But no (and for good reason). So it is usually better to use an explicit `otherwise` clause.

Function definition using guards

- Also, the more desirable “fix” to the issue of possible negative inputs for

```
factorial :: Integer -> Integer
factorial n
  | n == 0    = 1
  | otherwise = n * factorial (n - 1)
```

(instead of switching to `n <= 0` in the first clause) would be to statically prevent negative inputs from occurring at all, via the type system.

- But that is a topic for another lecture.

Function definition using guards

- **For now, let us apply our insights to this situation considered earlier:**

```
scene t
  | t <= pi                 = ...
  | pi < t && t <= 2 * pi = ...
  | 2 * pi < t              = ...
```

- **Here is how this should probably look instead:**

```
scene t
  | t <= pi      = ...
  | t <= 2 * pi = ...
  | otherwise    = ...
```

Function definition using guards

Some further syntactic variations:

```
factorial :: Integer -> Integer
factorial n | n == 0    = 1
factorial n | otherwise = n * factorial (n - 1)
```

```
factorial :: Integer -> Integer
factorial n | n == 0 = 1
factorial n          = n * factorial (n - 1)
```

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Function definition using guards

Another example:

```
ackermann :: Integer -> Integer -> Integer
ackermann 0 n | n >= 0 = n + 1
ackermann m 0 | m > 0  = ackermann (m - 1) 1
ackermann m n | m > 0 && n > 0
  = ackermann (m - 1) (ackermann m (n - 1))
```

This one gives some interesting non-exhaustiveness warnings.

Function definitions generally

General rules for function definitions:

- One or more equations, with or without guards.
- One or more arguments; so far, only variable names (can be anonymous) or constants.
- Uniqueness of variable names within one equation.
- Never expressions, in argument position at definition sites, that would require computation or “solving”.

Function definitions generally

A few more examples:

```
not :: Bool -> Bool
not True = False
not _    = True

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

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

Specific observations from exercises

Some observations based on exercise submissions

If the Autotool/`hlint` feedback mentions "eta reduction", here is what it means:

- **Instead of something like:**

```
ball :: Double -> Picture
ball t = solidCircle t
```

  one might just as well write:

```
ball :: Double -> Picture
ball = solidCircle
```

- **Also consider:**

```
opening :: Double -> Picture
opening = rectangle 10
```

Some observations based on exercise submissions

Almost every time one sees a use of access-by-index in Haskell code, it was not the best choice of expression.

A typical case is if something corresponding to this:

```
whatever [ computeFrom argument
         | argument <- list ]
```

was instead written like this:

```
whatever [ computeFrom (list !! index)
         | index <- [0..(length list - 1)] ]
```

Generally working with lists

A few words about lists up front

- We will consider a lot of examples in the lecture and exercises that deal with lists.
- But that is mostly for didactical reasons. In the “real world”, there are often more appropriate data structures (and we will eventually see how to define them ourselves).
- In part due to historical precedent (Lisp), Haskell has a very rich library of list processing functions.
- It also has specific syntactical support for lists (e.g., list comprehensions).
- As already mentioned, Haskell lists are homogeneous.

Examples of existing (first-order) functions on lists

```
take 3 [1..10]           ==       [1,2,3]
drop 3 [1..10]           ==       [4,5,6,7,8,9,10]
null []                  ==       True
null "abcde"             ==       False
length "abcde"           ==       5
head "abcde"             ==       'a'
last "abcde"             ==       'e'
tail "abcde"             ==       "bcde"
init "abcde"             ==       "abcd"
splitAt 3 "abcde"        ==       ("abc","de")
"abcde" !! 3             ==       'd'
reverse "abcde"          ==       "edcba"
"abc" ++ "def"           ==       "abcdef"
zip "abc" "def"          ==       [('a','d'),('b','e'),('c','f')]
concat [[1,2],[],[3]]    ==       [1,2,3]
```

Different ways of working with lists

We now have certain choices, such as whether to work with recursion or by just combining existing functions (and possibly list comprehensions).

For example:

```
isPalindrome :: String -> Bool
isPalindrome s | length s < 2 = True
isPalindrome s = head s == last s &&
                 isPalindrome (init (tail s))
```

vs.:

```
isPalindrome :: String -> Bool
isPalindrome s = reverse s == s
```

Infinite lists

- In Haskell there are even expressions and values for infinite lists, for example:

```
[1,3..]                 ≡ [1,3,5,7,9,...]
[ n^2 | n <- [1..] ] ≡ [1,4,9,16,...]
```

- And while we of course cannot print complete such lists, we can still work normally with them, as long as the ultimate output is finite:

```
take 3 [ n^2 | n <- [1..] ] == [1,4,9]
zip [0..] "ab" == [(0,'a'),(1,'b')]
```

Infinite lists

But there is no mathematical magic at work, so for example this:

```
[ m | m <- [ n^2 | n <- [1..] ], m < 100 ]
```

will “hang” after producing a finite prefix.

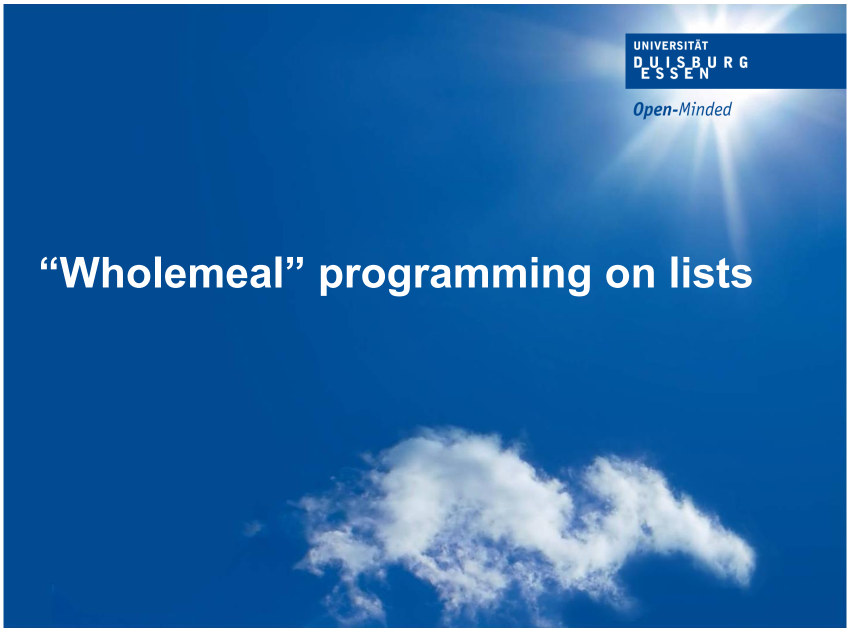
Why is that, actually?

Discussion: involves referential transparency!

An interesting function on finite lists

Essentially Quicksort:

```
sort :: [Integer] -> [Integer]
sort []   = []
sort list =
 let
  pivot = head list
  smaller = [ x | x <- tail list, x < pivot ]
  greater = [ x | x <- tail list, x >= pivot ]
 in sort smaller ++ [ pivot ] ++ sort greater
```



“Wholemeal” programming on lists

Wholemeal programming

- “Functional languages excel at wholemeal programming, a term coined by Geraint Jones. Wholemeal programming means to think big: work with an entire list, rather than a sequence of elements; …”
  Ralf Hinze

- “Wholemeal programming is good for you: it helps to prevent a disease called indexitis, and encourages lawful program construction.”
  Richard Bird

Wholemeal programming on lists

We earlier had this example:

```
main :: IO ()
main = drawingOf (pictures [ scene d | d <- [0..5] ])


scene :: Double -> Picture
scene d = translated d 0 (colored red triangle)
```

- This is already a wholemeal approach, since we express the application of `scene` to the elements of `[0..5]` “in one go”.
- Specifically, we do not conceptually consider “one after another”. Instead, the resulting values are completely independent, no individual instance influences any other.
- Just like in the mathematical notation $\{ f(n) \mid n \in \mathbb{N} \}$.

Wholemeal programming on lists

We earlier had this example:

```
main :: IO ()
main = drawingOf (pictures [ scene d | d <- [0..5] ])


scene :: Double -> Picture
scene d = translated d 0 (colored red triangle)
```

- Of course, the individual evaluations will, on a sequential machine, happen in some order. And the resulting list is really a sequence, not a set. But the individual values will be independent of all that.
- Indeed, one can show that for any `f` and `n`, in Haskell:

```
       [ f a | a <- [0..n] ]
    ≡  reverse [ f a | a <- reverse [0..n] ]
```

Contrast to for-loops in Java, C, etc.

- In contrast, it is not remotely true that in an imperative language we can always replace a piece of code written like this:

```
for (a = 0; a <= n; a++)
  result[a] = f(a);
```

by this:

```
for (a = n; a >= 0; a--)
  result[a] = f(a);
```

- And even for the cases where commands as above <u>are</u> equivalent, a formulation given that way is less useful than the Haskell equation we saw, or indeed its more general version:

```
       reverse [ f a | a <- list ]
    ≡  [ f a | a <- reverse list ]
```

Wholemeal programming on lists

- Another example: Assume we want to multiply each element of an array or list by its position in that data structure, and sum up over all the resulting values.
- It seems fair to say that this is a typical solution in C:

```
int array[n];
int result = 0;

for (int i = 0; i < n; i++)
  result = result + i * array[i];
```

- And that is about okay, but it does suffer from indexitis.

Wholemeal programming on lists

- The same example, in a wholemeal fashion, in Haskell:

```
sum [ i * v | (i, v) <- zip [0..] list ]
```

- Nice, short, declarative.
- Of course, one could consider this cheating, because it is using a conveniently predefined function `sum`.
- But actually, that is besides the point. Even without that convenience function, it would not have taken more than a dozen keystrokes to express the summation.
- And using a convenient array sum function would not exactly have made the C version any nicer than it is.

Wholemeal programming on lists

- So let us discuss the actual issues, expressivity and susceptibility to change and refactoring.
- Say, what if we decided that the counting of positions should start at 1 instead of 0?
- In the C version, that could mean we would switch from this:

```
for (int i = 0; i < n; i++)
  result = result + i * array[i];
```

  to this:

```
for (int i = 1; i <= n; i++)
  result = result + i * array[i-1];
```

- Indexitis!

Wholemeal programming on lists

- In the Haskell version, we simply switch from this:

```
sum [ i * v | (i, v) <- zip [0..] list ]
```

  to this:

```
sum [ i * v | (i, v) <- zip [1..] list ]
```

- To be fair again, in C we could have made a different edit:

```
for (int i = 0; i < n; i++)
  result = result + (i+1) * array[i];
```

- But actually, that is just indexitis in a different form.

Wholemeal programming on lists

- The fundamental issue in the C version is a lack of conceptual separation of values to enumerate/process on the one hand, and loop control on the other hand.
- Whereas the Haskell version has that separation in the `zip [k..] ...` expression.

- Basically, the Haskell version needs no explicit loop control, it does not access data structure elements by index (remember what I said about avoiding use of the `!!` operator whenever possible), and it does not need to increment a loop counter or talk about the "loop end" condition (because: infinite lists).

Wholemeal programming on lists

- Okay, but are we fooling ourselves, efficiency-wise?
- Certainly, code like

```
for (int i = 0; i < n; i++)
  result = result + i * array[i];
```

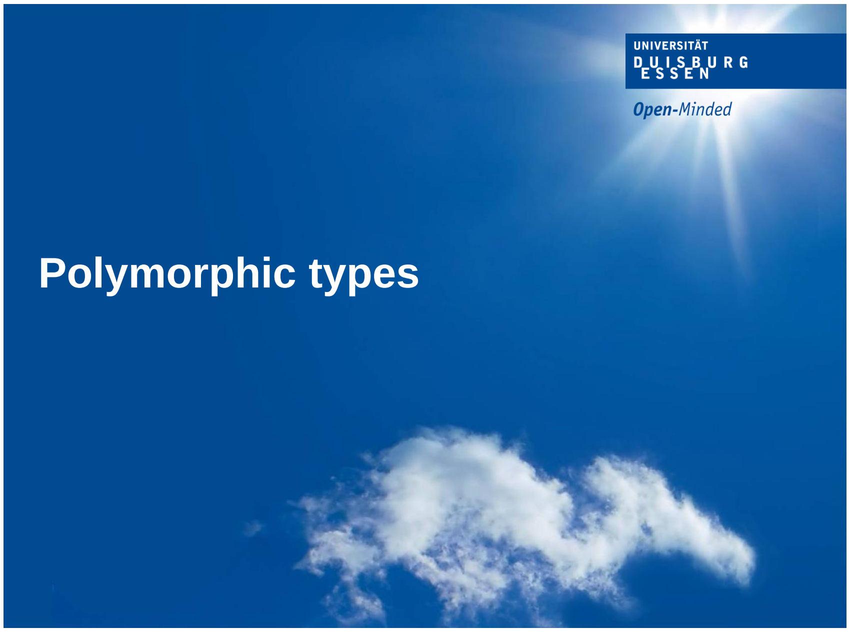
is more efficient than

```
sum [ i * v | (i, v) <- zip [0..] list ]
```

because it does not need to use extra memory, and does not need several data structure traversals?

Wholemeal programming on lists

- Well, no. Actually, a compiler can translate the declarative code into a tight C-like loop, not using an intermediate data structure, just fine.
- A compiler can even spot parallelization opportunities, thanks to the "independent values" aspect we already discussed when comparing list comprehensions against `for`-loops.
- That all has to do also with the "lawful program construction" aspect from the Richard Bird quote.
- We could also talk more about refactoring…
- But is what we saw for the somewhat artificial example now representative of real situations? Claim: Yes!



Polymorphic types

Polymorphic functions on lists

- Remember that each Haskell list is homogeneous, i.e., cannot contain elements of different types.

```
"abc"   :: [Char]
[1,2,3] :: [Integer]
['a',2] -- ill-typed
```

- At the same time, functions and operators on lists can be used quite flexibly:

```
reverse "abc"   == "cba"
reverse [1,2,3] == [3,2,1]
"abc" ++ "def"  == "abcdef"
[1,2] ++ [3,4]  == [1,2,3,4]
```

- We have already depended on this flexibility a lot!

Polymorphic functions on lists

- So there should be a way to reconcile the rigidity of types with flexible use of functions.
- We want to be able to write

  `"abc" ++ "def"` and `[1,2] ++ [3,4]`,

  as well as

  `elem 2 [1,2]` and `elem 'c' "ab"`,

  but at the same time prevent calls like

  `"ab" ++ [3,4]` and `elem 'a' [1,2,3]`.

Polymorphic functions on lists

- So what are the types of functions like those seen?
- We do not have, and clearly do not want, different functions like `reverseChar :: [Char] -> [Char]` and `reverseInteger :: [Integer] -> [Integer]`.
- Instead, we use type variables, as in:

```
reverse :: [a] -> [a]
```

- That is not, at all, like being untyped. For example, the type `(++) :: [a] -> [a] -> [a]` does not mean that “anything goes”.
  (Still not possible to write this: `"ab" ++ [3,4]`.)

Polymorphic functions on lists

- We have already seen a lot of functions that fit this pattern:

```
head   :: [a] -> a
tail   :: [a] -> [a]
last   :: [a] -> a
init   :: [a] -> [a]
length :: [a] -> Int
null   :: [a] -> Bool
concat :: [[a]] -> [a]
```

- In concrete applications, the type variable gets instantiated appropriately: `head "abc" :: Char`.

Polymorphic functions on lists

- Of course, a polymorphic function does not need to be polymorphic in all its arguments.
- For example:

```
(!!) :: [a] -> Int -> a
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
splitAt :: Int -> [a] -> ([a],[a])
```

- And what about `zip`?

Polymorphic functions on lists

- The function `zip` also takes homogeneous lists as arguments.
- But unlike the case of `(++)`, where we want to allow `"ab" ++ "cd"` and `[1,2] ++ [3,4]`, but to disallow `"ab" ++ [3,4]`, for `zip` we want to allow all of the following:

```
zip "ab" "cd"
zip [1,2] [3,4]
zip "ab" [3,4]
```

- So the type cannot be like that for `(++)`:

```
[a] -> [a] -> ...
```

Polymorphic functions on lists

- Instead:

```
zip :: [a] -> [b] -> [(a,b)]
```

- Different type variables can be, but do not have to be, instantiated by different types.

- Hence, all of these make sense:

```
zip "ab" "cd"     -- a = Char, b = Char
zip [1,2] [3,4]   -- a = Int, b = Int
zip "ab" [3,4]    -- a = Char, b = Int
```

- Whereas a mixed call for `(++)` does not:

```
"ab" ++ [3,4]     -- a = Char or Int?
```

Polymorphic functions in other languages

- Have you seen something like those types in another language before?

- Example in Java with Generics:

```
<T> List<T> reverse(List<T> list)
{ ... }
```

corresponding to:

```
reverse :: [a] -> [a]
reverse list = ...
```

Inference of polymorphic types

- One aspect (among several) that distinguishes polymorphism in Haskell and its FP predecessors from those other languages is type inference.
- We need not declare polymorphism, since the compiler will always infer the most general type automatically.
- For example, for `f (x,y) = x` the compiler infers `f :: (a,b) -> a`.
- And for `g (x,y) = if pi > 3 then x else y`, `g :: (a,a) -> a`.

Consequences of polymorphic types

- Polymorphism has really interesting semantic consequences.
- For example, earlier in the lecture, I mentioned that always:

```
      reverse [ f a | a <- list ]
    ≡ [ f a | a <- reverse list ]
```

- What if I told you that this holds, for arbitrary `f` and `list`, not only for `reverse`, but for any function with type `[a] -> [a]`, no matter how it is defined?
- Can you give some such functions (and check the above claim on an intuitive level)?

Consequences of polymorphic types

- Recall that the `reverse`-claim earlier in the lecture occurred in the context of comparing, in the imperative world, this:

```
for (a = 0; a <= n; a++)
  result[a] = f(a);
```

  vs. this:

```
for (a = n; a >= 0; a--)
  result[a] = f(a);
```

- Not only are these two loops not necessarily equivalent, but even when imposing conditions under which they are, we do not get an as general and readily applicable law as just seen in the declarative world.

Higher-order functions

Higher-order functions

- So far, we have mainly dealt with first-order functions, that is, functions that take “normal data” as input arguments and ultimately return some value.
- But we have also already seen functions to which we passed other functions as arguments. For example, `quickCheck` and `animationOf`.
- Indented, let us take a look at the type of the latter:
  `animationOf :: (Double -> Picture) -> IO ()`
- Note: Every function is a (mathematical) value, but not every value is a function.

The types of higher-order functions

- The type

  `animationOf :: (Double -> Picture) -> IO ()`

  means something completely different than the type

  `animationOf :: Double -> Picture -> IO ()`

- Indeed, parentheses in such places are very significant.
- Let us discuss this based on a simpler example type.

The types of higher-order functions

- **What are some functions of the following type?**

```
f :: Int -> Int -> Int
```

- **And what about the following type?**

```
f :: (Int -> Int) -> Int
```

- **What kinds of inputs does either of these take?**
- **And what can they do with their inputs?**

The types of higher-order functions

```
f :: Int -> Int -> Int
f x y = x+y  ┐
f x y = x-y  │ or
f _ y = y    │
f _ _ = 12   ┘
```

```
f :: (Int -> Int) -> Int
f h = h 7          ┐
f _ = 17           │ or
f h = h (h 13)     │
f h = h 4 + h 7    ┘
      ↑
```

pure extensional uses,
not looking at the
syntax of h

Functions to pass to higher-order functions

- **Where do we get functions from that we can pass as arguments to higher-order functions?**
- **Well, in Haskell functions are almost everywhere, right? So we should not have any shortage of supply.**
- **Of course, there are many predefined functions already.**
- **We could also use functions we have explicitly defined in our program (such as passing your own `scene` function to `animationOf`).**
- **Or partial applications of any of those. For example, `(+) :: Int -> Int -> Int`, and as a consequence, `(+) 5 :: Int -> Int`.**

Functions to pass to higher-order functions

$$f\ h = h\ 7$$

$$f\ \underbrace{((+)\ 5)}_{h} = h\ 7 = \underbrace{(+)\ 5}_{h}\ 7 = 12$$

$$f\ \underbrace{(5\ +)}_{h} = h\ 7 = \underbrace{5\ +}_{h}\ 7 = 12$$

again, h used purely extensionally!

Some syntactic specialties

- Indeed, the type `Int -> Int -> Int` could be read as `Int -> (Int -> Int)`.
- But <u>those</u> parentheses can be omitted.
- Two viewpoints here: a function that takes two `Int` values and returns one `Int` value, or a function that takes one `Int` value and returns a function that takes one `Int` value and returns one `Int` value.
- Both viewpoints are valid! No difference in usage (thanks to Haskell’s function application syntax).
- Another syntactic specialty: so-called “sections”. For example, “`(+) 5`” can be written as “`(5 +)`”.

Some syntactic specialties

prefix: (+) , mod   infix: 1+2 , \`mod\`

(+) 5 ≡ (5 +)   also (+ 5) , semantically the same

(5 -) and (- 5) do different things

also useful for predicates:

(<5) :: Int → Bool

(5 -) is a function, waiting for x, computing 5 - x

Lambda-abstractions

- We can also syntactically create new functions “on the fly”, instead of predefined or own, explicitly defined and named, functions already in the program.
- Such anonymous functions use the so-called lambda-abstraction syntax (which we have already seen in the context of QuickCheck tests): `\x -> x + x`
- So, some options of functions we could pass to a function `f :: (Int -> Int) -> Int` are:
  `id`, `succ`, `(gregorianMonthLength 2019)`, `(- 5)`, `(\x -> x + x)`, `(\n -> length [1..n])`

Lambda-abstractions

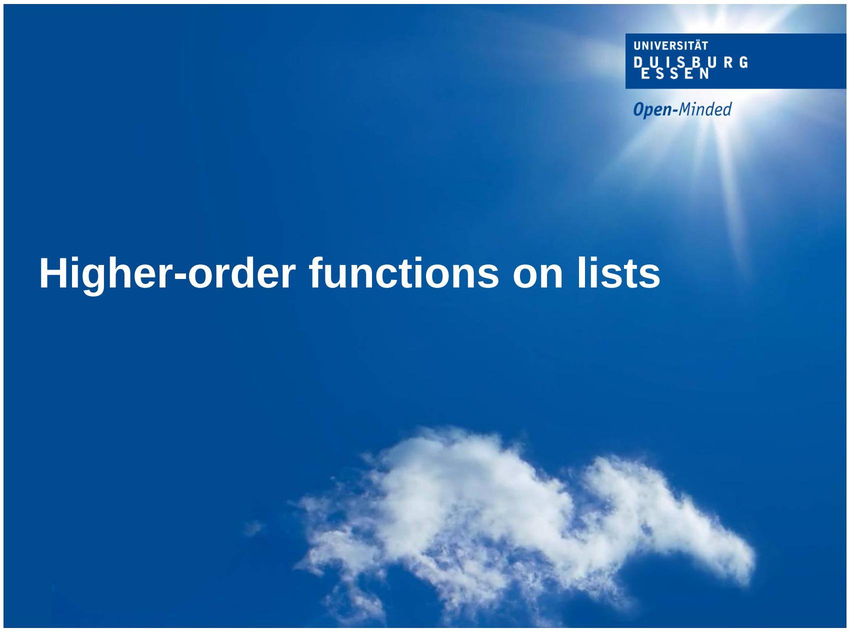
- The lambda-abstraction syntax also allows us to get a clearer view on Haskell’s function definition syntax (and its choice to be different from standard mathematical function definition syntax).
- Namely, the following four definitions are equivalent (each of type `add :: Int -> Int -> Int`):

```
add x y = x + y
add x = \y -> x + y
add = \x -> \y -> x + y
add = \x y -> x + y
```

- With standard mathematical notation, `add(x,y) = ,` such variations would not have been so fluent.

Usefulness of higher-order functions

- But is any of that really useful to us?
- The examples so far look somewhat esoteric and artificial, except maybe for the `animationOf` and `quickCheck` “drivers”, which we do not know how to write ourselves yet though, anyway (due in part to the involvement of `IO`).
- Well, there are many immediately useful higher-order functions on lists as well…



Higher-order functions on lists

Higher-order functions on lists

- For example, the function

```
foldl1 :: (a -> a -> a) -> [a] -> a
```

puts a (left-associative) function/operator between all elements of a non-empty list.

- So to compute the sum of such a list:

```
foldl1 (+) [1,2,3,4]
```

which will expand to:

```
1 + 2 + 3 + 4
```

Higher-order functions on lists

- Another useful function:

```
map :: (a -> b) -> [a] -> [b]
```

which applies a function to all elements of a list.

- For example:

```
map even [1..10]

map (dilated 5) [ pic1, pic2, pic3 ]
```

Higher-order functions on lists

- And another one:

```
filter :: (a -> Bool) -> [a] -> [a]
```

  which selects list elements that satisfy a certain predicate.

- For example,

```
filter isPalindrome completeDictionary

filter (> 0.5) bonusPercentageList
```

Relationship to list comprehensions

- While the following are not the actual definitions of `map` and `filter`, we can think of them as such:

```
map :: (a -> b) -> [a] -> [b]
map f list = [ f a | a <- list ]

filter :: (a -> Bool) -> [a] -> [a]
filter p list = [ a | a <- list, p a ]
```

- Conversely, <u>every</u> list comprehension expression, no matter how complicated with several generators, guards, etc., can be implemented via `map`, `filter`, and `concat`.

Relationship to list comprehensions

- Is programming with `map` and `filter` (and `foldl1` and the like) still “wholemeal programming”, which is what we have mostly used list comprehensions for so far?
- Yes, absolutely. In a sense even more so, since higher-order functions provide a further step in the direction of more abstraction.

- For example, if we want to square some numbers from a given list, subject to the condition that we are specifically interested in numbers divisible by four, but still have to work out whether we want to check this divisibility before or after squaring, then …

Relationship to list comprehensions

… with list comprehensions we would consider, and maybe experiment with,

```
[ x^2 | x <- list, x `mod` 4 == 0 ]
```

vs.

```
[ y | x <- list, let y = x^2, y `mod` 4 == 0 ]
```

While with `map` and `filter` we would simply decide between

```
map (^2) . filter (\x -> x `mod` 4 == 0)
```

and

```
filter (\x -> x `mod` 4 == 0) . map (^2)
```

Expressing laws

- Also, a law like (mentioned earlier):

```
  reverse [ f a | a <- list ]
≡ [ f a | a <- reverse list ]
```

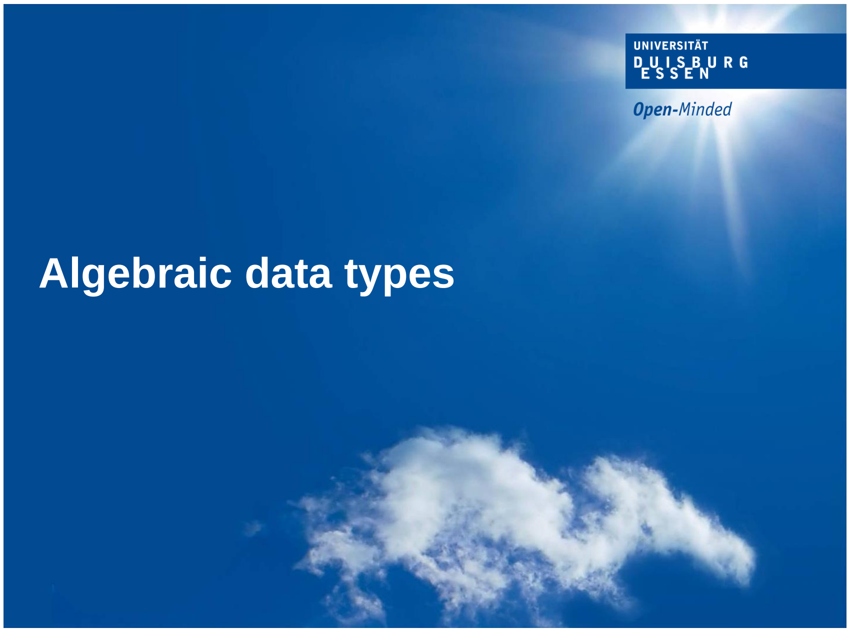
  can nicely be expressed as:

```
reverse . map f ≡ map f . reverse
```

- Then we can also ask under which conditions this holds:

```
filter p . map f ≡ map f . filter q
```

- Generally, higher-order functions are a boon for “lawful program construction” (see the Richard Bird quote).



Algebraic data types

Types in Haskell

- We have so far seen various types on which functions can operate, such as number types (`Integer`, `Float`, …), other base types like `Bool` and `Char`, as well as list and tuple constructions to make compound types, arbitrarily nested (`[…]`, `(…,…)`).
- We have also seen that libraries can apparently define their own, domain specific types, such as `Picture`.
- To do the same ourselves: algebraic data types.
- These are a more general and more stringent version of what is usually known as enumeration or union types. They are also the inspiration for features like Swift's (recursive) `enum` types.

Simple enumeration types

- Let us start simple. Assume we want to be able to talk about days of the week, and compute things like "this is a workday, yes/no".
- We could fix some encoding of Monday, Tuesday etc. as numbers (e.g., Monday = 1, Tuesday = 2, …) and define functions like:

```
workday :: Integer -> Bool
workday d = d < 6
```

- In a sense, we were lucky here that the intended property corresponds to number ranges 1–5 and 6–7.

Simple enumeration types

- So let us try to instead express on which days of the week there would have been an exercise session in the ProPa course.
- The answer this time is not a simple arithmetic comparison like `d < 6`, but we can for example implement:

```
exerciseDay :: Integer -> Bool
exerciseDay 3 = False
exerciseDay 6 = False
exerciseDay 7 = False
exerciseDay _ = True
```

- In either case, what if we call `workday` or `exerciseDay` with an input like `12`?

Simple enumeration types

- Alternative approach, explicit new values:

```
data Day
  = Monday | Tuesday | Wednesday | Thursday
  | Friday | Saturday | Sunday
```

- Now:

```
exerciseDay :: Day -> Bool
exerciseDay Wednesday = False
exerciseDay Saturday  = False
exerciseDay Sunday    = False
exerciseDay _         = True
```

… and it is impossible to pass illegal inputs (like `12`th day).

- Terminology: type constructors and data constructors.

Simple enumeration types

- In addition to excluding absurd inputs, we get more useful exhaustiveness (and also redundancy) checking.
- For example, remember the game level example:

```
level :: (Integer, Integer) -> Integer

aTile :: Integer -> Picture
aTile 1 = block
aTile 2 = water
aTile 3 = pearl
aTile 4 = air
aTile _ = blank
```

- Imagine that we introduce a new kind of tile, produce its new “number code” inside the `level`-function, but forget to also handle it in the `aTile`-function. No compiler warning!

Simple enumeration types

If we had instead introduced a new type:

```
data Tile = Blank | Block | Pearl | Water | Air
```

and used `level :: (Integer, Integer) -> Tile`

and:

```
aTile :: Tile -> Picture
aTile Blank = blank
aTile Block = block
aTile Pearl = pearl
aTile Water = water
aTile Air   = air
```

then adding another value to `data Tile` could not go unnoticed in `aTile`.

The compiler would actually warn us if we forgot to handle the new value there!

General algebraic data types

- Going beyond simple enumeration types, algebraic data types can encapsulate additional values in the alternatives.
- That is, the data constructors can take arguments.
- For example:

```
data Date = Day Integer Integer Integer
data Time = Hour Integer
data Connection = Train Date Time Time
                | Flight String Date Time Time
```

- A possible value of type `Connection`:

```
Train (Day 20 04 2011) (Hour 11) (Hour 14)
```

General algebraic data types

- Computation on such types is via pattern-matching:

```
travelTime :: Connection -> Integer

travelTime (Train _ (Hour d) (Hour a))
  = a - d + 1
travelTime (Flight _ _ (Hour d) (Hour a))
  = a - d + 2
```

- At the same time, the data constructors are also normal functions, for example:

```
Day :: Integer -> Integer -> Integer -> Date

Train :: Date -> Time -> Time -> Connection
```

Recursive types

- Algebraic data types can be recursive. For example:

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

- Values of this type:

```
Zero, Succ Zero, Succ (Succ Zero), …
```

- Computation by recursive function definitions:

```
add :: Nat -> Nat -> Nat
add Zero     m = m
add (Succ n) m = Succ (add n m)
```

Recursive types

- With several recursive occurrences, tree structures:

```
data Tree = Leaf | Node Tree Integer Tree
```

- Values: `Leaf`, `Node Leaf 2 Leaf`, …

- Computation:

```
height :: Tree -> Integer
height Leaf
  = 0
height (Node left _ right)
  = 1 + max (height left) (height right)
```

Polymorphism in algebraic data types

Just like functions, algebraic data types can be polymorphic:

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)

height :: Tree a -> Integer
height Leaf
  = 0
height (Node left _ right)
  = 1 + max (height left) (height right)
```

Polymorphism in algebraic data types

- Another example, from the standard library:

```
data Maybe a = Nothing | Just a
```

- Popular for functions that would otherwise be partial.
- Such as also in a re-design of the game level example:

```
data Tile = Block | Pearl | Water | Air

level :: (Integer, Integer) -> Maybe Tile

aTile :: Tile -> Picture
aTile Block = block
aTile Pearl = pearl
aTile Water = water
aTile Air   = air
```

Persistency of data structures

- **Note that, just as any other data in Haskell, values of algebraic data types are immutable.**
- **For example, we do not <u>change</u> any tree in a function like this:**

```
insert :: Integer -> Tree Integer
                  -> Tree Integer
insert n Leaf = Node Leaf n Leaf
insert n tree@(Node left m right)
  | n < m = Node (insert n left) m right
  | n > m = Node left m (insert n right)
  | otherwise = tree
```

- **Discuss what this means …**

Lists as algebraic data type

Another example data structure

- **If Haskell did not yet have a list type, we could implement one ourselves:**

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

- **Example value:** `Cons 1 (Cons 2 Nil) :: List Int`
- **Computation:**

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

Lists as just another algebraic data type

- In fact, modulo special syntax, that is exactly what Haskell lists are:

```
data [a] = [] | (:) a [a]
```

- So, for example, `[1,2]` is simply `1:(2:[])`, which thanks to right-associativity of “`:`” can also be written as `1:2:[]`.

- Functions on lists can then, of course, also be defined using pattern-matching.

Pattern-matching on lists

Some example functions:

```
length :: [a] -> Int
length []       = 0
length (_:rest) = 1 + length rest

append :: [a] -> [a] -> [a]
append []     ys = ys
append (x:xs) ys = x : append xs ys

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

zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _      _      = []
```

Pattern-matching on lists

- Note how clever arrangement of cases/equations can make function definitions more succinct.
- For example, we might on first attempt have defined `zip` as follows:

```
zip :: [a] -> [b] -> [(a,b)]
zip []     _      = []
zip (x:xs) []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

- But the version from the previous slide is equivalent.
- Both versions also work with infinite lists, btw.

Higher-order examples

Also, as another example of a function we have used:

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

And indeed related:

```
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap _ Leaf = Leaf
treeMap f (Node left x right)
  = Node (treeMap f left)
         (f x)
         (treeMap f right)
```

Higher-order examples

- Also remember the function

  ```
  foldl1 :: (a -> a -> a) -> [a] -> a
  ```

  which puts a (left-associative) function/operator between all elements of a non-empty list.

- It is a member of a whole family of related functions, the most prominent of which is `foldr`, defined thus:

  ```
  foldr :: (a -> b -> b) -> b -> [a] -> b
  foldr _ c []     = c
  foldr f c (x:xs) = f x (foldr f c xs)
  ```

Notes on pattern-matching

Evaluation by pattern-matching

- **Ultimately, pattern-matching is what drives (lazy) evaluation in Haskell.**
- **For example, let us consider how the expression**

```
head (tail (f [3, 3 + 1]))
```

is evaluated, given the following function definitions (and the known `head` and `tail` functions):

```
f :: [Int] -> [Int]          g :: Int -> Int
f []     = []                g 3 = g 4
f (x:xs) = g x : f xs        g n = n + 1
```

Explicit case-expressions

- **Pattern-matching is not restricted to the left-hand sides of function definitions, it can also occur inside expressions, using the `case`-keyword.**
- **For example, instead of something like this:**

```
if maybeThing == Nothing
then … something …
else … something else, using fromJust maybeThing …
```

we can (and would usually prefer to) write this:

```
case maybeThing of
  Nothing    -> … something …
  Just thing -> … something else, directly using thing …
```

Binding of variables

- **Pattern-matching always binds variable names that occur in patterns, possibly shadowing existing things of same name.**
- **That sometimes leads to confusion for beginners, such as why it does not work to write a function like the following one (given the existence of `red :: Color` etc., imported from `CodeWorld`):**

```
primaryColor :: Color -> Bool
primaryColor red   = True
primaryColor green = True
primaryColor blue  = True
primaryColor _     = False
```

Input / Output

"In short, Haskell is the world's finest imperative programming language."

Simon Peyton Jones

Input / Output in Haskell, general approach

- Even in declarative languages, there should be some (disciplined) way to embed "imperative" commands like "print something to the screen".
- In pure functions, no such interaction with the operating system / user / … is possible.
- And clearly it should not be, since it would defy referential transparency.
- But there is a special `do`-notation in Haskell that enables interaction, and from which one can call "normal" functions.
- All the features and abstraction concepts (higher-order, polymorphism, …) of Haskell remain available even in and with `do`-code.

Input / Output in Haskell, very simple example

- Getting two numbers from the user and then printing some value computed from them to the screen:

```
main :: IO ()
main = do n <- readLn
          m <- readLn
          print (prod [n..m])

prod :: [Integer] -> Integer
prod []     = 1
prod (x:xs) = x * prod xs
```

- Note the (apparent) type inference on `n` and `m`.

Input / Output in Haskell, the principles

- There is a predefined type constructor `IO`, such that for every type like `Int`, `Tree Bool`, `[(Int,Bool)]` etc., the type `IO Int`, `IO (Tree Bool)`, … can be built.
- The interpretation of a type `IO a` is that elements of that type are not themselves concrete values, but instead are (potentially arbitrarily complex) sequences of input and output operations, and computations depending on values read in, by which ultimately a value of type `a` is created.
- An (independently executable) Haskell program overall always has an "`IO` type", usually `main :: IO ()`.

Input / Output in Haskell, the principles

- To actually create "`IO` values", there are certain predefined primitives (and one can recognize their `IO`-related character based on their types).
- For example, there are `getChar :: IO Char` and `putChar :: Char -> IO ()`.
- Also, for multiple characters, `getLine :: IO String` and `putStr`, `putStrLn :: String -> IO ()`.
- More abstractly, for any type for which Haskell knows (or was instructed) how to convert from or to strings, `readLn :: Read a => IO a` for input as well as `print :: Show a => a -> IO ()` for output.

Input / Output in Haskell, the principles

To combine `IO`-computations (i.e., to build more complex action sequences based on the `IO` primitives), we can use the `do`-notation.

Its general form is: $\texttt{do}\ \texttt{cmd}_1$
$\qquad\qquad\qquad\qquad\quad \texttt{x}_2 \texttt{ <- } \texttt{cmd}_2$
$\qquad\qquad\qquad\qquad\quad \texttt{x}_3 \texttt{ <- } \texttt{cmd}_3$
$\qquad\qquad\qquad\qquad\quad \texttt{cmd}_4$
$\qquad\qquad\qquad\qquad\quad \texttt{x}_5 \texttt{ <- } \texttt{cmd}_5$
$\qquad\qquad\qquad\qquad\quad \texttt{...}$

where each $\texttt{cmd}_\texttt{i}$ has an `IO` type and to each $\texttt{x}_\texttt{i}$ (if present) a value of the type encapsulated in the $\texttt{cmd}_\texttt{i}$ will be bound (for use in the rest of the `do`-block), namely exactly the result of executing $\texttt{cmd}_\texttt{i}$.

Input / Output in Haskell, the principles

- The `do`-block as a whole has the type of the last $\texttt{cmd}_n$.
- For that last command, generally no $\texttt{x}_n$ is present.
- Often also useful (for example, at the end of a `do`-block): a predefined function `return :: a -> IO a` that simply yields its argument, without any actual `IO` action.
- What is never ever, at all, possible or allowed is to directly extract (beyond the explicit sequentialisation and binding structure within `do`-blocks) the encapsulated value from an `IO` computation, i.e., to simply turn an `IO a` value into an `a` value.

User defined “control structures”

- As mentioned, also in the context of `IO`-computations, all abstraction concepts of Haskell are available, particularly polymorphism and definition of higher-order functions.
- This can be employed for defining things like:

```
while :: a -> (a -> Bool) -> (a -> IO a)
         -> IO a
while a p body = loop a
  where loop x = if p x then do x' <- body x
                                loop x'
                        else return x
```

- Which can then be used thus:

```
 while 0
       (< 10)
       (\n -> do {print n; return (n+1)})
```

Programming Paradigms – Prolog part

Summer Term 2021

Prof. Janis Voigtländer
University of Duisburg-Essen

Programming Paradigms

Prolog Basics

Prolog in simplest case: facts and queries

- A kind of data base with a number of facts:

```
woman(mia).
woman(jody).
woman(yolanda).
playsAirGuitar(jody).
```

- Queries:

```
?- woman(mia).
true.

?- playsAirGuitar(jody).
true.

?- playsAirGuitar(mia).
false.

?- playsAirGuitar(vincent).
false.

?- playsPiano(jody).
false.
```

The dot is essential!

or an error message

Facts + simple implications

“if”

Head

Body

```
happy(yolanda).
listens2Music(mia).
listens2Music(yolanda) :- happy(yolanda).
playsAirGuitar(mia) :- listens2Music(mia).
playsAirGuitar(yolanda) :- listens2Music(yolanda).
```

- Queries:

```
?- playsAirGuitar(mia).
true.

?- playsAirGuitar(yolanda).
true.
```

because of:

```
happy(yolanda)
  ⇒ listens2Music(yolanda)
  ⇒ playsAirGuitar(yolanda)
```

More complex rules

“and”

```
happy(vincent).
listens2Music(butch).
playsAirGuitar(vincent) :- listens2Music(vincent),
                           happy(vincent).
playsAirGuitar(butch) :- happy(butch).
playsAirGuitar(butch) :- listens2Music(butch).
```

Alternatives

- Queries:

```
?- playsAirGuitar(vincent).
false.

?- playsAirGuitar(butch).
true.
```

- Alternative notation:

“or”

```
...
playsAirGuitar(butch) :- happy(butch);
                         listens2Music(butch).
```

Relations, and more complex queries

```
woman(mia).
woman(jody).
woman(yolanda).

loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).
loves(vincent,vincent).
```

multi-ary (concretely, binary) predicate

- Queries:

```
?- woman(X).
X = mia ;
X = jody ;
X = yolanda.

?- loves(vincent,X).
X = mia ;
X = vincent.

?- loves(vincent,X), woman(X).
X = mia ;
false.
```

semicolon entered by user

Variables in rules (not just in queries)

```
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

jealous(X,Y) :- loves(X,Z), loves(Y,Z).
```

- Queries:

```
?- jealous(marsellus,X).
X = vincent ;
X = marsellus ;
false.

?- jealous(X,_).
X = vincent ;
X = vincent ;
X = marsellus ;
X = marsellus ;
X = mia.
```

anonymous variable

Variables in rules (not just in queries)

```
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

jealous(X,Y) :- loves(X,Z), loves(Y,Z), X \= Y.
```

important that at end

- Queries:

```
?- jealous(marsellus,X).
X = vincent ;
false.

?- jealous(X,_).
X = vincent ;
X = marsellus ;
false.

?- jealous(X,Y).
X = vincent,
Y = marsellus ;
X = marsellus,
Y = vincent ;
false.
```

Some observations on variables

```
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

jealous(X,Y) :- loves(X,Z), loves(Y,Z), X \= Y.
```

- Variables in rules and in queries are independent from each other.

```
?- jealous(marsellus,X).
X = vincent ;
false.
```

- Within a rule or a query, the same variables represent the same objects.
- But different variables do not necessarily represent different objects.
- It is possible to have several occurrences of the same variable in a rule's head!
- In a rule's body there can be variables that do not occur in its head!

Intuition on "free" variables

```
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

jealous(X,Y) :- loves(X,Z), loves(Y,Z), X \= Y.
```

- What is the "logical" interpretation of `Z` above? (or of the whole rule?)
- Possibly, for arbitrary (but fixed) `X`, `Y`:
  <u>if</u> for every choice of `Z` holds: `loves(X,Z)`, and `loves(Y,Z)`, and `X \= Y`,
  <u>then</u> also holds: `jealous(X,Y)`
- Or, for arbitrary (but fixed) `X`, `Y`:
  for every choice of `Z` holds: <u>if</u> `loves(X,Z)`, and `loves(Y,Z)`, and `X \= Y`,
  <u>then</u> also holds: `jealous(X,Y)`

???

Intuition on “free” variables

```
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

jealous(X,Y) :- loves(X,Z), loves(Y,Z), X \= Y.
```

- What is the “logical” interpretation of `Z` above? (or of the whole rule?)

- Possibly, for arbitrary (but fixed) `X` , `Y`:
  if for every choice of `Z` holds: `loves(X,Z)`, and `loves(Y,Z)`, and `X \= Y`,
  then also holds: `jealous(X,Y)`

- Or, for arbitrary (but fixed) `X` , `Y`:
  for every choice of `Z` holds: if `loves(X,Z)`, and `loves(Y,Z)`, and `X \= Y`,
  then also holds: `jealous(X,Y)`

Intuition on “free” variables

```
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

jealous(X,Y) :- loves(X,Z), loves(Y,Z), X \= Y.
```

- What is the “logical” interpretation of `Z` above? (or of the whole rule?)

- Or, for arbitrary (but fixed) `X` , `Y`:
  for every choice of `Z` holds: if `loves(X,Z)`, and `loves(Y,Z)`, and `X \= Y`,
  then also holds: `jealous(X,Y)`

- Logically equivalent, for arbitrary (but fixed) `X` , `Y`:
  if for any choice of `Z` holds: `loves(X,Z)`, and `loves(Y,Z)`, and `X \= Y`,
  then also holds: `jealous(X,Y)`

Programming Paradigms

Operational intuition for Prolog

Operationalisation?

Specification (program) ≡ relation definitions

```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :-
        istVaterVon(G,V),istVaterVon(V,E).
istGrossvaterVon(G,E) :-
        istVaterVon(G,M),istMutterVon(M,E).
```

```
?- istGrossvaterVon(kurt,X)
  ⇝ ...
  ⇝ ...
  ⇝ ...
  ⇝ ...
  ⇝ X = paul ;  X = hans
```

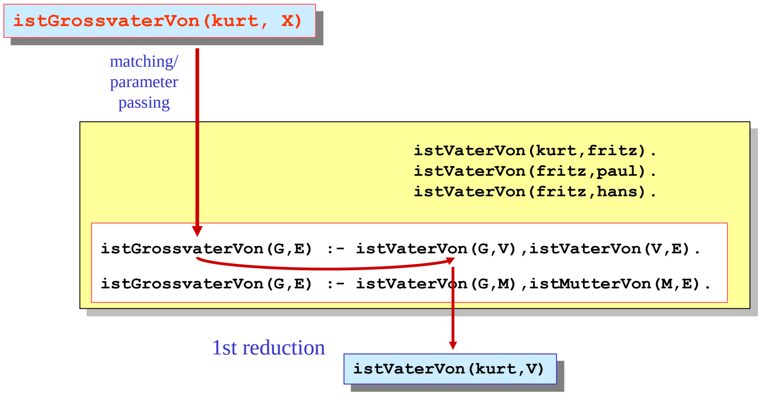
Input: a query

(repeated) resolution

Output: variable substitution(s)

Operationalisation in Prolog (1)

Principle: reduction to subproblems



```
istGrossvaterVon(kurt, X)
```

matching/
parameter
passing

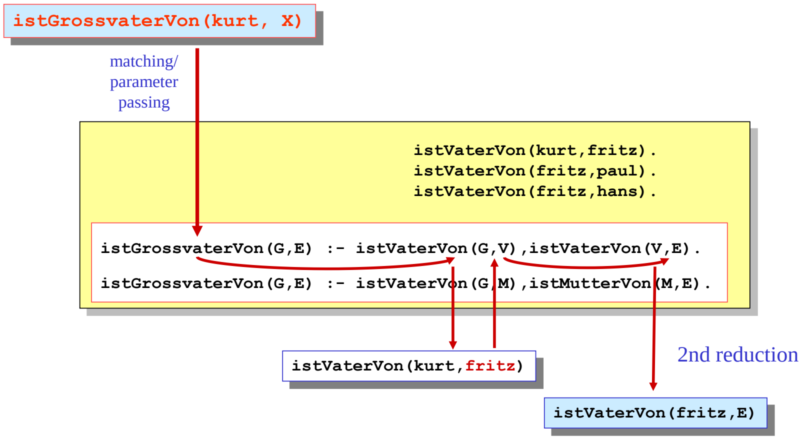
```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :- istVaterVon(G,V),istVaterVon(V,E).
istGrossvaterVon(G,E) :- istVaterVon(G,M),istMutterVon(M,E).
```

1st reduction

```
istVaterVon(kurt,V)
```

Operationalisation in Prolog (2)

Principle: reduction to subproblems, where new subqueries are found <u>from left to right</u>!



```
istGrossvaterVon(kurt, X)
```

matching/
parameter
passing

```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :- istVaterVon(G,V),istVaterVon(V,E).
istGrossvaterVon(G,E) :- istVaterVon(G,M),istMutterVon(M,E).
```

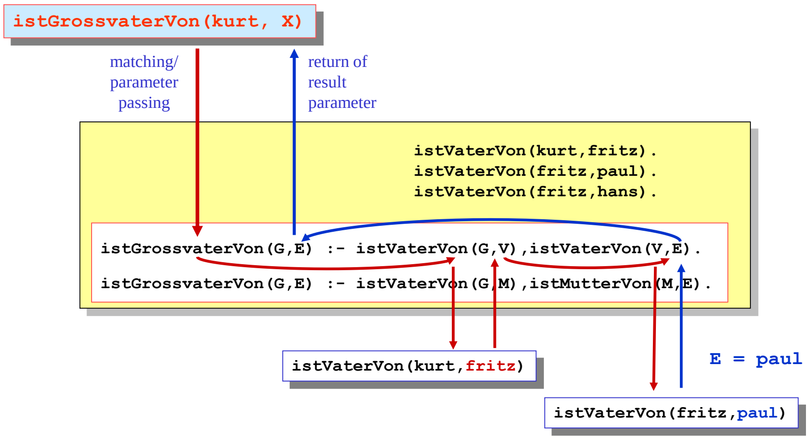
```
istVaterVon(kurt,fritz)
```

2nd reduction

```
istVaterVon(fritz,E)
```

Operationalisation in Prolog (3)

Principle: reduction to subproblems



`istGrossvaterVon(kurt, X)`

matching/ parameter passing

return of result parameter

```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :- istVaterVon(G,V),istVaterVon(V,E).
istGrossvaterVon(G,E) :- istVaterVon(G,M),istMutterVon(M,E).
```

`istVaterVon(kurt,fritz)`

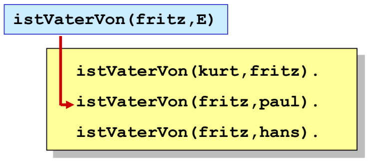
`E = paul`

`istVaterVon(fritz,paul)`

Operationalisation in Prolog (4)

- Prolog always looks for matching rules or facts from top to bottom in the program.

subquery: `istVaterVon(fritz,E)`



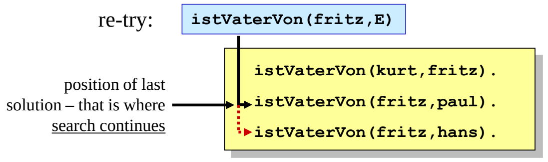
```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).
```

solution:

`E = paul`

- Since a relation generally is not a unique mapping, further answers for a (sub)query may exist. Prolog finds those using backtracking:

re-try: `istVaterVon(fritz,E)`



position of last solution – that is where search continues

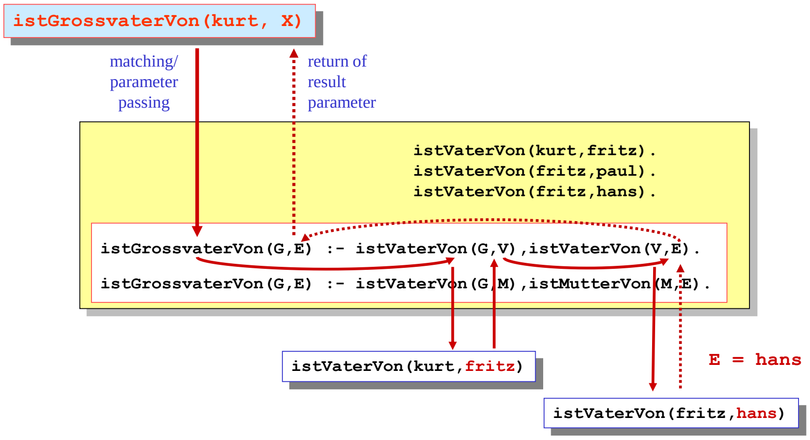
```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).
```

solution:

`E = paul;`
`E = hans`

Operationalisation in Prolog (5)

Principle: reduction to subproblems



`istGrossvaterVon(kurt, X)`

matching/ parameter passing

return of result parameter

```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :- istVaterVon(G,V),istVaterVon(V,E).
istGrossvaterVon(G,E) :- istVaterVon(G,M),istMutterVon(M,E).
```

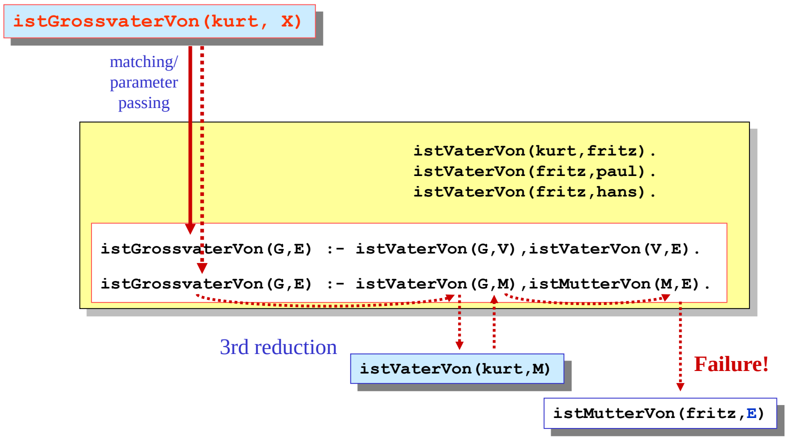
`istVaterVon(kurt,fritz)`

`E = hans`

`istVaterVon(fritz,hans)`

Operationalisation in Prolog (6)

The backtracking also concerns further matching rules:



```
istGrossvaterVon(kurt, X)
```

matching/
parameter
passing

```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :- istVaterVon(G,V),istVaterVon(V,E).
istGrossvaterVon(G,E) :- istVaterVon(G,M),istMutterVon(M,E).
```

3rd reduction

```
istVaterVon(kurt,M)
```

Failure!

```
istMutterVon(fritz,E)
```

Operationalisation on the example, presented differently

```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :-
        istVaterVon(G,V),istVaterVon(V,E).
istGrossvaterVon(G,E) :-
        istVaterVon(G,M),istMutterVon(M,E).
```

```
?- istGrossvaterVon(kurt, X).
?- istVaterVon(kurt, V), istVaterVon(V, X).
?- istVaterVon(fritz, X).
?- .
```

X = paul:

Compare (within a Prolog system): use of ?- trace.

Operationalisation on the example, presented differently

```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :-
        istVaterVon(G,V),istVaterVon(V,E).
istGrossvaterVon(G,E) :-
        istVaterVon(G,M),istMutterVon(M,E).
```

```
?- istGrossvaterVon(kurt, X).
?- istVaterVon(kurt, V), istVaterVon(V, X).
?- istVaterVon(fritz, X).
?- .
?- .
```

X = paul:
X = hans:

Compare (within a Prolog system): use of ?- trace.

Operationalisation on the example, presented differently

```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :-
       istVaterVon(G,V),istVaterVon(V,E).
istGrossvaterVon(G,E) :-
       istVaterVon(G,M),istMutterVon(M,E).
```

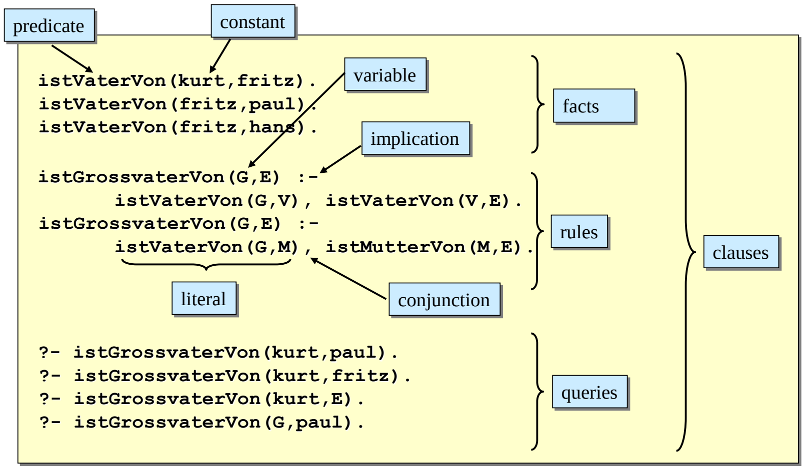
```
           ?- istGrossvaterVon(kurt, X).
           ?- istVaterVon(kurt, V), istVaterVon(V, X).
           ?- istVaterVon(fritz, X).
X = paul:  ?- .
X = hans:  ?- .
           ?- istVaterVon(kurt, M), istMutterVon(M, X).
           ?- istMutterVon(fritz, X).
           Failure!
```

Compare (within a Prolog system): use of ?- trace.

Programming Paradigms

Syntactical ingredients of Prolog

Syntax / notions in Prolog



```
istVaterVon(kurt,fritz).
istVaterVon(fritz,paul).
istVaterVon(fritz,hans).

istGrossvaterVon(G,E) :-
      istVaterVon(G,V), istVaterVon(V,E).
istGrossvaterVon(G,E) :-
      istVaterVon(G,M), istMutterVon(M,E).

?- istGrossvaterVon(kurt,paul).
?- istGrossvaterVon(kurt,fritz).
?- istGrossvaterVon(kurt,E).
?- istGrossvaterVon(G,paul).
```

Syntactical objects in Prolog

- To build clauses, Prolog uses different pieces:

- constants	(numbers, atoms – mainly lowercase identifiers, …)
- variables	(X,Y, ThisThing, _, _G107…)
- operator terms	(… 1 + 3 * 4 …)
- structures	(date(27,11,2007), person(fritz, mueller), … composite, recursive, “infinite”, …)

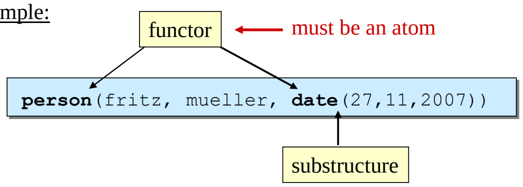
- Note: Prolog has no type system!

Syntactical objects in Prolog

Structures in Prolog

- Structures represent objects that are made up of other objects (like trees and subtrees).

- Example:



functor ← must be an atom

`person(fritz, mueller, date(27,11,2007))`

substructure

functors: `person/3, date/3` (notation for arity)

- Through this, modelling of essentially “algebraic data types” – but not actually typed. So, `person(1,2,'a')` would also be a legal structure.

- Arbitrary nesting depth allowed – in principle infinite.

Syntactical objects in Prolog

Predefined syntax for special structures:

- There is a predefined “list type” as recursive data structure:

```
[1,2,a]   .(1,.(2,.(a,[])))   [1|[2,a]]   [1,2|[a]]   [1,2|.(a,[])]
```

- Character strings are represented as lists of ASCII-Codes:

`"Prolog"` = [80, 114, 111, 108, 111, 103]
= .(80, .(114, .(111, .(108, .(111, .(103, [ ]))))))

Operators:

- Operators are functors/atoms made from symbols and can be written infix.

- Example: in arithmetic expressions
  - Mathematical functions are defined as operators.
  - `1 + 3 * 4` is to be read as this structure: `+(1,*(3,4))`

Syntactical objects in Prolog

Collective notion "terms":

- Terms are constants, variables or structures:

```
fritz
27
MM
[europe, asia, africa | Rest]
person(fritz, Lastname, date(27, MM, 2007))
```

- A ground term is a term that does not contain variables:

```
person(fritz, mueller, date(27, 11, 2007))
```

Programming Paradigms

More Prolog examples

Simple example for working with data structures

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

```
?- add(s(0),s(0),s(s(0))).
true.

?- add(s(0),s(0),N).
N = s(s(0)) ;
false.
```

- Recall, in Haskell:

```
data Nat = Zero | Succ Nat

add :: Nat → Nat → Nat
add Zero       x = x
add (Succ x)  y = Succ (add x y)
```

Systematic connection/derivation?

- Essential difference Haskell/Prolog:

Functions vs. Predicates/Relations

f x y = z “corresponds to” `p(X,Y,Z).`

- First a somewhat naïve attempt to exploit this correspondence:

```
add Zero     x = x
```

↓

add(Zero, x, x)

↓

```
add(0,X,X).
```

```
add (Succ x)  y = Succ (add x y)
```

↓

add(Succ x, y, Succ (add x y))

↓

???

Systematic connection/derivation?

- Essential difference Haskell/Prolog:

Functions vs. Predicates/Relations

f x y = z “corresponds to” `p(X,Y,Z).`

- Systematically avoiding nested function calls:

```
add (Succ x)  y = Succ (add x y)
```

↓

```
add (Succ x)  y = Succ z    where z = add x y
```

↓

add(Succ x, y, Succ z) if add(x, y, z)

↓

```
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

On the flexibility of Prolog predicates

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

```
?- add(N,M,s(s(0))).
N = 0,
M = s(s(0)) ;
N = s(0),
M = s(0) ;
N = s(s(0)),
M = 0 ;
false.

?- add(N,s(0),s(s(0))).
N = s(0) ;
false.

?- add(N,M,O).
```

???

On the flexibility of Prolog predicates

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

sub(X,Y,Z) :- add(Z,Y,X).
```

```
?- sub(s(s(0)),s(0),N).
N = s(0) ;
false.

?- sub(N,M,s(0)).
N = s(M) ;
false.
```

Another example

Computing the length of a list in Haskell:

```
length []       =  0
length (x:xs)   =  length xs + 1
```

Computing the length of a list in Prolog:

```
length([],0).
length([X|Xs],N) :- length(Xs,M), N is M+1.
```

```
?- length([1,2,a],Res).
   Res = 3.

?- length(List,3).
   List = [_G331, _G334, _G337]
```

list with 3 arbitrary (variable) elements

Arithmetics vs. symbolic operator terms

Caution: If instead of:

```
length([],0).
length([X|Xs],N) :- length(Xs,M), N is M+1.
```

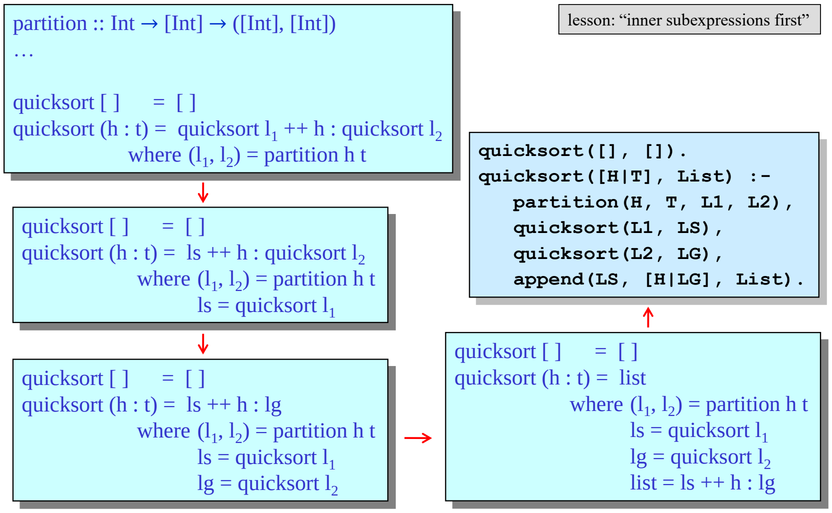
we use:

```
length([],0).
length([X|Xs],M+1) :- length(Xs,M).
```

then:

```
?- length([1,2,a],Res).
   Res = 0+1+1+1.

?- length(List,3).
   false.

?- length(List,0+1+1+1).
   List = [_G331, _G334, _G337].
```

An example corresponding to several nested calls



lesson: "inner subexpressions first"

partition :: Int → [Int] → ([Int], [Int])
…

quicksort [ ] = [ ]
quicksort (h : t) = quicksort l_1 ++ h : quicksort l_2
where (l_1, l_2) = partition h t

quicksort [ ] = [ ]
quicksort (h : t) = ls ++ h : quicksort l_2
where (l_1, l_2) = partition h t
ls = quicksort l_1

quicksort [ ] = [ ]
quicksort (h : t) = ls ++ h : lg
where (l_1, l_2) = partition h t
ls = quicksort l_1
lg = quicksort l_2

quicksort [ ] = [ ]
quicksort (h : t) = list
where (l_1, l_2) = partition h t
ls = quicksort l_1
lg = quicksort l_2
list = ls ++ h : lg

```
quicksort([], []).
quicksort([H|T], List) :-
   partition(H, T, L1, L2),
   quicksort(L1, LS),
   quicksort(L2, LG),
   append(LS, [H|LG], List).
```

Programming Paradigms

Declarative semantics of Prolog

Declarative semantics of Prolog

What is the "mathematical" meaning/semantics of a Prolog program?

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

Logical interpretation:

$$(\forall\ X.\ add(0,X,X))$$
$$\wedge\ (\forall\ X, Y, Z.\ add(X,Y,Z)\ \Rightarrow\ add(s(X),Y,s(Z)))$$

To give meaning to such formulas, the study of logics uses models:

- starting from a set of mathematical objects
- interpretation of constants (like "0") as elements of the above set, and of functors (like "s(…)") as functions thereover
- interpretation of predicates (like "add(…)") as relations between objects
- assignment of truth values to formulas according to certain rules
- consideration only of interpretations that make all given formulas true (these specific interpretations are called models)

Semantics of a program would be given by all statements/relationships that hold in all models for the program.

Herbrand models

Important: There is always a kind of "universal model".

Idea: Interpretation as simple as possible, namely purely syntactic.
Neither functors nor predicates really "do" anything.

the Herbrand universe

So: set of objects = all ground terms (over implicitly given signature)
interpretation of functors = syntactical application on terms
interpretation of predicates = some set of applications of predicate symbols on ground terms

a Herbrand interpretation

Example:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

Signature: `0` (of arity 0), `s` (of arity 1)
Herbrand universe: {`0`, `s(0)`, `s(s(0))`, `s(s(s(0)))`, …} (without predicate symbols!)
the Herbrand base: {`add(0,0,0)`, `add(0,0,s(0))`, `add(0,s(0),0)`, …}
(all applications of predicate symbols on terms from Herbrand universe)

Smallest Herbrand model

Can one compute, in a constructive fashion, the smallest (via the subset relation) Herbrand interpretation that is a model?

Yes, using the "immediate consequence operator": T_P

Definition: T_P takes a Herbrand interpretation I and produces all ground literals (elements of the Herbrand base) L_0 for which L_1, L_2, ..., L_n exist in I such that L_0 :- L_1, L_2, ..., L_n is a complete instantiation (i.e., no variables left) of any of the given program clauses (facts/rules).

The smallest Herbrand model is obtained as fixpoint/limit of the sequence

$$\emptyset\ ,\ T_P(\emptyset)\ ,\ T_P(T_P(\emptyset))\ ,\ T_P(T_P(T_P(\emptyset)))\ ,\ \ldots$$

Smallest Herbrand model

On the example:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

$T_P(\emptyset)$ = {`add(0,0,0)`, `add(0,s(0),s(0))`, `add(0,s(s(0)),s(s(0)))`, …}

$T_P(T_P(\emptyset))$ = $T_P(\emptyset) \cup$ {`add(s(0),0,s(0))`, `add(s(0),s(0),s(s(0)))`,
`add(s(0),s(s(0)),s(s(s(0))))`, …}

$T_P(T_P(T_P(\emptyset)))$ = $T_P(T_P(\emptyset)) \cup$ {`add(s(s(0)),0,s(s(0)))`,
`add(s(s(0)),s(0),s(s(s(0))))`,
`add(s(s(0)),s(s(0)),s(s(s(s(0)))))`, …}

…

In the limit: $\{\ \mathtt{add}(\mathtt{s}^{\mathtt{i}}(\mathtt{0}),\mathtt{s}^{\mathtt{j}}(\mathtt{0}),\mathtt{s}^{\mathtt{i+j}}(\mathtt{0}))\ |\ \mathtt{i},\mathtt{j} \geq 0\ \}$

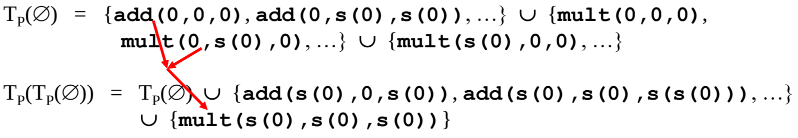
Applicability of the semantics based on Herbrand models

For which kind of Prolog programs can one work with the T_P -semantics?

- no arithmetics, no **`is`**
- no **`\=`**, no **`not`**
- generally, none of the “non-logical” features (not introduced in the lecture)

But for example programs like this (and would also work for mutual recursion):

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),s(Y),s(Z)) :- mult(X,s(Y),U), add(Y,U,Z).
```

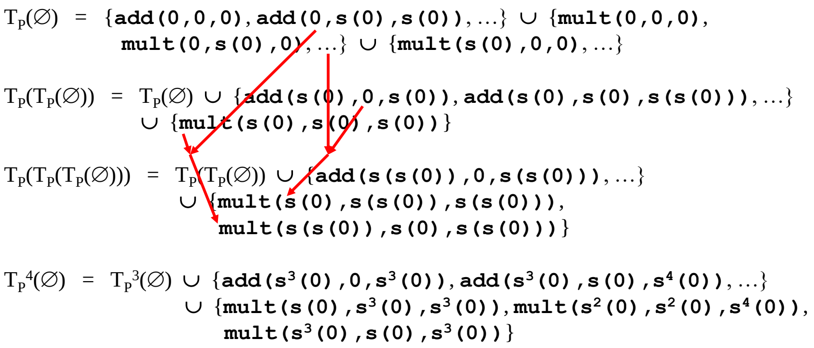


$$T_P(\varnothing) \;=\; \{\mathtt{add(0,0,0)}, \mathtt{add(0,s(0),s(0))}, \ldots\} \cup \{\mathtt{mult(0,0,0)}, \mathtt{mult(0,s(0),0)}, \ldots\} \cup \{\mathtt{mult(s(0),0,0)}, \ldots\}$$

$$T_P(T_P(\varnothing)) \;=\; T_P(\varnothing) \cup \{\mathtt{add(s(0),0,s(0))}, \mathtt{add(s(0),s(0),s(s(0)))}, \ldots\} \cup \{\mathtt{mult(s(0),s(0),s(0))}\}$$

Applicability of the semantics based on Herbrand models

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),s(Y),s(Z)) :- mult(X,s(Y),U), add(Y,U,Z).
```



$$T_P(\varnothing) \;=\; \{\mathtt{add(0,0,0)}, \mathtt{add(0,s(0),s(0))}, \ldots\} \cup \{\mathtt{mult(0,0,0)}, \mathtt{mult(0,s(0),0)}, \ldots\} \cup \{\mathtt{mult(s(0),0,0)}, \ldots\}$$

$$T_P(T_P(\varnothing)) \;=\; T_P(\varnothing) \cup \{\mathtt{add(s(0),0,s(0))}, \mathtt{add(s(0),s(0),s(s(0)))}, \ldots\} \cup \{\mathtt{mult(s(0),s(0),s(0))}\}$$

$$T_P(T_P(T_P(\varnothing))) \;=\; T_P(T_P(\varnothing)) \cup \{\mathtt{add(s(s(0)),0,s(s(0)))}, \ldots\} \cup \{\mathtt{mult(s(0),s(s(0)),s(s(0)))}, \mathtt{mult(s(s(0)),s(0),s(s(0)))}\}$$

$$T_P{}^4(\varnothing) \;=\; T_P{}^3(\varnothing) \cup \{\mathtt{add(s^3(0),0,s^3(0))}, \mathtt{add(s^3(0),s(0),s^4(0))}, \ldots\} \cup \{\mathtt{mult(s(0),s^3(0),s^3(0))}, \mathtt{mult(s^2(0),s^2(0),s^4(0))}, \mathtt{mult(s^3(0),s(0),s^3(0))}\}$$

Applicability of the semantics based on Herbrand models

The declarative semantics:

- is only applicable to certain, “purely logical”, programs
- does not directly describe the behaviour for queries containing variables
- is mathematically simpler than the still to be introduced operational semantics
- can be related to that operational semantics appropriately
- is insensitive against changes to the order of, and within, facts and rules (!)

Programming Paradigms

Operational semantics of Prolog

Motivation: Observing some not so nice (not so “logical”?) effects

```
direct(frankfurt,san_francisco).
direct(frankfurt,chicago).
direct(san_francisco,honolulu).
direct(honolulu,maui).

connection(X, Y) :- direct(X, Y).
connection(X, Y) :- direct(X, Z), connection(Z, Y).
```

```
?- connection(frankfurt,maui).
true.

?- connection(san_francisco,X).
X = honolulu ;
X = maui ;
false.

?- connection(maui,X).
false.
```

Motivation: Observing some not so nice (not so “logical”?) effects

```
direct(frankfurt,san_francisco).
direct(frankfurt,chicago).
direct(san_francisco,honolulu).
direct(honolulu,maui).

connection(X, Y) :- connection(X, Z), direct(Z, Y).
connection(X, Y) :- direct(X, Y).
```

```
?- connection(frankfurt,maui).
ERROR: Out of local stack
```

- Apparently, the implicit logical operations are not commutative.
- So concerning program execution, there must be more than the purely logical reading.

Somewhat more subtle…

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

sub(X,Y,Z) :- add(Z,Y,X).
```

```
?- sub(N,M,s(0)).
N = s(M) ;
false.
```

```
add(X,0,X).
add(X,s(Y),s(Z)) :- add(X,Y,Z).

sub(X,Y,Z) :- add(Z,Y,X).
```

```
?- sub(s(s(0)),s(0),N).
N = s(0) ;
false.

?- sub(N,M,s(0)).
N = s(0),
M = 0 ;
N = s(s(0)),
M = s(0) ;
```

So the choice/treatment of the order of arguments in definitions affects the quality of results.

…

… and (thus) sometimes less flexibility than desired

The nicely descriptive solution:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```

works very well for various kinds of queries:

```
?- mult(s(s(0)),s(s(s(0))),N).
N = s(s(s(s(s(s(0)))))).

?- mult(s(s(0)),N,s(s(s(s(0))))).
N = s(s(0)) ;
false.
```

We say that `mult` supports the "call modes" `mult(+X,+Y,?Z)` and `mult(+X,?Y,+Z)`

But there are also "outliers":

```
?- mult(N,M,s(s(s(s(0))))).
N = s(0),
M = s(s(s(s(0)))) ;
N = s(s(0)),
M = s(s(0)) ;
abort
```

… but not `mult(?X,?Y,+Z)`.

otherwise infinite search

… and (thus) sometimes less flexibility than desired

Whereas with just addition:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

the analogous call mode seemed to work pretty well:

```
?- add(N,M,s(s(s(0)))).
N = 0,
M = s(s(s(0))) ;
N = s(0),
M = s(s(0)) ;
N = s(s(0)),
M = s(0) ;
N = s(s(s(0))),
M = 0 ;
false.
```

Indeed, `add` supports all call modes, even `add(?X,?Y,?Z)`.

1. So why the difference?
2. And what can we do to also let **`mult`** function this way?

Moreover, caution needed when using/positioning negative literals

And now it gets really "strange":

```
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

jealous(X,Y) :- loves(X,Z), loves(Y,Z), X \= Y.
```

small change

```
...

jealous(X,Y) :- X \= Y, loves(X,Z), loves(Y,Z).
```

```
?- jealous(marsellus,X).
false.

?- jealous(X,_).
false.

?- jealous(X,Y).
false.
```

Whereas before the small change, we got meaningful results for these queries!

Operational semantics of Prolog

To investigate all these phenomena, we have to consider the concrete execution mechanism of Prolog.

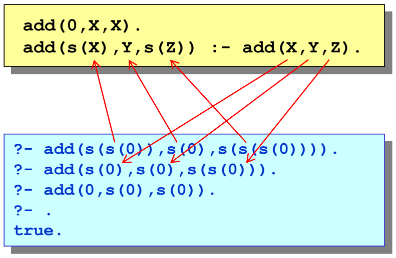
Ingredients for this discussion of the operational semantics, considered in what follows:

1. Unification
2. Resolution
3. Derivation trees

Programming Paradigms

Unification

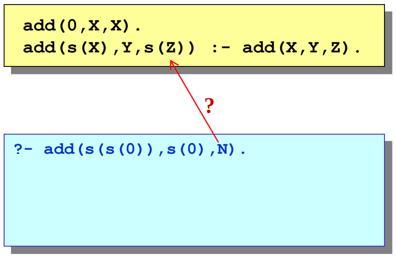
Analogy to Haskell: Pattern matching



```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

```
?- add(s(s(0)),s(0),s(s(s(0)))).
?- add(s(0),s(0),s(s(0))).
?- add(0,s(0),s(0)).
?- .
true.
```

But what about “output variables”?



```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

```
?- add(s(s(0)),s(0),N).
```

In some sense, we need a form of “bidirectional pattern matching”, that can also combine and propagate variable bindings.

Equality of terms (1)

- Checking equality of ground terms:

```
europe = europe ?                                          yes
person(fritz,mueller) = person(fritz,mueller) ?            yes
person(fritz,mueller) = person(mueller,fritz) ?            no
5 = 2 ?                                                    no
5 = 2 + 3 ?                                                no
2 + 3 = +(2, 3) ?                                          yes
```

⇒ Equality of terms means structural equality.

Terms are not “evaluated” before a comparison!

Equality of terms (2)

- Checking equality of terms with variables:

```
person(fritz, Lastname, datum(27, 11, 2007))
    = person(fritz, mueller, datum(27, MM, 2007)) ?
```

- For a variable, any term may be substituted:
  - in particular `mueller` for `Lastname` and `11` for `MM`.
  - After this substitution both terms are equal.

Equality of terms (3)

Which variables have to be substituted how, in order to make the terms equal?

```
          date(1, 4, 1985) = date(1, 4, Year) ?
  date(Day, Month, 1985) = date(1, 4, Year) ?
  a(b,C,d(e,F,g(h,i,J))) = a(B,c,d(E,f,g(H,i,K))) ?
                       X = Y + 1 ?
          [[the, Y]|Z] = [[X, dog], [is, here]] ?
```

As a reminder, list syntax:

```
[1,2,a] = [1|[2,a]] = [1,2|[a]] = [1,2|.(a,[])] = .(1,.(2,.(a,[])))
```

And what about: `p(X) = p(q(X)) ?`

“occurs check” (implementation detail)

Equality of terms (4)

Some further (problematic) cases:

```
      loves(vincent, X) = loves(X, mia) ?
  loves(marsellus, mia) = loves(X, X) ?
a(b,C,d(e,F,g(h,i,J))) = a(B,c,d(E,f,p(H,i,K))) ?
                p(b,b) = p(X) ?
                      …
```

Unification concepts, somewhat formally (1)

Substitution:

- Replacing variables by other variables or other kinds of terms (constants, structures, …)
- Extended to a function which uniquely maps each term to a new term, where the new term differs from the old term only by the replacement of variables.
- Notation: U = {`Lastname / mueller, MM / 11`}
- This substitution U changes only the variables `Lastname` and `MM` (in context), everything else stays unchanged.
- U(`person(fritz, Lastname, datum(27, 11, 2007))`)
  `== person(fritz, mueller, datum(27, 11, 2007))`

Unification concepts, somewhat formally (2)

- Unifier:
  - substitution that makes two terms equal
  - e.g., application of the substitution U = { `Lastname/mueller, MM/11` } :

    U(`person(fritz,Lastname,date(27,11,2007))`)
    == U(`person(fritz,mueller,date(27,MM,2007))`)
- Most general unifier:
  - unifier that leaves as many variables as possible unchanged, and does not introduce specific terms where variables suffice
  - Example: `date(DD,MM,2007)` and `date(D,11,Y)`
    - U_1 = { `DD/27, D/27, MM/11, Y/2007` } ✗
    - U_2 = { `DD/D, MM/11, Y/2007` } ✓
- Prolog always looks for a most general unifier.

Unification

We will now skip over some slides with a description of a concrete algorithm for computing most general unifiers.

The main reason is that the lecture “Logik” has already introduced an algorithm for this purpose, and it has been practiced in that course’s exercises.

And for our consideration of the operational semantics of Prolog you do not need to learn a specific unification algorithm by heart, you only need to be able to determine what the most general unifier for a pair of terms **is**.
(We will encounter various examples.)

Aside: The issue of the “occurs check” will not come up in any examples considered in lecture, exercises or exam (though it is relevant in Prolog implementations).

Unification – Computing a most general unifier

Input: two terms T_1 and T_2 (in general possibly containing common variables)

Output: a most general unifier U for T_1 and T_2 in case T_1 and T_2 are unifiable, otherwise failure

Algorithm:

1. If T_1 and T_2 are the same constant or variable, then $U = \emptyset$
2. If T_1 is a variable that does not occur in T_2, then $U = \{T_1 / T_2\}$
3. If T_2 is a variable that does not occur in T_1, then $U = \{T_2 / T_1\}$

“occurs check”

Unification – Computing a most general unifier

Algorithm (cont.):

4. If $T_1 = f(T_{1,1},\ldots,T_{1,n})$ and $T_2 = f(T_{2,1},\ldots,T_{2,n})$ are structures with the same functor and the same number of components, then
   1. Find a most general unifier U_1 for $T_{1,1}$ and $T_{2,1}$
   2. Find a most general unifier U_2 for $U_1(T_{1,2})$ and $U_1(T_{2,2})$

   …

   n. Find a most general unifier U_n for

   $U_{n-1}(\ldots(U_1(T_{1,n})\ldots)$ and $U_{n-1}(\ldots(U_1(T_{2,n}))\ldots)$

   If all these unifiers exist, then

   $U = U_n \circ U_{n-1} \circ \ldots \circ U_1$ (function composition of the unifiers, always applied recursively along term structure)

5. Otherwise: T_1 and T_2 are not unifiable.

Unification algorithm – Examples

`date(1, 4, 1985) = date(1, 4, Year) ?`

Structures with the same functor, same number of components, hence:

1. Find a most general unifier U_1 for `1` and `1`

   ⇒ same constants, thus $U_1 = \emptyset$
2. Find a most general unifier U_2 for U_1(`4`) and U_1(`4`)

   ⇒ same constants, thus $U_2 = \emptyset$
3. Find a most general unifier U_3 for U_2(U_1(`1985`)) and U_2 (U_1(`Year`))

   ⇒ constant vs. variable, thus U_3 = {`Year` /`1985`}

A most general unifier overall is:

$U = U_3 \circ U_2 \circ U_1$ = {`Year` /`1985`}

Unification algorithm – Examples

`loves(marsellus, mia) = loves(X, X)` ?

Structures with the same functor, same number of components, hence:

1. Find a most general unifier U_1 for `marsellus` and `X`

   ⇒ constant vs. variable, thus U_1 = {`X` /`marsellus`}
2. Find a most general unifier U_2 for U_1(`mia`) = `mia` and U_1(`X`) = `marsellus`

   ⇒ different constants, hence U_2 does not exist!

Consequently, also no unifier exists for the original terms!

Unification algorithm – Examples

`d(E,g(H,J)) = d(F,g(H,E))` ?

Structures with the same functor, same number of components, hence:

1. Find a most general unifier U_1 for `E` and `F`

   ⇒ different variables, thus U_1 = {`E`/`F`}
2. Find a most general unifier U_2 for U_1(`g(H,J)`) and U_1(`g(H,E)`)

   `g(H,J) = g(H,F)` ?

   ⇒ Structures with the same functor, same number of components, hence:

   - Find a most general unifier $U_{2,1}$ for `H` and `H`

     ⇒ same variables, thus $U_{2,1} = \varnothing$
   - Find a most general unifier $U_{2,2}$ for $U_{2,1}$(`J`) and $U_{2,1}$(`F`)

     ⇒ different variables, thus $U_{2,2}$ = {`F`/`J`}

   $U_2 = U_{2,2} \circ U_{2,1}$ = {`F`/`J`}

A most general unifier overall is:

$U = U_2 \circ U_1$ = {`E`/`J` , `F`/`J`}

Relevance of the “occurs check”

As a reminder:

2. If T_1 is a variable that does not occur in T_2, then $U = \{T_1 / T_2\}$

3. If T_2 is a variable that does not occur in T_1, then $U = \{T_2 / T_1\}$

“occurs check”

So, for example:

`X = q(X)` ?

⇒ No unifier exists.

But in Prolog this check is actually not performed by default (in can be enabled in implementations, though)!

Relevance of the “occurs check”

Without “occurs check”:

`p(X) = p(q(X))`?

Structures with the same functor, same number of components, hence:

1. Find a most general unifier U_1 for `X` and `q(X)`

   ⇒ variable vs. term, thus U_1 = {`X`/`q(X)`}

$U = U_1$ = {`X`/`q(X)`} !

Although it actually is <u>not</u> true that U(`p(X)`) and U(`p(q(X))`) are equal!

Programming Paradigms

Resolution

Resolution in Prolog (1)

Resolution (proof principle) – without variables

One can reduce proving the query

`?- P, L, Q.` (let `L` be a variable free literal and `P` and `Q` be sequences of such)

to proving the query

$?\!-\ P,\ L_1,\ L_2,\ \ldots\ ,\ L_n,\ Q.$

provided that $L\ :\!-\ L_1,\ L_2,\ \ldots,\ L_n.$ is a clause in the program (where n ≥ 0).

- The choice of the literal `L` and the clause to use are in principle arbitrary.
- If n = 0, then the query becomes smaller by the resolution step.

Resolution in Prolog (2)

Resolution – with variables

One can reduce proving the query

`?- P, L, Q.` (let `L` be a literal and `P` and `Q` be sequences of literals)

to proving the query

$$?\!-\ U(P),\ U(L_1),\ U(L_2),\ \ldots,\ U(L_n),\ U(Q).$$

provided that:

- there is a program clause L_0 `:-` L_1, L_2, ..., L_n. (where n ≥ 0), with – just in case – renamed variables (so that there is no overlap with those in `P, L, Q`),
- and *U* is a most general unifier for `L` and L_0.

Programming Paradigms

Derivation trees

Reminder: Motivation for considering operational semantics…

We wanted to understand why, for example, for

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```

various kinds of queries/“call modes” work very well:

```
?- mult(s(s(0)),s(s(s(0))),N).
N = s(s(s(s(s(s(0)))))).

?- mult(s(s(0)),N,s(s(s(s(0))))).
N = s(s(0)) ;
false.
```

but others don’t:

```
?- mult(N,M,s(s(s(s(0))))).
N = s(0),
M = s(s(s(s(0)))) ;
N = s(s(0)),
M = s(s(0)) ;
abort
```

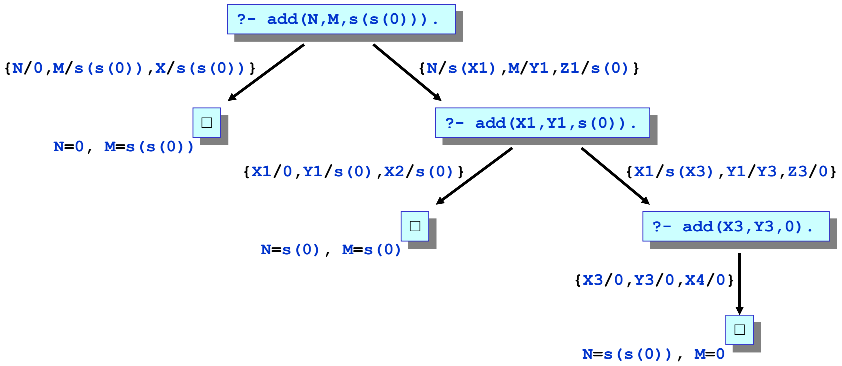
otherwise infinite search

Explicit enumeration of solutions

Let us start with a simple example just for addition:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

Exhaustive search:



?- add(N,M,s(s(0))).

{N/0,M/s(s(0)),X/s(s(0))}

{N/s(X1),M/Y1,Z1/s(0)}

☐

N=0, M=s(s(0))

?- add(X1,Y1,s(0)).

{X1/0,Y1/s(0),X2/s(0)}

{X1/s(X3),Y1/Y3,Z3/0}

☐

N=s(0), M=s(0)

?- add(X3,Y3,0).

{X3/0,Y3/0,X4/0}

☐

N=s(s(0)), M=0

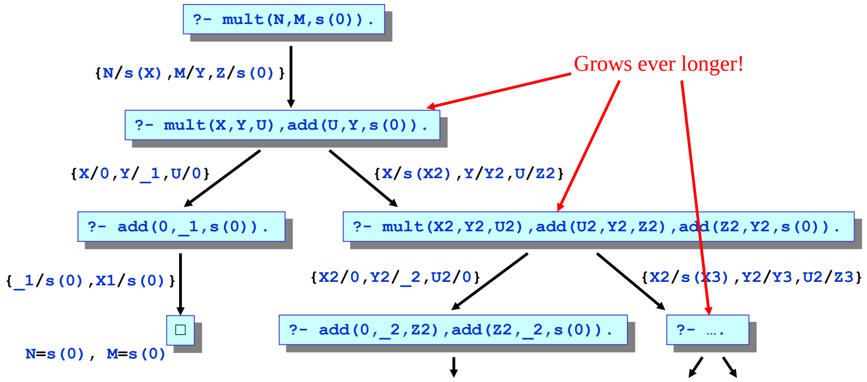
Detailed description of the generation of derivation trees

1. Generate root node with query, remember it as still to be worked on.

2. As long as there are still nodes to be worked on:
   - select left-most such node
   - determine all facts/rules (with consistently renamed variables) whose head is unifiable with the left-most literal in that node
   - generate for each such fact/rule a (still to be worked on) successor node via a resolution step
   - arrange successor nodes from left to right according to the order of appearance of the used facts/rules in the program (from top to bottom)
   - annotate the unifier used in each case
   - mark nodes as finished if they are empty or if their left-most literal is not unifiable with any fact/rule head
   - at successful nodes (the ones that are finished as empty), annotate the solution (the composition of unifiers – as functions on terms – along the path from the root, applied to all variables that occurred in the original query)

An example with infinite search

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```



?- mult(N,M,s(0)).

{N/s(X),M/Y,Z/s(0)}

Grows ever longer!

?- mult(X,Y,U),add(U,Y,s(0)).

{X/0,Y/_1,U/0}

{X/s(X2),Y/Y2,U/Z2}

?- add(0,_1,s(0)).

?- mult(X2,Y2,U2),add(U2,Y2,Z2),add(Z2,Y2,s(0)).

{_1/s(0),X1/s(0)}

{X2/0,Y2/_2,U2/0}

{X2/s(X3),Y2/Y3,U2/Z3}

☐

?- add(0,_2,Z2),add(Z2,_2,s(0)).

?- ….

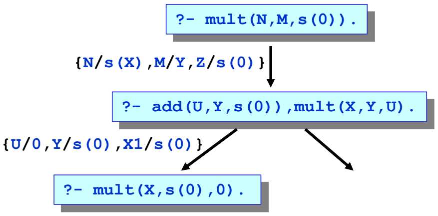
N=s(0), M=s(0)

Experiment with changed order of literals

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```

⬇

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```



`?- mult(N,M,s(0)).`

{N/s(X),M/Y,Z/s(0)}

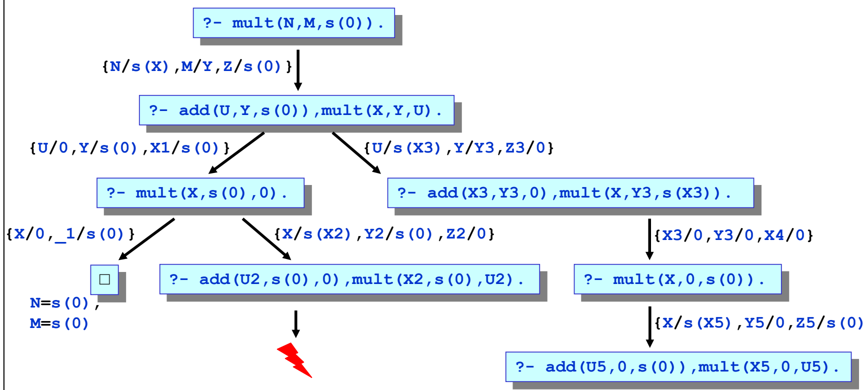
`?- add(U,Y,s(0)),mult(X,Y,U).`

{U/0,Y/s(0),X1/s(0)}

`?- mult(X,s(0),0).`

Experiment with changed order of literals

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```



`?- mult(N,M,s(0)).`

{N/s(X),M/Y,Z/s(0)}

`?- add(U,Y,s(0)),mult(X,Y,U).`

{U/0,Y/s(0),X1/s(0)}

{U/s(X3),Y/Y3,Z3/0}

`?- mult(X,s(0),0).`

`?- add(X3,Y3,0),mult(X,Y3,s(X3)).`

{X/0,_1/s(0)}

{X/s(X2),Y2/s(0),Z2/0}

{X3/0,Y3/0,X4/0}

□

N=s(0),
M=s(0)

`?- add(U2,s(0),0),mult(X2,s(0),U2).`

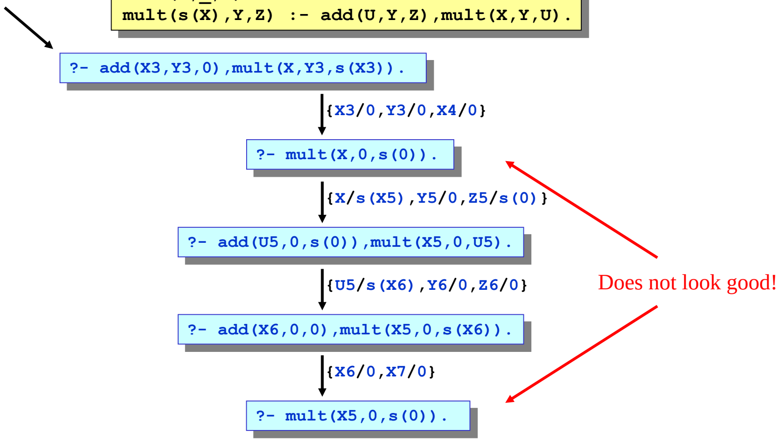
`?- mult(X,0,s(0)).`

{X/s(X5),Y5/0,Z5/s(0)}

`?- add(U5,0,s(0)),mult(X5,0,U5).`

Experiment with changed order of literals

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```



`?- add(X3,Y3,0),mult(X,Y3,s(X3)).`

{X3/0,Y3/0,X4/0}

`?- mult(X,0,s(0)).`

{X/s(X5),Y5/0,Z5/s(0)}

`?- add(U5,0,s(0)),mult(X5,0,U5).`

{U5/s(X6),Y6/0,Z6/0}

`?- add(X6,0,0),mult(X5,0,s(X6)).`

{X6/0,X7/0}

`?- mult(X5,0,s(0)).`

Does not look good!

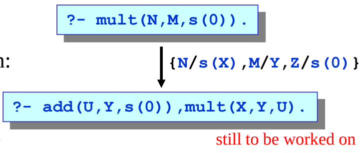
Detailed description of the generation of derivation trees

Input: query and program, for example `mult(N,M,s(0))` and:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

Output: tree, generated by following steps:

1. Generate root node with query, remember it as still to be worked on.
2. As long as there are still nodes to be worked on:
   - select left-most such node
   - determine all facts/rules (with consistently renamed variables) whose head is unifiable with the left-most literal in that node
   - generate for each such fact/rule a (still to be worked on) successor node via a resolution step
   - arrange successor nodes from left to right according to the order of appearance of the used facts/rules in the program (from top to bottom)
   - annotate the unifier used in each case



?- mult(N,M,s(0)).

{N/s(X),M/Y,Z/s(0)}

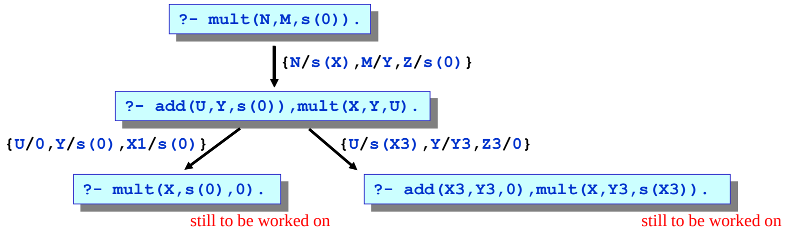
?- add(U,Y,s(0)),mult(X,Y,U).

still to be worked on

Detailed description of the generation of derivation trees

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

2. As long as there are still nodes to be worked on:
   - select left-most such node
   - determine all facts/rules (w. cons. renamed variables) whose head is unifiable with the left-most literal in that node
   - generate for each such fact/rule a (still to be worked on) successor node via a resolution step
   - arrange successor nodes from left to right according to the order of appearance of the used facts/rules in the program (from top to bottom)
   - annotate the unifier used in each case



?- mult(N,M,s(0)).

{N/s(X),M/Y,Z/s(0)}

?- add(U,Y,s(0)),mult(X,Y,U).

{U/0,Y/s(0),X1/s(0)}

{U/s(X3),Y/Y3,Z3/0}

?- mult(X,s(0),0).

still to be worked on

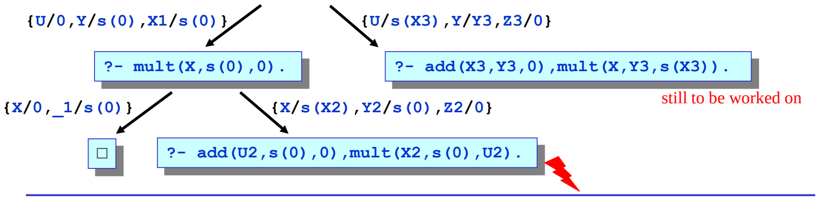
?- add(X3,Y3,0),mult(X,Y3,s(X3)).

still to be worked on

Detailed description of the generation of derivation trees

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

2. As long as there are still nodes to be worked on:
   - select left-most such node
   - determine all facts/rules (w. cons. renamed variables) whose head is unifiable with the left-most literal in that node
   - generate for each such fact/rule a (still to be worked on) successor node via a resolution step
   - arrange successor nodes from left to right according to the order of appearance of the used facts/rules in the program (from top to bottom)
   - annotate the unifier used in each case
   - mark nodes as finished if they are empty (□) or if their left-most literal is not unifiable with any fact/rule head (⚡)



{U/0,Y/s(0),X1/s(0)}

{U/s(X3),Y/Y3,Z3/0}

?- mult(X,s(0),0).

?- add(X3,Y3,0),mult(X,Y3,s(X3)).

still to be worked on

{X/0,_1/s(0)}

{X/s(X2),Y2/s(0),Z2/0}

□

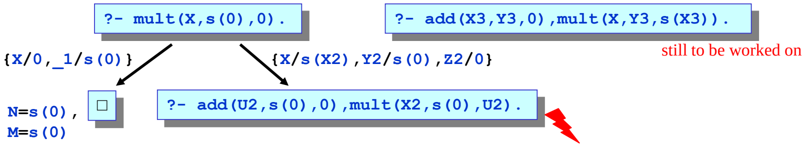
?- add(U2,s(0),0),mult(X2,s(0),U2).

⚡

Detailed description of the generation of derivation trees

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

2. As long as there are still nodes to be worked on:
   - select left-most such node
   - determine all facts/rules (w. cons. renamed variables) whose head is unifiable with the left-most literal in that node
   - generate for each such fact/rule a (still to be worked on) successor node via a resolution step
   - arrange successor nodes from left to right according to the order of appearance of the used facts/rules in the program (from top to bottom)
   - annotate the unifier used in each case
   - mark nodes as finished if they are empty or if their left-most literal is not unifiable with any fact/rule head
   - at successful nodes, annotate the solution (the composition of unifiers along the path from the root, applied to all variables that occurred in the original query)

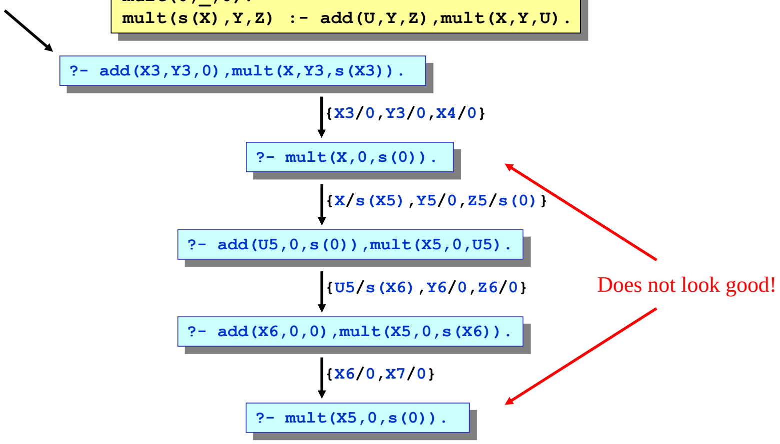


```
?- mult(X,s(0),0).
{X/0,_1/s(0)}
N=s(0),
M=s(0)
□
{X/s(X2),Y2/s(0),Z2/0}
?- add(U2,s(0),0),mult(X2,s(0),U2).

?- add(X3,Y3,0),mult(X,Y3,s(X3)).
```

still to be worked on

Back to the example: What to do?

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

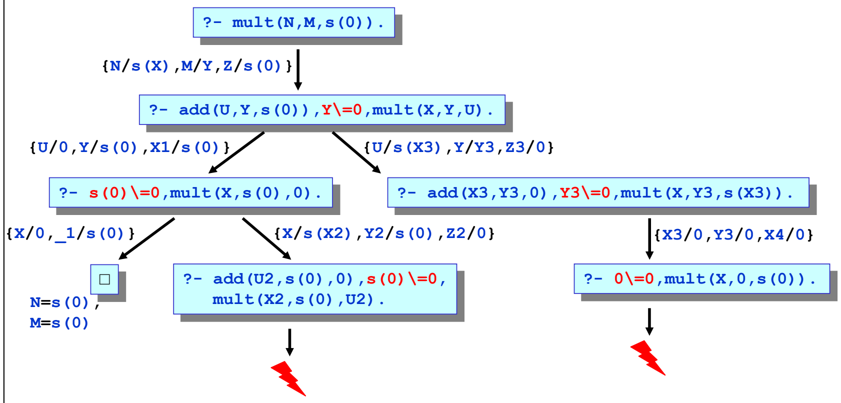


```
?- add(X3,Y3,0),mult(X,Y3,s(X3)).
{X3/0,Y3/0,X4/0}
?- mult(X,0,s(0)).
{X/s(X5),Y5/0,Z5/s(0)}
?- add(U5,0,s(0)),mult(X5,0,U5).
{U5/s(X6),Y6/0,Z6/0}
?- add(X6,0,0),mult(X5,0,s(X6)).
{X6/0,X7/0}
?- mult(X5,0,s(0)).
```

Does not look good!

Attempt: introducing an extra test

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),Y\=0,mult(X,Y,U).
```



```
?- mult(N,M,s(0)).
{N/s(X),M/Y,Z/s(0)}
?- add(U,Y,s(0)),Y\=0,mult(X,Y,U).
{U/0,Y/s(0),X1/s(0)}
?- s(0)\=0,mult(X,s(0),0).
{X/0,_1/s(0)}
□
N=s(0),
M=s(0)
{X/s(X2),Y2/s(0),Z2/0}
?- add(U2,s(0),0),s(0)\=0,
   mult(X2,s(0),U2).
{U/s(X3),Y/Y3,Z3/0}
?- add(X3,Y3,0),Y3\=0,mult(X,Y3,s(X3)).
{X3/0,Y3/0,X4/0}
?- 0\=0,mult(X,0,s(0)).
```

Only partial success

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),Y\=0,mult(X,Y,U).
```

```
?- mult(N,M,s(s(s(s(0))))).
N = s(0),
M = s(s(s(s(0)))) ;
N = s(s(0)),
M = s(s(0)) ;
N = s(s(s(s(0)))),
M = s(0) ;
false.
```

```
?- mult(s(0),0,0).
false.
```

New results found, old results lost!

Yet another “repair”

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(U,Y,Z),Y\=0,mult(X,Y,U).
```

Now this works:

```
?- mult(s(0),0,0).
true.
```

And it even works generally `mult(?X,?Y,+Z).`

But unfortunately (only noticed now):

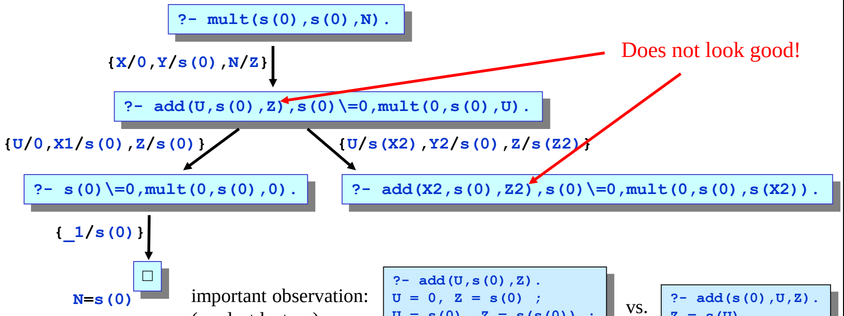
```
?- mult(s(0),s(0),N).
N = s(0) ;
abort
```

otherwise infinite search

So `mult(+X,+Y,?Z).` does not anymore work.

A new “infinity trap”

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(U,Y,Z),Y\=0,mult(X,Y,U).
```



`?- mult(s(0),s(0),N).`

`{X/0,Y/s(0),N/Z}`

`?- add(U,s(0),Z),s(0)\=0,mult(0,s(0),U).`

`{U/0,X1/s(0),Z/s(0)}`

`{U/s(X2),Y2/s(0),Z/s(Z2)}`

`?- s(0)\=0,mult(0,s(0),0).`

`?- add(X2,s(0),Z2),s(0)\=0,mult(0,s(0),s(X2)).`

`{_1/s(0)}`

□

`N=s(0)`

Does not look good!

important observation: (see last lecture)

```
?- add(U,s(0),Z).
U = 0, Z = s(0) ;
U = s(0), Z = s(s(0)) ;
...
```

vs.

```
?- add(s(0),U,Z).
Z = s(U).
```

Exploiting commutativity

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(Y,U,Z),Y\=0,mult(X,Y,U).
```

important observation:
(see last lecture)

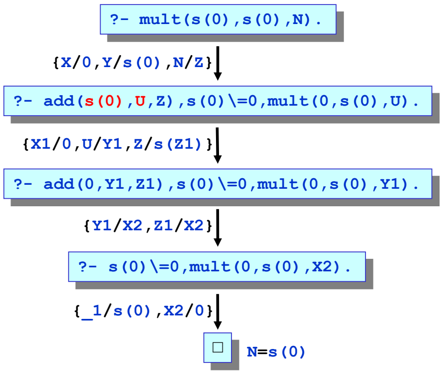
```
?- add(U,s(0),Z).
U = 0, Z = s(0) ;
U = s(0), Z = s(s(0)) ;
...
```

vs.

```
?- add(s(0),U,Z).
Z = s(U).
```

Exploiting commutativity

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(Y,U,Z),Y\=0,mult(X,Y,U).
```



```
?- mult(s(0),s(0),N).
```

{X/0,Y/s(0),N/Z}

```
?- add(s(0),U,Z),s(0)\=0,mult(0,s(0),U).
```

{X1/0,U/Y1,Z/s(Z1)}

```
?- add(0,Y1,Z1),s(0)\=0,mult(0,s(0),Y1).
```

{Y1/X2,Z1/X2}

```
?- s(0)\=0,mult(0,s(0),X2).
```

{_1/s(0),X2/0}

□ N=s(0)

Indeed a generally useful definition

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(Y,U,Z),Y\=0,mult(X,Y,U).
```

```
?- mult(N,M,s(s(s(0)))).
N = s(0),
M = s(s(s(0))) ;
N = s(s(0)),
M = s(s(0)) ;
N = s(s(s(0))),
M = s(0) ;
false.

?- mult(s(0),s(0),N).
N = s(0).

?- add(X,0,X),not(mult(s(s(_)),s(s(_)),X)).
...
```

Now <u>all</u> call modes work well, <u>except</u> `mult(?X,?Y,?Z)`!

Conclusion

The operational semantics:

- reflects the actual Prolog search process, with backtracking
- makes essential use of unification and resolution steps
- enables understanding of effects like non-termination
- gives insight into impact of changes to the order of, and within, facts and rules

---

Programming Paradigms

Negation in Prolog

Negation (1)

- Logic programming is primarily based on a positive logic.

  A literal is provable if it can be reduced (possibly via several resolution steps) to the validity of known facts.

- But Prolog also offers the possibility to use **negation**.
  - However, Prolog negation is not fully compatible with the expected logical meaning.
  - `\+ Goal`, or `not(Goal)`, is provable if and only if `Goal` is not provable.

Example: `\+ member(4,[2,3])` is provable, since `member(4,[2,3])` is not provable, i.e., it exists a "finite failure tree".

Caution:

```
?- member(X,[2,3]).          ⇒ X = 2; X = 3.
?- \+ member(X,[2,3]).       ⇒ false.
?- \+ \+ member(X,[2,3]).    ⇒ true.
```

(Negation does not yield variable bindings.)

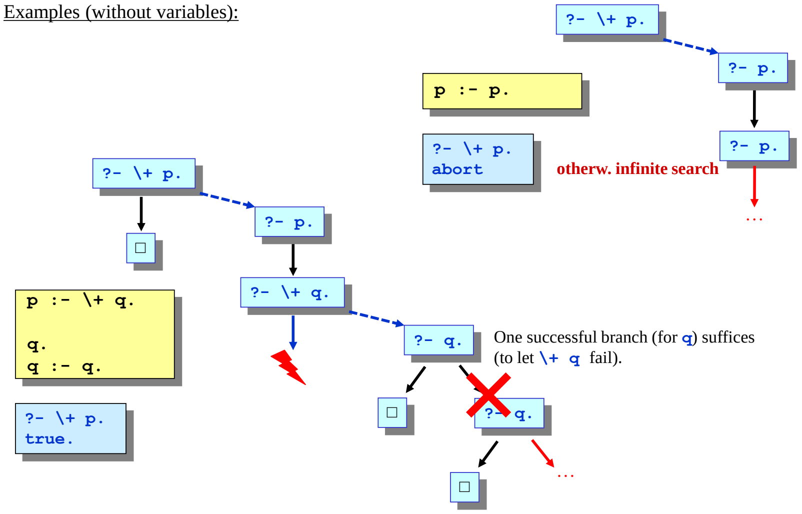
Negation (2)

- Why “finite failure tree”?
  - We cannot, in general, show that from the clauses of a program a certain negative statement follows.
  - We can only show that a certain positive statement can not be deduced. (negation as failure)
  - Here, “show” means to attempt a proof of the positive statement but to fail.
  - That any such attempt will necessarily fail (for some given positive statement) can only be said with certainty if the search space is finite.
- Underlying assumption:

closed world assumption

Negation (3)

Examples (without variables):



```
p :- p.
```

```
?- \+ p.
abort
```

otherw. infinite search

```
p :- \+ q.

q.
q :- q.
```

```
?- \+ p.
true.
```

One successful branch (for `q`) suffices (to let `\+ q` fail).

Negation (4)

Examples with variables:

```
human(marsellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X)  :- human(X), \+ married(X,Y).
```

```
?- single(X).
X = marsellus.

?- single(marsellus).
true.

?- single(vincent).
false.
```

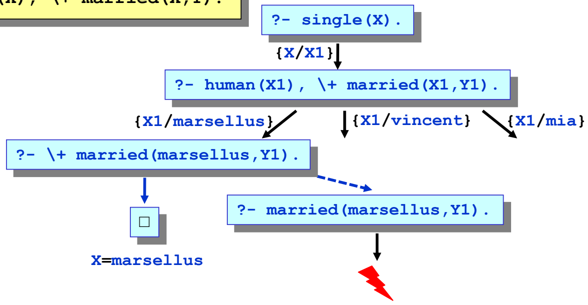
```
human(marsellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X)  :- \+ married(X,Y), human(X).
```

```
?- single(X).
false.

?- single(marsellus).
true.

?- single(vincent).
false.
```

Negation (5)

Examples with variables:

```
human(marsellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X)  :- human(X), \+ married(X,Y).
```



?- single(X).

{X/X1}

?- human(X1), \+ married(X1,Y1).

{X1/marsellus}

{X1/vincent}

{X1/mia}

?- \+ married(marsellus,Y1).

?- married(marsellus,Y1).

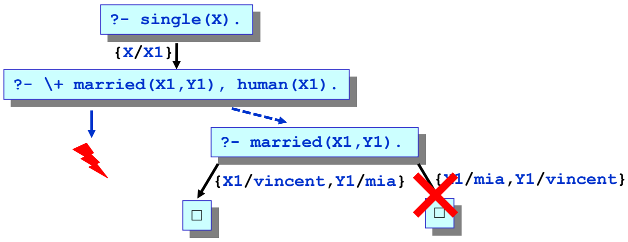
□

X=marsellus

Negation (6)

Examples with variables:

```
human(marsellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X)  :- \+ married(X,Y), human(X).
```



?- single(X).

{X/X1}

?- \+ married(X1,Y1), human(X1).

?- married(X1,Y1).

{X1/vincent,Y1/mia}

{X1/mia,Y1/vincent}

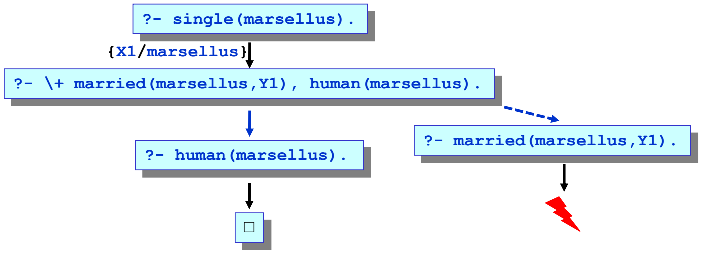
□

□

Negation (7)

Examples with variables:

```
human(marsellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X)  :- \+ married(X,Y), human(X).
```



?- single(marsellus).

{X1/marsellus}

?- \+ married(marsellus,Y1), human(marsellus).

?- human(marsellus).

?- married(marsellus,Y1).

□

Negation (8)

Explanation from "logical perspective" :

Under the assumptions that **`X`** is originally unbound and by **`human(X)`** will always be bound, this:

```
single(X) :- human(X), \+ married(X,Y).
```

means: $\forall$ X : human(X) $\wedge \neg(\exists$ Y : married(X,Y)) $\Rightarrow$ single(X).

But under the same assumptions, this:

```
single(X) :- \+ married(X,Y), human(X).
```

means: $\forall$ X : $\neg(\exists$ X,Y : married(X,Y)) $\wedge$ human(X) $\Rightarrow$ single(X).

Summary on Negation

- no real logical negation: instead, negation as failure
- proof search in "side branch", does not bind variables to the outside
- can only be truly understood procedurally/operationally
- problems with attempting a declarative perspective:
  - not compositional
  - sensitive against changes to the order of, and within, facts and rules
  - T_P-operator would be non-monotone

Potentielle Probleme mit Rekursion

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Alte Beispielaufgabe:

Given is an arbitrary database of facts about (true) lines between points in the plane, for example:

```
line(a, b). line(c, b). line(d, a).
line(b, d). line(d, c). line(d, e).
```

Implement predicates triangle with arity 3 and tetragon with arity 4, for the (true) triangles and tetragons created by the lines in the database. A line or triangle or tetragon is "true" if no two listed points are the same.

Also note that the line relation given above is not symmetric, even though lines between points should conceptually be considered to be so.

Lösungsversuch:

```
triangle(X,Y,Z) :- line(X,Y),line(Y,Z),line(Z,X).

tetragon(X,Y,Z,U) :- line(X,Y),line(Y,Z),line(Z,U),
                     line(U,X), X \= Z, Y \= U.
```

Potentielle Probleme mit Rekursion

Um die „fehlenden“ (per Symmetrie) Fakten wie „line (b,a)“ etc. zu berücksichtigen, liegt folgende Ergänzung nahe:

```
line(X,Y) :- line(Y,X).
```

Allerdings ist das leider „zu rekursiv“ (bei Ausführung).

Besser hier, Einführung eines gesonderten Prädikates und dann:

```
sline(X,Y) :- line(X,Y).
sline(Y,X) :- line(X,Y).

triangle(X,Y,Z) :- sline(X,Y), sline(Y,Z), sline(Z,X).

tetragon(X,Y,Z,U) :- sline(X,Y), sline(Y,Z), sline(Z,U),
                     sline(U,X), X \= Z, Y \= U.
```

Lösung bestand hier also im Verzicht auf Rekursion.

Potentielle Probleme mit Rekursion

```
direct(frankfurt,san_francisco).
direct(frankfurt,chicago).
direct(san_francisco,honolulu).
direct(honolulu,maui).

connection(X, Y) :- direct(X, Y).
connection(X, Y) :- direct(X, Z), connection(Z, Y).
```

```
?- connection(frankfurt,maui).
true.

?- connection(san_francisco,X).
X = honolulu ;
X = maui ;
false.

?- connection(maui,X).
false.
```

Potentielle Probleme mit Rekursion

```
direct(frankfurt,san_francisco).
direct(frankfurt,chicago).
direct(san_francisco,honolulu).
direct(honolulu,maui).

connection(X, Y) :- connection(X, Z), direct(Z, Y).
connection(X, Y) :- direct(X, Y).
```

```
?- connection(frankfurt,maui).
ERROR: Out of local stack
```

Potentielle Probleme mit Rekursion

```
direct(frankfurt,san_francisco).
direct(frankfurt,chicago).
direct(san_francisco,honolulu).
direct(honolulu,maui).
direct(honolulu,san_francisco).

connection(X, Y) :- direct(X, Y).
connection(X, Y) :- direct(X, Z), connection(Z, Y).
```

```
?- connection(san_francisco,Y).
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
Y = honolulu ;
Y = maui ; …
```

Ziel sollte sein: Endlossuche vermeiden

Potentielle Probleme mit Rekursion

Idee: schon bereiste Zwischenstationen merken, zum Beispiel als Liste:

```
direct(frankfurt,san_francisco).
…
direct(honolulu,san_francisco).

connection(X, Y) :- connection1(X, Y, [X]).

connection1(X, Y, _) :- direct(X, Y).
connection1(X, Y, L) :- direct(X, Z), not(member(Z,L)),
                        append([Z], L, L1),
                        connection1(Z, Y, L1).
```

```
?- connection(san_francisco,Y).
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
false.
```

Spezielle Datenstruktur: Listen

Neben Konstanten und per Schachtelung von Datenkonstruktoren wie `s/1` und `z/0` zu erhaltenden Datenstrukturen, wurden auch Listen mit Syntax wie `[1,2,3,4,5]` und `[duisburg,X,essen]` zuvor bereits kurz erwähnt.

Zur Arbeit mit Listen hält Prolog diverse Prädikate bereit, zum Beispiel:

- `member/2`, um auszudrücken, dass ein Element in einer Liste vorkommt
- `append/3`, um auszudrücken, dass eine Liste die Aneinanderhängung zweier bestimmter Listen ist
- `length/2`, um auszudrücken, welche Länge eine Liste hat

Vordefinierte Prädikate auf Listen

Interessant dabei ist, dass (ganz im Sinne „unseres“ add/3-Prädikates) diverse Aufrufmodi der Listenprädikate funktionieren. Zum Beisiel:

```
?- member(3,[1,2,3,4,5]).
true.

?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3.

?- member(3,[X,Y,Z]).
X = 3 ;
Y = 3 ;
Z = 3.
```

Vordefinierte Prädikate auf Listen

Auch für die anderen Listenprädikate, zum Beispiel:

```
?- append([1,2,3],[4,5],L).
L = [1,2,3,4,5].

?- append(X,Y,[a,b]).
X = [], Y = [a,b] ;
X = [a], Y = [b] ;
X = [a,b], Y = [].

?- append(X,X,[a,b]).
false.

?- append(X,X,[a,Y]).
X = [a], Y = a.
```

Vordefinierte Prädikate auf Listen

Oder:

```
?- length([a,b,c],N).
N = 3.

?- length(L,3).
L = [_1570, _1576, _1582].

?- length(L,3),append(X,X,L).
false.

?- length(L,4),append(X,X,L).
L = [_1610, _1616, _1610, _1616], X = [_1610, _1616].

?- length(L,2),member(a,L),member(b,L),member(c,L).
false.
```

Definition von Prädikaten auf Listen

Definiert werden Prädikate auf Listen typischerweise durch Verwendung bereits vorhandener:

```
insert(X,L,R) :- append(U,V,L),append(U,[X],Y),
                 append(Y,V,R).
```

und/oder Rekursion:

```
permutation([],[]).
permutation(L,P) :- append([X],Y,L),permutation(Y,Z),
                    insert(X,Z,P).
```

Mit obigen Definitionen, zum Beispiel:

```
?- permutation([1,2,3],L)
L = [1,2,3] ;
L = [2,1,3] ;
...
```

Generate and Test – Was und Wie

Angenommen, wir möchten Listen sortieren können, also zum Beispiel Anfragen wie

```
?- sortingTo([4,2,6,9,1],R).
```

stellen können, und darauf als Antwort

```
R = [1,2,4,6,9].
```

erhalten.

Also **was** wollen wir genau?

Generate and Test – Was und Wie

Die gewünschte Eigenschaft der Ergebnisliste können wir relativ leicht als Prolog-Prädikat ausdrücken:

```
isSorted([]).
isSorted([_]).
isSorted(Xs) :- append([X,Y],Ys,Xs), X =< Y,
                append([Y],Ys,Zs),isSorted(Zs).
```

Dann gilt zum Beispiel:

```
?- isSorted([4,2,6,9,1]).
false.
?- isSorted([1,2,4,6,9]).
true.
```

Und **wie** können wir eine solche passende Liste erhalten bzw. herstellen?

Generate and Test – Was und Wie

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Nun, eine recht naive, aber funktionierende Lösung wäre:

```
sortingTo(Xs,Ys) :- permutation(Xs,Ys),isSorted(Ys).
```

Dann in der Tat:

```
?- sortingTo([4,2,6,9,1],R).
R = [1,2,4,6,9].
```

Prinzip hier:

Um eine Regel auf Eingaben zu formulieren, die genau dann **true** liefert, wenn eine gültige Lösung des Problems vorliegt, Zerlegung in zwei Teile:

- **Generate**-Teil definiert einen Suchraum.
- **Test**-Teil definiert die Bedingung, die erfüllt sein muss.

Generate and Test – Weitere Beispiele

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Aufgabe: Ermittle alle Möglichkeiten, bei dreimaligem Würfeln insgesamt 15 Punkte zu erzielen.

Lösung:

```
?- W = [1,2,3,4,5,6], member(A,W), member(B,W),
   member(C,W), A + B + C =:= 15.
```

Generate and Test – Weitere Beispiele

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Aufgabe: Ermittle alle Möglichkeiten, bei dreimaligem Würfeln mit verschiedenen Augenzahlen insgesamt 15 Punkte zu erzielen.

Lösung:

```
?- W = [1,2,3,4,5,6], member(A,W), member(B,W),
   member(C,W), A \= B, A \= C, B \= C,
   A + B + C =:= 15.
```

oder:

```
?- permutation([1,2,3,4,5,6],[A,B,C,_,_,_]),
   A + B + C =:= 15.
```

Generate and Test – Weitere Beispiele

Aufgabe: Ermittle alle Möglichkeiten, bei dreimaligem Würfeln mit verschiedenen Augenzahlen in aufsteigender Reihenfolge insgesamt 15 Punkte zu erzielen.

Lösung:

```
?- permutation([1,2,3,4,5,6],[A,B,C,_,_,_]),
   isSorted([A,B,C]), A + B + C =:= 15.
```

Generate-and-Test ist sinnvoll einzusetzen bei nicht-trivialen kombinatorischen Problemen, wenn

- die Menge der potentiellen Lösungen endlich oder besser sogar recht klein ist, oder
- man keine Vorstellung darüber hat, wie systematisch schneller eine Lösung gefunden werden könnte.

Beispiel: Krypto-Arithmetik

ABB	**–**	**CD**	**=**	**EED**
–		**–**		*****
FD	**+**	**EF**	**=**	**CE**
=		**=**		**=**
EGD	*****	**FH**	**=**	**?**

Jeder Buchstabe entspreche einer anderen Ziffer.

Wie lautet eine gültige Belegung?

Beispiel: Krypto-Arithmetik

```
solve(A,B,C,D,E,F,G,H) :- generate(A,B,C,D,E,F,G,H),
                          test(A,B,C,D,E,F,G,H).

generate(A,B,C,D,E,F,G,H) :-
    permutation([0,1,2,3,4,5,6,7,8,9],
                [A,B,C,D,E,F,G,H,_,_]).

test(A,B,C,D,E,F,G,H) :- ???
```

Zum Beispiel die erste Zeile entspricht:

```
(A * 100 + B * 10 + B) - (C * 10 + D)
    =:= E * 100 + E * 10 + D
```

Und die erste Spalte:

```
(A * 100 + B * 10 + B) - (F * 10 + D)
    =:= E * 100 + G * 10 + D
```

Beispiel: Krypto-Arithmetik

Zweite Zeile und zweite Spalte:

```
(F * 10 + D) + (E * 10 + F) =:= C * 10 + E,
(C * 10 + D) - (E * 10 + F) =:= F * 10 + H
```

Schließlich noch die Bedingung, dass gleiches Ergebnis in letzter Zeile und letzter Spalte:

```
(E * 100 + E * 10 + D) * (C * 10 + E)
    =:= (E * 100 + G * 10 + D) * (F * 10 + H)
```

Beispiel: Krypto-Arithmetik

Insgesamt für den Test-Teil:

```
test(A,B,C,D,E,F,G,H) :-
    (A * 100 + B * 10 + B) - (C * 10 + D)
        =:= E * 100 + E * 10 + D,
    (A * 100 + B * 10 + B) - (F * 10 + D)
        =:= E * 100 + G * 10 + D,
    (F * 10 + D) + (E * 10 + F) =:= C * 10 + E,
    (C * 10 + D) - (E * 10 + F) =:= F * 10 + H,
    (E * 100 + E * 10 + D) * (C * 10 + E)
        =:= (E * 100 + G * 10 + D) * (F * 10 + H).
```

Als eindeutige erfüllende Belegung findet Prolog mit der Anfrage

```
?- solve(A,B,C,D,E,F,G,H).
```

dies: A = 2, B = 0, C = 8, D = 5, E = 1, F = 6, G = 3, H = 9.

Beispiel: Krypto-Arithmetik

200	**−**	**85**	**=**	**115**
–		**–**		*****
65	**+**	**16**	**=**	**81**
=		**=**		**=**
135	*****	**69**	**=**	**9315**

Beispiel: Einstein’s Riddle

Zur Erinnerung:

1. The Englishman lives in the red house.
2. The Spaniard owns the dog.
3. Coffee is drunk in the green house.
4. The Ukrainian drinks tea.
5. The green house is immediately to the right of the ivory house.
6. The Winston smoker owns snails.
7. Kools are smoked in the yellow house.
8. Milk is drunk in the middle house.
9. The Norwegian lives in the leftmost house.
10. The man who smokes Chesterfield lives in the house next to the man with the fox.
11. Kools are smoked in the house next to the house where the horse is kept.
12. The Lucky Strike smoker drinks orange juice.
13. The Japanese smokes Parliaments.
14. The Norwegian lives next to the blue house.

Beispiel: Einstein’s Riddle

Versuchen wir, das Rätsel per Generate-and-Test zu lösen.

Für den Generate-Teil wäre zunächst einfach denkbar:

```
Houses = [ _, _, _, _, _ ]
```

Oder auch bereits:

```
Houses = [ [ _, _, _, _, _ ]
         , [ _, _, _, _, _ ]
         , [ _, _, _, _, _ ]
         , [ _, _, _, _, _ ]
         , [ _, _, _, _, _ ] ]
```

Beispiel: Einstein’s Riddle

Für den Test-Teil nehmen wir uns die einzelnen Hinweise vor.

Zum Beispiel:

1. The Englishman lives in the red house.

Unter der Festlegung, dass wir die einzelnen Attribute jeweils in der Reihenfolge „color“, „nationality“, „drink“, „pet“, „smoke“ angeben werden, können wir diesen ersten Hinweis wie folgt ausdrücken:

```
member([ red, englishman, _, _, _ ], Houses)
```

Analog:

2. The Spaniard owns the dog.

wird zu:

```
member([ _, spaniard, _, dog, _ ], Houses)
```

Beispiel: Einstein's Riddle

Die nächsten beiden Hinweise:

3. Coffee is drunk in the green house.
4. The Ukrainian drinks tea.

werden auch analog behandelt:

```
member([ green, _, coffee, _, _ ], Houses)
```

bzw.:

```
member([ _, ukrainian, tea, _, _ ], Houses)
```

Beispiel: Einstein's Riddle

Dann wird es wieder etwas interessanter:

5. The green house is immediately to the right of the ivory house.

Das könnten wir so ausdrücken:

```
rightOf([ green, _, _, _, _ ],
        [ ivory, _, _, _, _ ],
        Houses)
```

wenn wir ein solches Prädikat hätten.

Definieren wir es uns doch einfach:

```
rightOf(R,L,List) :- append(Prefix,_,List),
                     append(_,[L,R],Prefix).
```

Beispiel: Einstein's Riddle

Nun kommen nochmal zwei sehr einfach umzusetzende Hinweise:

6. The Winston smoker owns snails.
7. Kools are smoked in the yellow house.

Dann wieder spannender:

8. Milk is drunk in the middle house.
9. The Norwegian lives in the leftmost house.

Diese beiden können wir umsetzen, indem wir

```
Houses = ...
```

verfeinern zu:

```
Houses = [ [ _, norwegian, _, _, _ ], _
         , [ _, _, milk, _, _ ], _, _ ]
```

Beispiel: Einstein's Riddle

Für den nächsten Hinweis:

10. The man who smokes Chesterfield lives in the house next to the man with the fox.

brauchen wir nochmal ein Hilfsprädikat:

```
nextTo([ _, _, _, _, chesterfield ],
       [ _, _, _, fox, _ ],
       Houses)
```

welches wir wie folgt definieren können:

```
nextTo(X,Y,List) :- rightOf(X,Y,List).
nextTo(X,Y,List) :- rightOf(Y,X,List).
```

Beispiel: Einstein's Riddle

Die restlichen Hinweise:

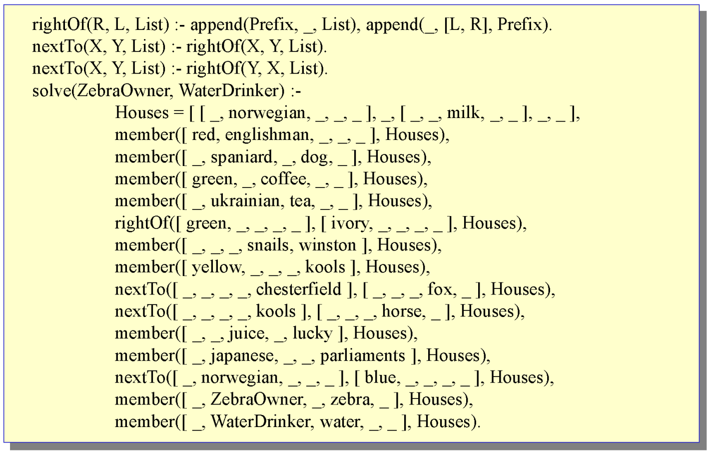
11. Kools are smoked in the house next to the house where the horse is kept.
12. The Lucky Strike smoker drinks orange juice.
13. The Japanese smokes Parliaments.
14. The Norwegian lives next to the blue house.

lassen sich dann alle analog zu schon vertrauten umsetzen.

Es bleibt noch, letztlich den Zebra-Besitzer und den Wasser-Trinker zu bestimmen.

Dazu können Variablen und weitere member-Aufrufe verwendet werden.

Beispiel: Einstein's Riddle – Gesamtlösung



```
rightOf(R, L, List) :- append(Prefix, _, List), append(_, [L, R], Prefix).
nextTo(X, Y, List) :- rightOf(X, Y, List).
nextTo(X, Y, List) :- rightOf(Y, X, List).
solve(ZebraOwner, WaterDrinker) :-
            Houses = [ [ _, norwegian, _, _, _ ], _, [ _, _, milk, _, _ ], _, _ ],
            member([ red, englishman, _, _, _ ], Houses),
            member([ _, spaniard, _, dog, _ ], Houses),
            member([ green, _, coffee, _, _ ], Houses),
            member([ _, ukrainian, tea, _, _ ], Houses),
            rightOf([ green, _, _, _, _ ], [ ivory, _, _, _, _ ], Houses),
            member([ _, _, _, snails, winston ], Houses),
            member([ yellow, _, _, _, kools ], Houses),
            nextTo([ _, _, _, _, chesterfield ], [ _, _, _, fox, _ ], Houses),
            nextTo([ _, _, _, _, kools ], [ _, _, _, horse, _ ], Houses),
            member([ _, _, juice, _, lucky ], Houses),
            member([ _, japanese, _, _, parliaments ], Houses),
            nextTo([ _, norwegian, _, _, _ ], [ blue, _, _, _, _ ], Houses),
            member([ _, ZebraOwner, _, zebra, _ ], Houses),
            member([ _, WaterDrinker, water, _, _ ], Houses).
```