Script:First steps in Haskell

Headline

First steps of programming in Haskell

Summary

Functional programming in Language:Haskell is introduced in this lecture. First, basic programming concepts (such as program and algorithm) are briefly rehashed. Then, key concepts of functional programming (such as function, recursion, list, and pattern matching) are discussed. Simple program examples are used for illustrations (such as the factorial function and linear search). Eventually, a major running example of this course is introduced: the so-called 101system, which is essentially a trivial information system for human resource management.

Video

Some version of the lecture has been recorded.

Introduction to functional programming

Concepts

These are all the concepts that play a role in this lecture.

Basic programming concepts

- Software system
- Requirement
- Program
- Programming language
- Compiler
- Interpreter
- Algorithm
- Algorithmic problem
- Data structure

Key functional programming concepts

- Functional programming
- Function
- Function definition
- Function application
- Function composition
- Immutable list
- Pattern matching

Simple program samples

- Factorial
- Greatest common divisor
- Linear search

Features of the 101system

- Feature:Flat company
- Feature:Total
- Feature:Cut

Languages

These are all the software languages that play a role in this lecture.

- Language:Haskell

Technologies

These are all the software technologies that play a role in this lecture.

- Technology:Haskell platform
- Technology:GHC
- Technology:GHCi

Contributions

These are the implementations of the 101system that are discussed in this lecture.

- Contribution:haskellStarter

Information

These are pointers to some general pieces of information as to how this wiki works.

- Information:Download
- Information:Wikipedia
- Information:Translate
- Information:Contact

Metadata

This lecture is part of the functional programming course in Koblenz.

- Course:Lambdas in Koblenz

101system

Headline

An imaginary HRMS system in the 101project

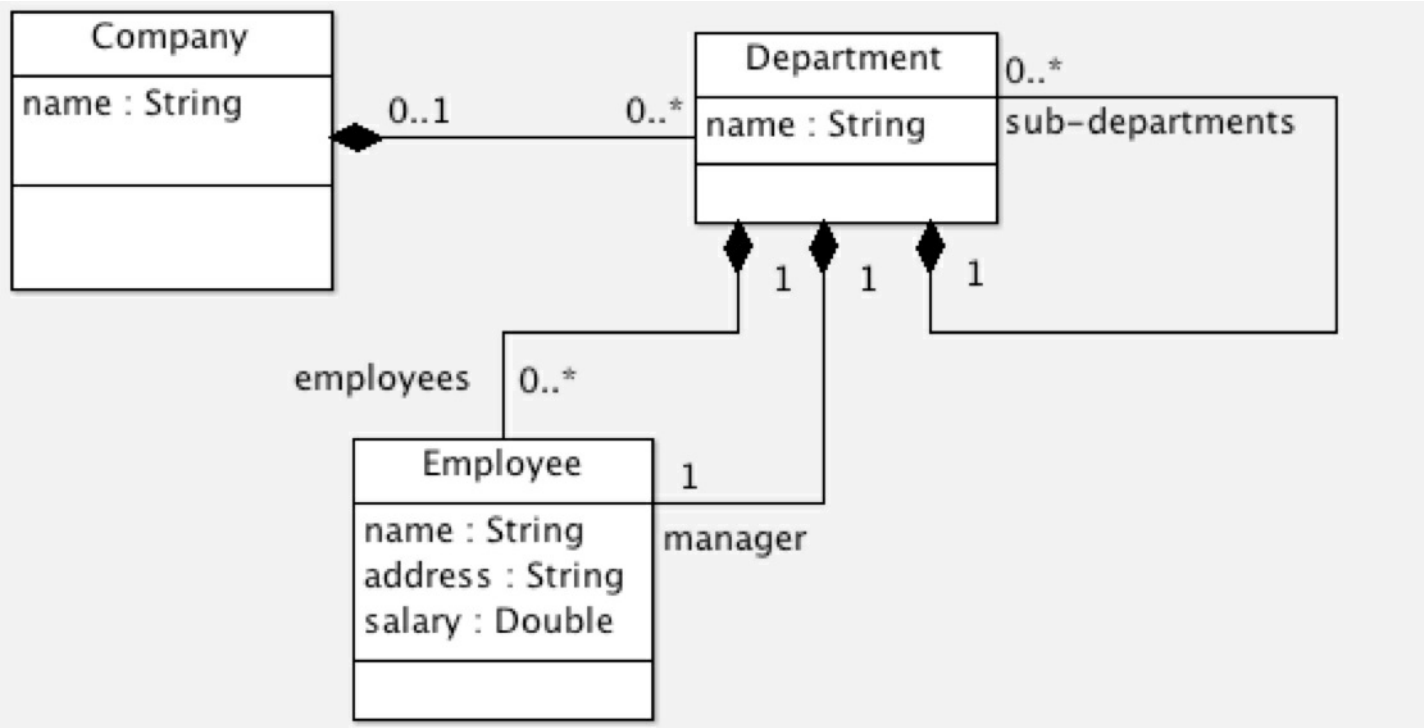
Description

The 101system (or just "the system") is an imaginary *Human resource management system (HRMS)* which serves as the "running example" in the 101project. That is, "contributions" to the project are meant to implement or model or otherwise exercise the system for a conceived *company* as a client.

The system is supposed to model the company structure in terms of employees and possibly the hierarchical structure of departments. Employees are modeled in terms of their names, addresses, salaries, and possibly additional properties. The system is supposed to meet certain functional requirements such as totaling all salaries in the company. The system may also be subjected to non-functional requirements such as persistence or distribution. All requirements are organized in Namespace:Feature. The features are not collected for the sake of an interesting HRMS system. Instead, the features are designed to exercise interesting characteristics of software languages and software technologies. Most features are optional so that contributions have the freedom of choice to focus on features that are particularly interesting for a certain objective of language or technology demonstration.

Illustration

The following UML class diagram models the basic structure of the 101system.



See Theme:Starter for a few very simple contributions in varying languages. These are mostly implementations of the system in varying programming languages, but a UML-based model (as shown above) is also included.

Metadata

- Human resource management system
- Namespace:Feature
- Namespace:101

Algorithm

Headline

A step-by-step procedure for achieving a certain IO behavior

Illustration

See greatest common divisor and linear search for examples of an algorithm which is described both semi-formally and in an actual programming language.

Citation

(http://en.wikipedia.org/wiki/Algorithm, 14 April 2013)

In mathematics and computer science, an algorithm ... is a step-by-step procedure for calculations. Algorithms are used for calculation, data processing, and automated reasoning.

More precisely, an algorithm is an effective method expressed as a finite list ... of well-defined instructions ... for calculating a function. ... Starting from an initial state and initial input (perhaps empty), ... the instructions describe a computation that, when executed, proceeds through a finite ... number of well-defined successive states, eventually producing "output" ... and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input.

Metadata

- http://en.wikipedia.org/wiki/Algorithm
- Vocabulary:Programming
- Program
- Algorithmic problem
- Namespace:Concept

Algorithmic problem

Headline

A problem that can be solved with an algorithm

Illustration

- Examples of algorithmic problems: greatest common divisor, the factorial, and the search problem.
- An example of non-algorithmic problems: the Halting problem.

Metadata

- Vocabulary:Programming
- http://en.wikipedia.org/wiki/Algorithm
- Concept

Function

Headline

The central abstraction in functional programming

Description

In programming, a function is an abstraction for computation. In functional programming, a function is typically close to what we call a function in mathematics. That is, a function maps values of its domain to values of its range. Outside (pure) functional programming, a function may also involve additional computational aspects or side effects. Functions are defined (or declared) in a program and then elsewhere applied.

Illustration

Consider the following function definition in Language:Haskell.

```
-- The increment function
inc :: Int -> Int
inc x = x + 1
```

The definition consists of a type signature assigning a type to the function *inc* and one equation providing the actual definition. The type signature declares *inc* as function from *Int* to *Int*. The equation binds the variable *x* on the left-hand side and returns the expression *x+1* on the right-hand side.

Consider the following function application: *inc* is applied to 41; the result is 42.

```
> inc 41
42
```

Metadata

- Abstraction mechanism
- Vocabulary:Functional programming
- http://en.wikipedia.org/wiki/Function_(mathematics)

Function composition

Headline

Apply two functions, one after another

Description

Function composition is a fundamental operation on functions so that they are applied one after another. That is the composition of *f* and *g* when applied to an argument *x* is the same as first computing the result *y* of applying *g* to *x* and then applying *f* to *y*.

Illustration

Function composition in Haskell

Let's assume the following function:

```
-- Test an Int to be even
even :: Int -> Bool
even x = x `mod` 2 == 0
```

The test for a number to be *odd* can be trivially described by composing *even* and logical negation. In Language:Haskell, function composition is denoted by the infix operator ".". Thus:

```
-- Test an Int to be odd
odd :: Int -> Bool
odd = not . even
```

We can always replace function composition by some pattern of function application. For instance, *odd* could also be expressed like this:

```
-- Test an Int to be odd
odd' :: Int -> Bool
odd' x = not (even x)
```

Metadata

- Vocabulary:Functional programming
- http://en.wikipedia.org/wiki/Function_composition
- http://en.wikipedia.org/wiki/Function_composition_(computer_science)

Function definition

Headline

The abstraction mechanism for functions

Description

See the concept of function.

Metadata

- Vocabulary:Functional programming

Greatest common divisor

Headline

The greatest common divisor of two integers

Illustration

We face an algorithmic problem because the greatest common divisor can be computed by a (simple) algorithm.

- We assume (for simplicity) two positive operands *x* and *y*.
- Perform the following steps to return the greatest common divisor of *x* and *y*.
    - Repeat the following step until *x* equals *y*:
        - If *x* > *y*
            - then subtract *y* from *x* and
            - else subtract *x* from *y*.
    - Return *x*.

Here is also an implementation in Language:Java:

```
// Compute greatest common divisor
public static int gcd(int x, int y) {
  // This version requires positive integers.
  assert x > 0 && y > 0;
  while (x != y) {
    if (x > y)
      x = x - y;
    else
      y = y - x;
  }
  return x;
}
```

Here is also an implementation in Language:Haskell:

```
-- Operands are supposed to be positive integers.
gcd :: Int -> Int -> Int
gcd x y | x > y = gcd (x-y) y
        | x < y = gcd x (y-x)
        | otherwise = x
```

Citation

(http://en.wikipedia.org/wiki/Greatest_common_divisor, 15 April 2013)

In mathematics, the greatest common divisor (gcd), also known as the greatest common factor (gcf), or highest common factor (hcf), of two or more integers (at least one of which is not zero), is the largest positive integer that divides the numbers without a remainder. For example, the GCD of 8 and 12 is 4.

Metadata

- http://en.wikipedia.org/wiki/Greatest_common_divisor
- Algorithmic problem

Interpreter

Headline

A program that executes programs in a software language

Illustration

Let's assume that we want to interpret the Language:Python variation on the Hello world program.

That is, the source code of the program is available as a file "HelloWorld.py".

We simply invoke the Python interpreter at the command line:

```
$ python HelloWorld.py
Hello, world!
```

This is different from compilation, where we would first use a compiler to produce object code that is eventually also interpreted. In fact, Python scripts could also be compiled in the interest of performance. The use of the interpreter may be more convenient in scripting.

Metadata

- Vocabulary:Programming
- http://en.wikipedia.org/wiki/Interpreter_(computing)
- Compiler

List

Headline

An ordered collection of values

Illustration

Lists can be represented in different ways and different sets of operations can be provided. One fundamental choice to be made is whether lists are considered mutable or immutable; see the illustration of linked and immutable lists.

Metadata

- http://en.wikipedia.org/wiki/List_(abstract_data_type)
- Data structure
- Vocabulary:Data
- List type
- Record
- Tuple

Pattern matching

Headline

Matching values against patterns to bind variables

Description

Pattern matching may be concerned with different kinds of types, e.g.,text or trees. In the case of text,regular expressions provide the foundation for patterns. In the case of trees and specifically in the context of functional programming, algebraic data types provide the foundation for patterns; in this case, pattern matching is concerned with case discrimination on different constructor patterns such that variables are bound in successfully matched patterns for use in expressions.

Illustration

Pattern matching in Haskell

The basics of Haskell's pattern matching are very similar to those of other functional programming languages.

Pattern matching on pairs

```
-- Project a pair to first component
fst :: (a,b) -> a
fst (x,_) = x

-- Project a pair to second component
snd :: (a,b) -> b
snd (_,x) = x
```

These two functions *fst* and *snd* are defined like this (or similarly) in thePrelude module of Language:Haskell. They are defined by pattern matching on the structure of tuples; see the the left-hand sides of the function definitions. The idea of such pattern matching is of course that variables in the pattern (on the left-hand side) can be used in the expression of the definition (on the right-hand side).

Pattern matching on lists

```
-- Retrieve head (first element) of a list
head :: [a] -> a
head (x:_) = x

-- Retrieve tail (all but first element) of a list
tail :: [a] -> [a]
tail (_:xs) = xs
```

These two functions *head* and *tail* are defined like this (or similarly) in thePrelude module of Language:Haskell. They demonstrate that non-empty lists are constructed with the cons constructor ":" from a head and a tail.

Pattern matching is particularly convenient, when functions should be defined by case discrimination on the different constructor patterns for a data type. Consider, for example, the length function (again borrowed from the Prelude); this definition consists of two equations: one for the case of an empty list and another case for non-empty lists:

```
-- Determine length of list
length :: [a] -> Int
length [] = 0
length (_:xs) = length xs + 1
```

Other forms of pattern matching

- Pattern matching is particularly useful for user-defined algebraic data types.
- Pattern matching is not limited to the use on left-hand sides of equations. Instead, pattern matching can also be performed throughcase expressions in an expression context.
- Haskell patterns may involve so-called guards to control the selection of equations (cases) not just purely on the grounds structure but also computations on top of bound variables.
- Haskell provides different forms of patterns to deal withlaziness. This is not further discussed here.

Metadata

- Vocabulary:Functional programming
- Vocabulary:Data
- http://en.wikipedia.org/wiki/Pattern_matching
- http://en.wikibooks.org/wiki/Haskell/Pattern_matching

Recursion

Headline

The use of self-reference in defining abstractions

Illustration

Clearly, there are different forms of recursions, as they are different abstraction mechanisms that permit recursion. For instance, in functional programming, both Functions and data types may be defined recursively.

Recursive functions

Consider the following recursive formulation of the factorial function in Language:Haskell:

```
-- A recursive definition of the factorial function
factorial n =
  if n==0
    then 1
    else n * factorial (n-1)
```

This is essentially a form of primitive recursion: the function definition checks for the argument *n* to be "0" for the base case and applies the function recursively to the predecessor of the argument otherwise. For the record, non-recursive formulations are feasible, too, depending on the helper functions we are willing to use. For instance, we may use the ".." operator to enumerate all values in a range and then apply the *product* function:

```
-- A non-recursive definition of the factorial function
factorial' n = product [1..n]
```

Recursive data types

Under construction.

Metadata

- Vocabulary:Programming theory
- Vocabulary:Programming
- http://en.wikipedia.org/wiki/Recursion_(computer_science)
- http://mathworld.wolfram.com/Recursion.html
- Concept

Requirement

Headline

A requirement for a new or altered software system or component

Illustration

Consider the following functional requirement for the @system: "The system must be able to total the salaries of all employees of the company and to report the total to the user." Feature:Total describes this requirement in more detail.

Details

Requirements can be classified in different way. The following classifiers are used on the 101wiki specifically also for the classification of the requirements of the @system:

- Functional requirement
- Non-functional requirement
- Data requirement
- UI requirement

Requirements may invoke different abstraction levels. On the 101wiki, requirements may refer to the abstraction levels of software architecture, software design, and data modeling as opposed to any narrow focus on requirements analysis.

Metadata

- http://en.wikipedia.org/wiki/Requirement
- http://en.wikipedia.org/wiki/Requirements analysis
- http://en.wikipedia.org/wiki/Software requirements specification
- Vocabulary:Software engineering
- Software feature

Contribution:haskellStarter

Headline

Basic functional programming in Language:Haskell.

Characteristics

The contribution demonstrates basic style of functional programming in Language:Haskell. Only very basic language constructs are exercised. Companies are represented via tuples over primitive data types. (No algebraic data types are used; type synonyms suffice.) Only flat companies are modeled, i.e., nested departments are not modeled. Pure, recursive functions implement operations for totaling and cutting salaries by pattern matching. The types for companies readily implement read and show functions for closed serialization.

Illustration

101companies contribution haskellStarter

The data model relies on tuples for data composition:

```
-- | Companies as pairs of company name and employee list
type Company = (Name, [Employee])

-- | An employee consists of name, address, and salary
type Employee = (Name, Address, Salary)
```

Basic types for strings and floats are leveraged for names, addresses, and salaries.

```
-- | Addresses as strings
type Address = String

-- | Total all salaries in a company
total :: Company -> Float
total = sum . salaries

-- | Salaries as floats
type Salary = Float
```

A sample company looks like this:

```
-- | A sample company useful for basic tests
sampleCompany :: Company
sampleCompany =
  ( "Acme Corporation",
    [
      ("Craig", "Redmond", 123456),
      ("Erik", "Utrecht", 12345),
      ("Ralf", "Koblenz", 1234),
      ("Ray", "Redmond", 234567),
      ("Klaus", "Boston", 23456),
      ("Karl", "Riga", 2345),
      ("Joe", "Wifi City", 2344)
    ]
  )
```

Features for functional requirements are implemented by families of functions on the company types. For instance, Feature:Total is implemented as follows:

```
-- | Total all salaries in a company
total :: Company -> Float
total = sum . salaries

-- | Extract all salaries in a company
salaries :: Company -> [Salary]
salaries (n, es) = getSalaries es

-- | Extract all salaries of lists of employees
getSalaries :: [Employee] -> [Salary]
getSalaries [] = []
getSalaries (e:es) = getSalary e : getSalaries es

-- | Extract the salary from an employee
getSalary :: Employee -> Salary
getSalary (_, _, s) = s
```

We may test these functions with the following function application:

```
total sampleCompany
```

The function application evaluates to the following total:

```
399747.0
```

All the remaining functions are implemented in the same module:

```
-- | Companies as pairs of company name and employee list
type Company = (Name, [Employee])

-- | An employee consists of name, address, and salary
type Employee = (Name, Address, Salary)

-- | Names as strings
```

```
type Name = String

-- | Addresses as strings
type Address = String

-- | Salaries as floats
type Salary = Float

-- | A sample company useful for basic tests
sampleCompany :: Company
sampleCompany =
  ( "Acme Corporation",
    [
      ("Craig", "Redmond", 123456),
      ("Erik", "Utrecht", 12345),
      ("Ralf", "Koblenz", 1234),
      ("Ray", "Redmond", 234567),
      ("Klaus", "Boston", 23456),
      ("Karl", "Riga", 2345),
      ("Joe", "Wifi City", 2344)
    ]
  )

-- | Total all salaries in a company
total :: Company -> Float
total = sum . salaries

-- | Extract all salaries in a company
salaries :: Company -> [Salary]
salaries (n, es) = getSalaries es

-- | Extract all salaries of lists of employees
getSalaries :: [Employee] -> [Salary]
getSalaries [] = []
getSalaries (e:es) = getSalary e : getSalaries es

-- | Extract the salary from an employee
getSalary :: Employee -> Salary
getSalary (_, _, s) = s

-- | Cut all salaries in a company
cut :: Company -> Company
cut (n, es) = (n, cutEmployees es)

-- | Cut salaries for lists of employees
cutEmployees :: [Employee] -> [Employee]
cutEmployees [] = []
cutEmployees (e:es) = cutEmployee e : cutEmployees es

-- | Cut the salary of an employee in half
cutEmployee :: Employee -> Employee
cutEmployee (n, a, s) = (n, a, s/2)

-- | Illustrative function applications
main = do
  print (total sampleCompany)
  print (total (cut sampleCompany))
```

Relationships

In the interest of maintaining a very simple simple beginner's example, the present contribution is the only contribution which does not commit to modularization, packaging, unit testing. See Contribution:haskellEngineer for a modularized and packaged variation with also unit tests added.

Architecture

The contribution only consists of a single module "Main.hs" which includes all the code as shown above.

Usage

See a designated README.

Metadata

- Language:Haskell
- Language:Haskell 98
- Technology:GHCi
- Feature:Flat company
- Feature:Closed serialization
- Feature:Total
- Feature:Cut
- Contributor:rlaemmel
- Theme:Starter
- Theme:Haskell introduction
- Theme:Haskell data

Software system

Headline

A system of intercommunicating software components

Illustration

The @project is obviously concerned with one particular system: the@system which is actually an imaginary software system that is built time and again in different ways by the various contributions of the project. The @system it is a human resource management system. Thus, it deals, for example, with payroll-related management aspects in a company.

Citation

(http://en.wikipedia.org/wiki/Software_system, 14 April 2013)

A software system is a system of intercommunicating components based on software forming part of a computer system (a combination of hardware and software). It "consists of a number of separate programs, configuration files, which are used to set up these programs, system documentation, which describes the structure of the system, and user documentation, which explains how to use the system".

Metadata

- http://en.wikipedia.org/wiki/Software_system
- Vocabulary:Software engineering
- Concept

Feature:Cut

Headline

Cut the salaries of all employees in half

Description

For a given company, the salaries of all employees are to be cut in half. Let's assume that the management of the company is interested in a salary cut as a response to a financial crisis. Clearly, any real company is likely to respond to a financial crisis in a much less simplistic manner.

Motivation

The feature may be implemented as a transformation, potentially making use of a suitable transformation or data manipulation language. Conceptually, the feature corresponds to a relatively simple and regular kind of transformation, i.e., an iterator-based transformation, which iterates over a company' employees and updates the salaries of the individual employees along the way. It shall be interesting to see how different software languages, technologies, and implementations deal with the conceptual simplicity of the problem at hand.

Illustration

The feature is illustrated with a statement in Language:SQL to be applied to an instance of a straightforward relational schema for companies where we assume that all employees belong to a single company:

```
UPDATE employee
 SET salary = salary / 2;
```

The snippet originates from Contribution:mySqlMany.

Relationships

- See Feature:Total for a query scenario instead of a transformation scenario.
- In fact, Feature:Total is likely to be helpful in a *demonstration* of Feature:Salary cut.
- The present feature should be applicable to any data model of companies, specifically Feature:Flat company and Feature:Hierarchical_company.

Guidelines

- The *name* of an operation for cutting salaries thereof should involve the term "cut". This guideline is met by the above illustration, if we assume that the shown SQL statement is stored in a SQL script with name "Cut.sql". Likewise, if OO programming was used for implementation, then the names of the corresponding methods should involve the term "cut".
- A suitable *demonstration* of the feature's implementation should cut the salaries of a sample company. This guideline is met by the above illustration, if we assume that the shown SQL statement is executed on a database which readily contains company data. Queries according to Feature:Total may be used to compare salaries before and after the cut. All such database preparation, data manipulation, and query execution should preferably be scripted. By contrast, if OO programming was used, then the demonstration could be delivered in the form of unit tests.

Metadata

- http://www.thefreedictionary.com/salary+cut
- Functional requirement
- Transformation
- Type-preserving transformation
- Iterator-based transformation
- Optional feature
- Type-preserving transformation

Feature:Flat company

Headline

Support companies as plain collections of employees

Description

Flat companies are sufficiently described by the base feature Feature:Company.

Motivation

The data model for flat companies is supposed to provide a simple (in fact, trivial) starting point for contributions. Despite its simplicity, the data model readily exercises some data modeling facets: basic types such as strings and floats, compound types based on tuples or records, mandatory as opposed to optional properties, and list-based containers.

Illustration

The feature is illustrated with a data model in Language:Haskell; type synonyms instead of algebraic data types are used to emphasize the fact that no recursive are defined here:

```
type Company = (Name, [Employee])
type Employee = (Name, Address, Salary)
type Name = String
type Address = String
type Salary = Float
```

A sample company takes the following form:

```
( "Acme Corporation",
  [
    ("Craig", "Redmond", 123456),
    ("Erik", "Utrecht", 12345),
    ("Ralf", "Koblenz", 12342),
    ("Ray", "Redmond", 234567),
    ("Klaus", "Boston", 23456),
    ("Karl", "Riga", 2345),
    ("Joe", "Wifi City", 2344)
  ]
)
```

Company data is a pair consisting of the company name and a list of employees. Each employee has a name, an address, and a salary.

These snippet originate from Contribution:haskellList.

Relationships

- See Feature:Hierarchical_company for hierarchical, i.e., non-flat companies.
- The features Feature:Flat company and Feature:Hierarchical_company are mutually exclusive and either of them must be selected for any realization of the 101system.
- Several features cannot be usefully combined with Feature:Flat company. For instance, Feature:Depth for the computation of the nesting depth of departments makes no sense without (nested) departments.

Guidelines

- The terms "company", "employee", "manager", "name", "address", and "salary" should appear as part of the abstractions that realize the corresponding ingredients of the data model.
- A *sample company* should be described and processed in some ways, depending on what other features are implemented.

Metadata

- http://en.wikipedia.org/wiki/Hierarchical organization
- Data requirement
- Feature:Company
- Data modeling

Feature:Total

Headline

Sum up the salaries of all employees

Description

The salaries of a company's employees are to be summed up. Let's assume that the management of the company is interested in the salary total as a simple indicator for the amount of money paid to the employees, be it for a press release or otherwise. Clearly, any real company faces other expenses per employee, which are not totaled in this manner.

Motivation

The feature may be implemented as a query, potentially making use of a suitable query language. Conceptually, the feature corresponds to a relatively simple and regular kind of query, i.e., an iterator-based query, which iterates over a company' employees and aggregates the salaries of the individual employees along the way. It shall be interesting to see how different software languages, technologies, and implementations deal with the conceptual simplicity of the problem at hand.

Illustration

Totaling salaries in SQL

Consider the following Language:SQL query which can be applied to an instance of a straightforward relational schema for companies. We assume that all employees belong to a single company; The snippet originates from Contribution:mySqlMany.

```
SELECT SUM(salary) FROM employee;
```

Totaling salaries in Haskell

Consider the following Language:Haskell functions which are applied to a simple representation of companies.

```
-- Total all salaries in a company
total :: Company -> Float
total = sum . salaries

-- Extract all salaries in a company
salaries :: Company -> [Salary]
salaries (n, es) = salariesEs es

-- Extract all salaries of lists of employees
salariesEs :: [Employee] -> [Salary]
salariesEs [] = []
salariesEs (e:es) = getSalary e : salariesEs es

-- Extract the salary from an employee
getSalary :: Employee -> Salary
getSalary ( ,  , s) = s
```

Relationships

- See Feature:Cut for a transformation scenario instead of a query scenario.
- See Feature:Depth for a more advanced query scenario.
- The present feature should be applicable to any data model of companies, specifically Feature:Flat company and Feature:Hierarchical_company.

Guidelines

- The *name* of an operation for summing up salaries thereof should involve the term "total". This guideline is met by the above illustration, if we assume that the shown SQL statement is stored in a SQL script with name "Total.sql". By contrast, if OO programming was used for implementation, then the names of the corresponding methods should involve the term "total".
- A suitable *demonstration* of the feature's implementation should total the salaries of a sample company. This guideline is met by the above illustration, if we assume that the shown SQL statement is executed on a database which readily contains company data. All such database preparation and query execution should preferably be scripted. Likewise, if OO programming was used, then the demonstration could be delivered in the form of unit tests.

Metadata

- Optional feature
- Functional requirement
- Aggregation

Information:Contact

Headline

How to contact 101 HQ

Description

You can reach the headquarter via email:101companies@gmail.com. If you are an actual or a prospective contributor, please feel very much encouraged to get in touch. You may also reach out on Twitter, Tumblr, and YouTube. If you are student in a course that leverages 101, please consider kindly to leverage means of communication as you were instructed, but your input is appreciated by all means.

Metadata

- Namespace:Information

Technology:Haskell platform

Headline

A collection of tools and libraries for program development in Haskell

Metadata

- http://hackage.haskell.org/platform
- Platform
- Namespace:Technology

Functional programming

Headline

The functional programming paradigm

Illustration

Consider the following definition of the factorial function in Language:Haskell:

```
-- A recursive definition of the factorial function
factorial n =
  if n==0
    then 1
    else n * factorial (n-1)
```

This definition describes the computation of factorial in terms of basic arithmetic operations ("functions") and the recursive application of the factorial function itself. There are no variables that are assigned different values over time. This situation is representative of the functional programming paradigm.

For comparison, consider the following definition of the factorial function in Language:Java:

```
 // An imperative definition of the factorial function
 public static int factorial(int n) {
   int result = 1;
   for (int i=n; i>1; i--)
     result = result * i;
   return result;
 }
```

A *result* variable is used in a loop to aggregate the product. Also, the loop uses a variable *i* to iterate from *n* down to *1*. Arguably, the recursive formulation is also straightforward in Java, but Java with its emphasis on variables and assignment as well as mutable data structures and encapsulation of state in objects does not encourage functional programming.

Citation

(http://en.wikipedia.org/wiki/Functional_programming, 14 April 2013)

In computer science, functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state. ... Functional programming has its roots in lambda calculus, a formal system developed in the 1930s to investigate computability, the Entscheidungsproblem, function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus.

Metadata

- http://en.wikipedia.org/wiki/Functional_programming
- Programming paradigm
- Vocabulary:Programming
- Lambda calculus

Language:Haskell

Headline

The functional programming language Haskell

Details

There are plenty of Haskell-based contributions to the101project. This is evident from corresponding back-links. More selective sets of Haskell-based contributions are organized in themes: Theme:Haskell data, Theme:Haskell potpourri, and Theme:Haskell genericity.

Metadata

- http://www.haskell.org/
- http://en.wikipedia.org/wiki/Haskell_(programming_language)
- Functional programming language

Factorial

Headline

The product 1 * ... * *n* for a given natural number *n*

Illustration

We face an algorithmic problem because the factorial can be computed by a (simple) algorithm. Assume that*n* is a natural number. Then, the following steps compute the factorial of *n*:

- Initialize a variable *r* with 1.
- Repeat the following step until *n* equals *0*:
  - Assign *n***r* to *r*.
  - Decrement *n*.
- Return *r*.

Here is also an implementation in Language:Java:

```
// An imperative definition of the factorial function
public static int factorial(int n) {
  int result = 1;
  for (int i=n; i>1; i--)
    result = result * i;
  return result;
}
```

Here is also an implementation in Language:Haskell:

```
-- A recursive definition of the factorial function
factorial n =
  if n==0
    then 1
    else n * factorial (n-1)
```

See also the following collection of implementations:

http://www.willamette.edu/~fruehr/haskell/evolution.html

Citation

(http://en.wikipedia.org/wiki/Factorial, 21 April 2013)

In mathematics, the factorial of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n. For example,

5! = 5 * 4 * 3 * 2 * 1 = 120

The value of 0! is 1, according to the convention for an empty product.

Metadata

- Algorithmic problem
- http://en.wikipedia.org/wiki/Factorial

Function application

Headline

Apply a function to an argument

Description

See the concept of function.

Metadata

- Vocabulary:Functional programming
- http://en.wikipedia.org/wiki/Function_application
- Concept

Compiler

Headline

A program that transforms source code into object code

Illustration

Let's assume that we want to compile theLanguage:Java variation on the Hello world program.

That is, the source code of the program is available as a file "HelloWorld.java".

Compilation can be issued in different ways. For instance, anIDE may take care of it automatically.

For clarity, we perform compilation at the command line:

```
$ ls
HelloWorld.java
$ javac HelloWorld.java
$ ls
HelloWorld.class HelloWorld.java
```

Thus, the compiler produced the object code, in fact,bytecode, in the file "HelloWorld.class".

The byte code can be executed as follows:

```
$ java HelloWorld
Hello, world!
```

The byte code appears to be interpreted, i.e., "java" appears to be aninterpreter for byte code. In fact, "java" performsjust-in-time compilation so that byte code is translated on the fly to machine code, which is then eventually executed (so to say interpreted) by the actual machine.

Metadata

- Vocabulary:Programming
- http://en.wikipedia.org/wiki/Compiler
- Interpreter
- Concept

Linear search

Headline

Solve the search problem by iterating over the input list

Description

Consider the search problem, i.e., the problem of determining whether a given value occurs in a given list. This problem is an algorithmic problem, i.e., it can be solved by an algorithm, e.g., by linear search.

Semi-formally, linear search can be described by an algorithm as follows:

- Given is a list *l* and a value *v* of the element type of *l*.
- Perform the following steps to search *v* in *l*:
  - Initialize an index variable *i* to 0 (to refer to the first element of *l*, if any).
  - Repeat the following steps until a result is returned.
    - Return *False*, if *i* equals the length of *l*.
    - Return *True*, if *l* stores *v* at the index *i*.
    - Increment *i*.

Please note that this formulation also expresses that the element type must admit comparison for equality.

Illustration

Linear search in Haskell

```
-- Polymorphic linear search
search :: Eq a => [a] -> a -> Bool
search [] _ = False
search (x:xs) y = x==y || search xs y
```

The type of the *search* function is polymorphic in that admits arbitrary element types for as long as equality ("Eq") is supported for the type.

The implemented search function can be applied as follows:

```
-- Illustrate linear search
main = do
  let input = [2,4,3,1,4]
  print $ search input 1 -- True
  print $ search input 5 -- False
```

Citation

(http://en.wikipedia.org/wiki/Linear_search, 21 April 2013 with Knuth's "The Art of Computer Programming" credited for citation)

In computer science, linear search or sequential search is a method for finding a particular value in a list, that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

Metadata

- http://en.wikipedia.org/wiki/Linear_search
- Search algorithm
- Search problem
- Binary search

Data structure

Headline

A particular way of storing and organizing data in a computer

Illustration

See linked lists as a simple example of animperative data structure.

See immutable lists as a simple example of afunctional data structure.

Metadata

- Vocabulary:Data
- Vocabulary:Programming
- http://en.wikipedia.org/wiki/Data_structure
- Concept

Immutable list

Headline

A form of list without basic operations for mutation

Description

An immutable list is a data structure for lists (ordered sequences) of elements of a common type. An immutable list can be manipulated in a basic sense like this:

- Observations
  - Determine whether the list is *null* (i.e., empty).
  - Retrieve the *head* of the list, if it exists (i.e., the first element of the list)
  - Retrieve the *tail* of the list, if it exists (i.e., the rest or all but the head of the list).
- Construction
  - Construct a *nil* list (i.e., the empty list).
  - Construct a *cons* list (i.e., a non-empty list from given head and tail).

Notably, there are no operations like these:

- Adding an element in ways other than by using*cons*.
- Removing an element in ways other than by using*tail*.

Such operations are not available because of immutability (lack of support of changes, i.e., mutations). Such operations may be still encoded, but only by means of copying elements from a given list to a new list.

Illustration

Immutable lists in Haskell

Here are some lists with an increasing number of elements:

```
[]
[1]
[1,2]
[1,2,3]
```

We showed convenience notation for list construction. Fundamentally, lists are constructed from two constructor functions: *nil* (square brackets) and *cons* (colon). Let us construct the same lists with the fundamental constructors:

```
[]
1:[]
1:(2:[])
1:(2:(3:[]))
```

These are the functions to retrieve the head and the tail of a list:

```
head :: [a] -> a
tail :: [a] -> [a]
```

(In reality, these functions have more general types, but let's simplify things here.)

Here are some applications of *head* and *tail*:

```
> head [1,2]
1
> tail [1,2]
[2]
```

Here is how we test a list to be empty:

```
null :: [a] -> Bool
```

For instance:

```
> null []
True
> null [1,2]
False
```

Further operations on lists can be expressed in terms of the operations described so far. Let us define an operation*snoc* for adding an element at the end of a list. (*snoc* is inverse of *cons* in that cons adds an element at the start of a list.) Here is the function definition:

```
snoc :: [a] -> a -> [a]
snoc [] x = [x]
snoc (x:xs) y = x : snoc xs y
```

Here is an illustrative function application:

```
> snoc [1,2] 3
[1,2,3]
```

The function definition for *snoc* is representative for functions on lists in that we leveragepattern matching with two cases: one for the empty list, another one for nonempty lists. Further, we leverage recursion: the function *snoc* is defined in terms of itself; the first equation is the base case for recursion.

Metadata

- List
- Functional data structure
- Immutable data type
- Linked list

Program

Headline

An executable software artifact that solves a certain problem

Description

According to a classic definition, a program is (Document:Principles of information systems) "a sequence of instructions written to perform a specified task with a computer". This style of definition is possibly too much focused on an imperative view of programming.

More intuitively, more inclusively, and shorter:**a program is an executable software artifact that solves a certain problem (that is amenable to automation on a computer). For instance, a program may solve an algorithmic problem.**

A program may count as a "small" software system or an (executable) software component. A "proper" software system or component typically comprises of multiple software artifacts that may be elements of different software languages, may or may not be elements of programming languages, may reside at different levels of abstraction, and may interact in various ways.

Strictly speaking, a program, as far as this term is used in practice, may very well also break down into multiple software artifacts because of, for example, modular programming. Thus, the line between programs and software systems or (executable) software components is somewhat blurred.

Illustration

See the Hello world program for a very simple program.

Metadata

- http://en.wikipedia.org/wiki/Computer program
- Software artifact
- Vocabulary:Programming
- Vocabulary:Software engineering

Programming language

Headline

A software language in which programs are written

Illustration

Have a look at Hello world programs in different programming languages.

Metadata

- http://en.wikipedia.org/wiki/Programming_language
- Vocabulary:Programming
- Vocabulary:Software engineering
- Software language

Information:Download

Headline

How to download contributions of the101project

Description

Download 101companies contributions

If you want to try out contributions, there are these means of downloading them.

ZIP file as linked on 101wiki page

1. Go to the 101wiki page for the contribution.
2. Select the "Code tab" and the item "Download .zip".

In this manner, one downloads the zipped directory for the contribution. It should be noted that some contributions may rely on artifacts that are not located in the contribution directory, that are located instead in other directories of the same repo. The following options are more generally applicable.

ZIP file as linked on GitHub page

1. Go to the 101wiki page for the contribution.
2. Select the "Code tab" and the item "View code at GitHub".
3. Go to the root of the repo.
4. Select the "ZIP" button to download the entire repo.

One will only get to see the "ZIP" button when position at the root of the directory.

git clone the GitHub repo

1. Go to the 101wiki page for the contribution.
2. Select the "Code tab" and the item "View code at GitHub".
3. Go to the root of the repo.
4. Select "Git Read-Only", if necessary.
5. Copy the URL next to the button into the clipboard.
6. Run "git clone <url></url>" on your system.

Here it is assume that some git client is available on the system. In fact, the last step, i.e., the "git clone" command, is given in the form as appropriate with a typical git client using the command line rather than a GUI.

Metadata

- Namespace:Contribution
- Namespace:Information

Information:Translate

Headline

How to translate terms used on 101wiki

Description

Specifically, concepts according to Namespace:Concept may need to be translated. Any standard means of translation can be used, of course, but Wikipedia can also be leveraged directly, whenever a page is linked to Wikipedia. For instance, Sorting algorithm links to http://en.wikipedia.org/wiki/Sorting_algorithm (using, in fact, an Property:Identifies), which in turn is linked to the German Wikipedia page http://de.wikipedia.org/wiki/Sortierverfahren. Thus, one cannot just translate terms, in fact, one can read about concepts in the preferred language, without using any unstructured means of web search.

Metadata

- Namespace:Information

Technology:GHC

Headline

A Haskell compiler

Details

GHC stands for Glasgow Haskell Compiler. Strictly speaking, GHC is more than just a compiler; it is a distribution that contains other components of language implementation. In particular, the distribution also contains Technology:GHCi which is an interpreter.

Metadata

- Compiler
- Technology:Haskell Platform
- http://www.haskell.org/ghc/
- Namespace:Technology

Technology:GHCi

Headline

The Language:Haskell interpreter as part of Technology:GHC

Metadata

- Interpreter
- Namespace:Technology

Information:Wikipedia

Headline

How Wikipedia is used on 101wiki

Description

Wikipedia is considered a key resource on 101wiki. Whenever possible, entities on 101wiki are linked to suitable Wikipedia pages. To this end, the Property:Identifies is used, when the reference Wikipedia page is judged to deal with exactly the entity (e.g., language, technology, or concept) at hand. The Property:LinksTo is used for semantically weaker links. Of course, 101wiki also links to other resources (including wikis), but Wikipedia is referenced as much as possible.

Further, Wikipedia is also a popular source for citation. That is, text from Wikipedia pages may end up in the "Citation" section of 101wiki pages.

Illustration

See Sorting algorithm for a 101wiki pages that indeed links to Wikipedia for the relevant concept and also quotes from the linked Wikipedia page in the "Citation" section.

Metadata

- Namespace:Information
